Paper 22-26

A Pattern for Dynamic Web Content: SAS® Server Pages

Jack E. Fuller, Technology Professionals Corporation, Grand Rapids, MI

ABSTRACT

As more and more applications move to a web environment, it becomes increasingly critical to have a reusable and tested design to build applications. This paper presents a pattern that can be used by the advanced application developer to implement a consistent, tested and extensible design for using SAS/AF in collaboration with other technologies such as HTML and JavaScript to dynamically generate web pages.

MOTIVATION

"The purpose of life is a life of purpose."

Robert Byrne

After developing several projects that used SAS to dynamically generate web pages, it became increasingly apparent that a pattern was repeating itself in each application. In this context, a *pattern* is defined as a proven solution to a recurring problem. In *A Pattern Language*, Christopher Alexander wrote that a pattern

> describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

This paper presents a proven, extensible, and reusable solution to the problem of generating dynamic web pages while using the SAS system.

APPLICABILITY

"There is nothing so easy to learn as experience and nothing so hard to apply."

Josh Billings

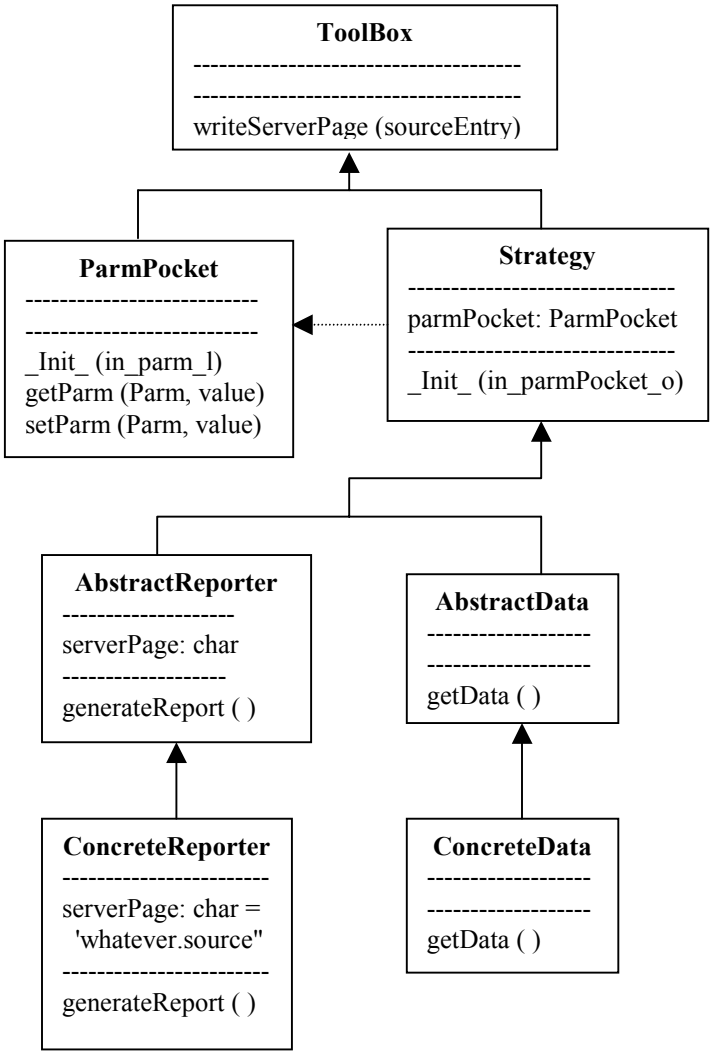
Use SAS Server Pages when

- using SAS to dynamically generate web pages
- maintaining different methods of data access
- generating different custom reports
- coordinating different technologies (e.g. HTML, JavaScript, Oracle, SAS)
- providing an interactive environment via the web

STRUCTURE

"A road map tells you everything except how to refold it. "

Isaac Asimov



"I either want less corruption, or more chance to participate in it."

