# SANDIA REPORT

SAND98–2781
Unlimited Release
Printed February 1999



# Web Application Design Using Server-Side JavaScript

Jeff Hampton and Randall Simons

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000.



Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.



Available to DOE and DOE contractors from
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Prices available from (703) 605-6000
Web site: http://www.ntis.gov/ordering.htm

Available to the public from
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Rd
Springfield, VA 22161

NTIS price codes
Printed copy: A03
Microfiche copy: A01

SAND98-2781
Unlimited Release
Printed February 1999

# Web Application Design Using Server-Side JavaScript

Jeff Hampton
Decision Support Systems - Software Engineering

Randall Simons
Decision Support Systems - Architectures

*Sandia National Laboratories*
P.O. Box 5800
Albuquerque, NM 87185-1138

## Abstract

This document describes the application design philosophy for the Comprehensive Nuclear-Test Ban Treaty Research & Development Web Site. This design incorporates object-oriented techniques to produce a flexible and maintainable system of applications that support the web site. These techniques will be discussed at length along with the issues they address. The overall structure of the applications and their relationships with one another will also be described. The current problems and future design changes will be discussed as well.

# Acknowledgments

The authors would like to thank Matt Chown for his help in developing this architecture. He played an important role in making sure the architecture that was created would allow the web team to build in future changes the customer may require. He was also instrumental in migrating the data from the old Comprehensive Nuclear-Test Ban Treaty Research and Development web site to the new web site using the architecture described in this document.

This work was supported by the United States Department of Energy under Contract DE-AC04-94AL85000. Sandia is a multiprogram laboratory operated by Sandia Corporation, A Lockheed Martin Company, for the United States Department of Energy.

## Contents

## Figures



## Tables



## Examples

## Introduction

The Comprehensive Nuclear-Test Ban Treaty (CTBT) Research & Development (R&D) Web Site (http://www.ctbt.rnd.doe.gov) design team consisting of Randy Simons, Jeff Hampton, and Matt Chown has built what we believe to be a robust set of web applications that utilize server-side JavaScript (SSJS) [ssjs] to support the web site. This web site provides access to data products and other information, especially reports, regarding the CTBT R&D program's full-scope effort to develop technologies and analysis algorithms for monitoring the comprehensive nuclear-test ban treaty. Most of the applications mentioned in this paper are used within the CTBT R&D Coordination Web Site (http://www.ctbt.rnd.doe.gov/coordination). The purpose of the CTBT R&D Coordination Web Site is to facilitate the integration of research products into customer applications, as well as prevent duplication of research efforts. The coordination web site provides the following capabilities:

- A variety of displays of basic information about funded research.
- A summary of the direction, trends, and priorities of CTBT research.
- A "How To..." guide to proposal submittal opportunities and procedures.
- Notification of upcoming meetings of interest to the CTBT R&D community.
- Listings of research products (technical reports and/or electronic products) developed by CTBT R&D contracts.
- A collection and organization of CTBT related Internet links.
- Group and individual communication tools for use amongst CTBT researchers and affiliates.
- Organizations involved in CTBT research and development.

The web development team has made use of object-oriented techniques in developing the SSJS code that are not typically taken advantage of in client-side (browser) JavaScript [csjs] code. The sets of objects we have developed have increased our productivity in maintaining and adding new capabilities to the current code.

The CTBT R&D Web Site applications we have developed involve 3 types of HTML pages. These are:

1. List Page - This page is essentially a query and list type of page. The user chooses the search criteria and is presented with a tabular list of records matching the query.
2. Data Page - This page displays a single record in its entirety in a read-only HTML view.

3. Form Page - This page displays a single record in its entirety as a form allowing the user to edit the information in the record.

We have been able to make use of this framework and apply it to different applications that use database records about people, research contracts, documents, web links, organizations, and others. This structure has allowed us to prototype new applications without a lot of extra coding. This, of course, results in a major cost savings in terms of time and money for our United States Department of Energy (DOE) customer.

## Data Representations

One can look at a web application as consisting of multiple representations of the same data, and the transformations between them. In the simplest case, there is only one representation the developer is concerned with, that is, HTML. The browser takes care of converting it to a screen display. Of course, this limits displays to showing only static information and being non-interactive. Another more flexible alternative would be to use Java, but we ruled that out because some of our users are behind firewalls that do not allow Java to pass.

The next level of complexity is to add interactivity. HTML has <FORM> and <INPUT> tags, but some additional means are required to receive and handle the input. Using CGI to receive the data, one writes programs in languages such as Perl or C++ to process it. Or one can use server-side JavaScript, in which case all the input values are returned as properties of the request object. In either case, we now have a second representation of the data to deal with. We chose to use server-side JavaScript because we liked its general purpose interface to various database management systems. The mapping from HTML to the request object is handled transparently by the server, but creating new HTML from values in the request object has to be programmed by the developer.

If the purpose of adding interactivity is to allow querying and editing of data, as it is in our application, then you need somewhere to store that data. One can not use the request object to store this data if it needs to stick around from one server request to the next, since the request object only exists during the time the request is being processed. Server-side JavaScript provides longer-lived objects, such as the client and project objects, but none of these will survive a restart of the web server. We are using a database to make the data truly persistent. An earlier version of this web site used custom files to store data, but we found this to be inflexible and fragile. By adding a database we have added a third data representation.

The developer now has to implement the transformations from the database into HTML, and from the request object to the database. In reality, adding a database that is accessed using server-side JavaScript adds two data representations: the database tables themselves, and the SQL statements with their resultant cursor objects. We choose to treat these representations as one, since they are so tightly interconnected, and they generally do not restructure the data, just change its format.