Ashleigh (Ellwood) Brilliant

- The **SAS Participants** consist of those components implemented using SAS
  - The **Greeter** is a SAS SCL entry that
    - welcomes the user to the SAS system
    - drives the main flow of program execution
  - The **Toolbox** is an abstract class that
    - maintains common methods used in different applications
    - contains the *writeServerPage* method that will be reused across many different applications
  - The **Parameter Pocket** is a concrete class that
    - encapsulates parameter information
    - defines an interface for accessing parameters
    - stores parameters in its *pocket* and then retrieves them as requested
  - The **Data** participant is an abstract class that
    - encapsulates data access (e.g. SAS datasets, Oracle® tables, Sybase® tables, etc.)
    - defines an interface for retrieving or updating stored information
  - The **Reporter** is an abstract class that
    - encapsulates the report layout
    - defines an interface for report generation
    - calls the *writerServerPage* method and passes it the value of its *serverPage* instance variable
  - The **SAS Server Page** is a SAS source entry that
    - stores the report template used to generate the page that will be returned to the browser
    - marks the portions of the page which need to be dynamically generated are indicated by a special mark up language (i.e. the *SAS Server Page* mark up)
    - provides an environment for resolving *SAS Server Page* mark up via method calls
- The **Web Participants** consist of those web-based components implemented outside the SAS system
  - The **Redirector** is a web page that
    - provides the web-based initial entry to the application
    - implements a mechanism for overcoming unwanted browser caching
  - The **Source Web Files** consist of all other web-based source files
    - Static non-SAS source files (e.g. HTML, JavaScript, etc.)

COLLABORATIONS

"The ultimate test of a relationship is to disagree but hold hands."

Alexander Penney

1. The user points their browser to the *Redirector*
2. The *Redirector* then
   a. points to the SAS broker and specifies that the *Greeter* should be executed
   b. adds a parameter with a random value to the URL of the passed to the *Greeter*
3. The *Greeter* serves as
   a. the entry point to the SAS system
   b. the controlling application driver
4. The *Greeter* drives the application by
   a. Instantiating the *ParmPocket* class
   b. Instantiating the appropriate *ConcreteData* class
   c. Instantiating the appropriate *ConcreteReport* class
   d. Instructing the newly created *ConcreteReport* object to generate the report
   e. Cleaning up the objects and returning control to the browser
5. The *ConcreteReporter* generates the appropriate page view by
   a. Instructing the *ConcreteData* object to retrieve all of the data
   b. Calling the *writeServerPage ()* method on itself and passing it the value of its *serverPage* instance variable

CONSEQUENCES

"Sometimes when I consider what tremendous consequences come from little things -- I am tempted to think there are no little things."

Bruce Barton

- It supports a common interface and code repository for different data access implementations while simultaneously allowing each data retrieval to have its own idiosyncrasies.
- It supports a common interface and code repository for different report generation and/or data gathering implementations while simultaneously allowing each report to maintain its own idiosyncrasies.
- The actual *ConcreteReporter* and *ConcreteData* classes may be varied at run-time. Usually, the actual classes to instantiate are determined by examining the variables passed to the *Greeter* and then instantiating the appropriate classes.
- Generic code is captured in the *ParmPocket* and *Toolbox* classes so that it can be reused across applications. In the following sample code, *Strategy.scl* could also be reused across applications.
- Storing the parameter information in the *ParmPocket* allows simpler and more robust method signatures. For example, if it is determined that a specific method needs additional information, that data can be added to the *ParmPocket* with no further impact to any other methods.

IMPLEMENTATION

"Rule three hundred of obscure leadership: if it's your idea, you get to implement it."

Leland Exton Modesitt, Jr.

- *How are the actual Reporter and Data sub classes derived from the input parameters?* Methods for determining the correct classes to instantiate include such techniques as table lookup and hashing. The following code samples use SAS formats to provide the table lookup.
- *What form should the special mark up language take?* This is really up to the individual programmer; however, in order to have a robust implementation, one should consider quite strongly the use of multi-character delimiters.
- *When is the data retrieved?* Since the *writeServerPage* is nothing more than a glorified *data _null_*, all of the data must retrieved prior to calling *writeServerPage*.

SAMPLE CODE

"There are two ways to write error free programs. Only the third one works."

Annonymous

The Redirector

This HTML implementation of *the Redirector*

- redirects the browser to a new location
- maintains the pre-existing window title
- adds the X parameter to the URL and gives it a random value in order to overcome unwanted caching

Redirector.html

```
<HTML>
<HEAD>
<TITLE>ROPCP</TITLE>
<SCRIPT type="text/javascript">
  var oldTitle = document.title;
  this.location.replace (
    location.protocol + "//"
    + location.host
    + "/scripts/broker.exe"
    + "?_SERVICE=default"
    + "&_PROGRAM="
      + "Web.MyApp.Greeter.scl"
    + "&X=" + Math.random ());
  document.title = oldTitle;
</SCRIPT>
</HEAD>
<BODY>
</BODY>
</HTML>
```

The Greeter

This implementation of the Greeter

- uses SAS formats to Determine the correct classes to be instantiated
- performs the instantiations of the different classes
- instructs the *Reporter* to generate the new page
- performs any necessary clean up

Note the use of automatic delegation to obtain the parameter values (see **The Strategy Class** below for additional explanation).

Greeter.scl

```
entry in_arg_l 8;

length theClass $200;

declare
  SASHelp.FSP.Class.class
    parm_c report_c data_c,
  Tools.Parm.Parm.class
    parm_O
  Strategy.Report.MyRpt.class
    report_o
  Strategy.Data.MyData.class
    data_o;

INIT:
  /* Instantiate the parm object */
  parm_c = loadclass
    ('Tools.Parm.Parm.class');
  parm_c._new_(parm_o, in_arg_l);

  /* Create the reporting object */
  theClass = put (
    _self_.getParm ('X'),
    $X2Rpt.);
  report_c = loadclass (theClass);
  report_c._new_(report_o, parm_o);

  /* Create the data object */
  theClass = put (
    _self_.getParm ('y'),
    $Y2Dat.);
  report_c = loadclass (theClass);
  data_c._new_(data_o, parm_o);

  /* Create the report */
  report_o.generateReport(data_o);

  /* Clean up */
  data_o._term_();
  report_o._term_();
  parm_o._term_();
return;
```

The Strategy Class

The *Strategy* class is used to capture that functionality which is common to both the *Data* and the *Reporter* classes. In this example, the *_Init_* method is used to remember the current instance of the *ParmPocket* class. In addition, in order to take advantage of SAS' automatic delegation facility, the *parm_o* instance variable is defined as a delegate. This means that a *Strategy* object can get or set the value of a parameter by simply calling *getParm* or *setParm* on themselves.

Strategy.scl (_Init_ method only)

```
length _method_ $200;

_method_ = _method_;
_self_   = _self_

declare
   Strategy.class   _self_;

_INIT_:     /* _INIT_ method */
method
   in_parm_o         8
;
   call super (_self_, _method_);

   /* Remember the parm object */
   _self_.parm_o = in_parm_o;
endmethod;
```

The Reporter Class

The *Reporter* class is used to actually control the generation of the new page. This example does the following:

- implements the *generateReport ( )* method
- implements the *writeData ( )* method; it would also implement any other methods which may be referenced in the *serverPage*
- holds the *serverPage* instance variable

Report.scl
(generateReport and writeRows methods only)

```
length _method_ $200;

_method_ = _method_;
_self_   = _self_;

genRpt:  /* generateReport method */
method
  in_data_o       8
  ;
  /* Retrieve the data */
  in_data_o.getData ();

  /* Generate the report */
  _self_.writeServerPage (
    _self_.serverPage);
endmethod;

writRows: /* writeRows method */
method
;
  /* Open the data */
  dsid = open (
    _self_.getParm ('data_01'));

  /* Loop through the data */
  rc = fetch (dsid);
  do while (rc=0);
    c1 = getvarn (dsid,
      varnum (dsid, 'col1'));
    c2 = getvarn (dsid,
      varnum (dsid, 'col2'));
    c3 = getvarn (dsid,
      varnum (dsid, 'col3'));

    /* Write the table rows */
    submit;
      put "<TR>"
        "<TD>&c1</TD>"
        "<TD>&c2</TD>"
```

```
        "<TD>&c3</TD>"
        "</TR>";
    endsubmit;

    rc = fetch (dsid);
  end;

  /* Clean up */
  dsid = close (dsid);
endmethod;
```

The Data Class

The *Data* class is used to retrieve the data. In most situations, the actual retrieval will be considerably more complex and will often create more than one dataset. The name of the newly created dataset is stored in the *ParmPocket*.

Data.scl (getData method only)

```
length _method_ $200;

_method_ = _method_;
_self_   = _self_;

getData: /* getData method */
method
;
  /* Retrieve the data */
  submit continue sql;
    create table data_01 as
      select col1, col2, col3
      from myData.whatever;
  endsubmit;

  /* Store the dataset name */
  _self_.setParm('dsname',
    'data_01');
endmethod;
```

The SAS Server Page

This *SAS Server Page* consists of HTML/JavaScript that has been additionally marked up with our custom *SAS Server Page* markup (i.e. "~{" and "}~"). If the page needed to reference any static defined JavaScript files, these could be included via a <SOURCE> tag in the HTML header.

ServPage.source

```
<HTML>
<HEAD>
  <TITLE>SAS Server Page</TITLE>
</HEAD>
<BODY>
  <TABLE>
    <THEAD>
      <TR>
        <TH>Column One</TH>
        <TH>Column Two</TH>
        <TH>Column Three</TH>
      <TR>
    </THEAD>
    <TBODY>
      ~{writeRows}~
    </TBODY>
  </TABLE>
</BODY>
</HTML>
```

The ParmPocket

An actual example of the code in the *ParmPocket* is not provided since it is quite straightforward. At a minimum, it should contain the following methods (in SAS 7.00 and later, this would include versions of the *getParm* and *setParm* methods for each value type):

- *_Init_ (in_parm_l)* -- this method provides the initial names and values for the parameters
- *getParm (parm, value)* -- this method retrieves the value of the specified parameter; a robust implementation should also handle the situation when a parameter does not exist
- *setParm (parm, value)* -- this method sets the value of a parameter; additionally, it should also add a new parameter when the name is not found

The ToolBox

The *writeServerPage( )* method is really nothing more than an exercise in reading, parsing, and then writing a text file that happens to be stored in a SAS catalog. There are several things to consider when implementing this method:

- Remember to use multiple-character delimiters; the obvious delimiters have already been taken by web programming languages such as HTML and JavaScript
- If programming for SAS 6.12 or earlier, remember to allow for lines of text which are greater than 200 characters in length
- It is often useful to remove any null characters ('00'x) from the source text prior to parsing
- Specifying quotes in the source file can prove troublesome; to make things simpler, require that the source file use doubled quotes as necessary (e.g. " becomes "")

- Use the macro function *%nrstr ( )* in any *put* statements to avoid unwanted macro resolutions
- When encountering a method which must be resolved:
  - Simplify the markup language by not allowing any marked up methods to specify parameters directly; instead, specify any parameters indirectly by
    - calling a method that doesn't take any parameters
    - which in turn makes the call to the method which takes parameters
  - Do not try to have the methods return the desired text; it is much simpler and much more robust to simply have them issue their own *put* statements

CONCLUSION

This paper has presented a pattern for creating web applications while use SAS. However, the idea of documenting good, tested design can be applied to any programming language.

Don't try to solve every problem from the beginning. Instead, build on you experience and on the experience of others. Document your own design patterns and investigate the design patterns of others.

REFERENCES

Alexander, Christopher, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, Shlomo Angel (1977), *A Pattern Language,* New York, NY: Oxford University Press.

Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal (1996) *Pattern-Oriented Software Architecture: A System of Patterns,* Chichester, West Sussex, England: John Wiley & Sons Ltd.

Coplien, James O., Douglas C. Schmidt (1995), *Pattern Languages of Program Design*, Reading, MA: Adison-Wesley Publishing Company.

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides (1995), *Design Patterns: Elements of Reusable Object-Oriented Software,* Reading, MA: Addison-Wesley.

Harrison, Neil, Brian Foote, Hans Rohnert (2000), *Pattern Languages of Program Design 4*, Reading, MA: Adison-Wesley Longman, Inc..

Martin,Robert, Dirk Riehle, Frank Buschmann (1998), *Pattern Languages of Program Design 3*, Reading, MA: Adison-Wesley Longman, Inc..

SAS Institute Inc (1995*), SAS/AF Software: FRAME Class Dictionary, Version 6, First Edition*, Cary, NC: SAS Institute Inc.

SAS Institute Inc (1994) *SAS Screen Control Language: Reference, Version 6, Second Edition*, Cary, NC: SAS Institute Inc.

Vlissides, John M., James O. Coplien, Norman L. Kerth (1996), *Pattern Languages of Program Design 2*, Reading, MA: Adison-Wesley Publishing Company.

ACKNOWLEDGEMENTS

CONTACT INFORMATION

Jack E. Fuller
Project Manager
Technology Professionals Corporation
1 Ionia SW, Suite 500, Grand Rapids, MI 49503
jefuller@teamtpc.com