Figure 1 shows the architecture of the current system in terms of data representations and the transformations between them. The box labelled "Application" contains all the functionality required to convert between the different representations.

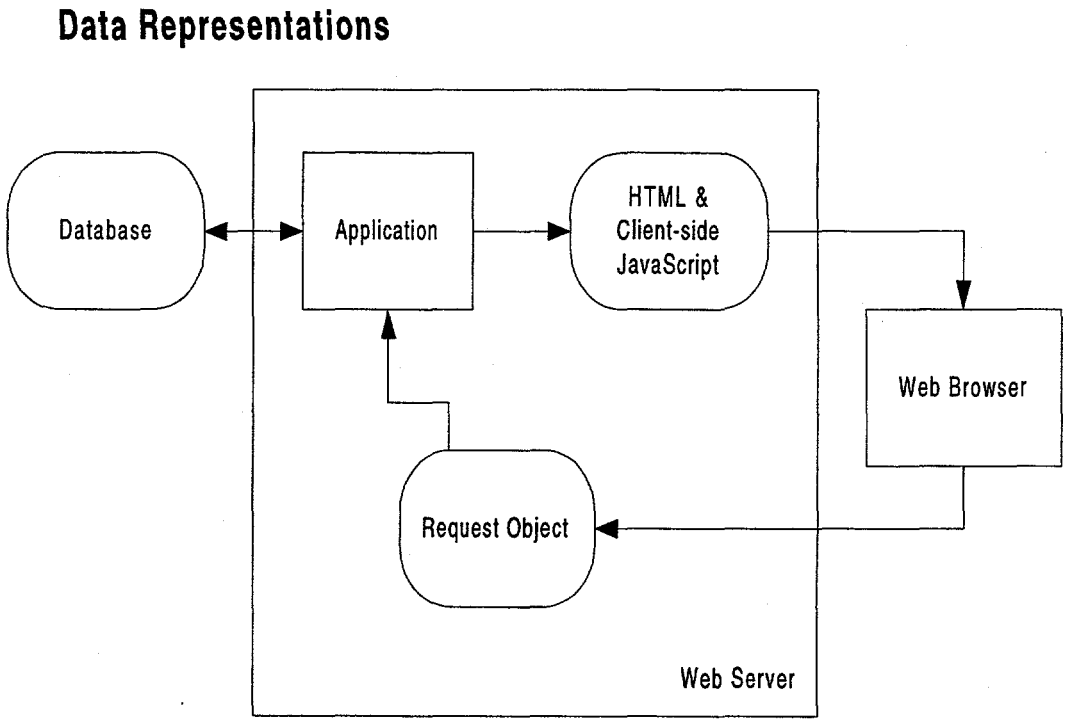


**FIGURE 1.** Data Representation Diagram

## Object-Oriented Techniques in SSJS Design

Our team began building the applications for the web site using a functional library approach. We were trying to build function libraries that would be called from the

different HTML pages we were creating. We quickly realized that application design changes were becoming unmanageable using this approach. We would have to build functions that made certain assumptions about the context in which they were called. These functions would have to rely on the caller to build an "environment" of variables that represented the current state of the page and pass that into every function that was called. It became very cumbersome to keep this "environment" intact across invocations of the page as well as the functions themselves. We decided to take an object-oriented approach to our design. We took the bulk of these library functions and created objects that would use these functions as methods.

This transition from our functional library module architecture to an object-oriented one was relatively straightforward. We took groups of the library functions and built the object "environments" to support these groups of functions. These functions merged nicely into methods of a common object from which these functions could be called. The resulting object and its properties were built based on the parameter needs of the functions that were attached to it as methods. When we were finished grouping functions into methods, and building the object encapsulation, we ended up with a one level hierarchy of objects that we could now use to support our code development.

From this one tier object hierarchy, we gathered all the common methods and properties from some objects and began to take advantage of object inheritance. This consolidated a lot of code from the different objects. Because much of the code now resided in a single place, changing that code was much easier and had more far reaching effects. Since many other objects may depend on that code, changes in the higher levels of the hierarchy would also propagate to all of the subclasses related to that object. In other words, we could affect larger portions of the system by crossing application boundaries and still retain consistency in how a method is implemented.

This change in design began to have immediate benefits. We were able to reuse almost all the code we developed for our first application that involved contract information to quickly build an application to support listing and maintaining information about organizations and people. The parts we couldn't reuse were simply rewritten as method overrides for those objects.

The core JavaScript language supports an object framework [Shafer11/97][Shafer10/97]. This framework is based on the prototype model [jsref]. Objects in Javascript are defined dynamically in terms of their prototype instead of statically at compile time like in Java or C++. In JavaScript you simply define a constructor function that sets the properties and methods of the object and thus the

object is defined. This function encapsulates the initial attributes and behavior of an object. For example:

```
function myObject() {
   this.propA = "Do";
   this.propB = "Re";
   this.propC = "Mi";
}

var o = new myObject();
write(o.propA); // Will print the string "Do".
```

**EXAMPLE 1.** JavaScript object construction.

The big difference between JavaScript and Java or C++ is that the structure such as the properties or methods of an object can change at run-time. Extending Example 1 above:

```
o.propD = "So"; // Is perfectly legal.
write(o.propD); // Will print the string "So".
```

**EXAMPLE 2.** Dynamic properties.

Methods can also be defined in the constructor function of an object. These methods give the object its behavior. Once again extending from Example 1 above:

```
function displaySong() {
   for (prop in this) {
      write(prop + "\n");
   }
}

o.displaySong = displaySong;
o.displaySong();

// Would print each property value on a single
// line.

Prints:
Do
Re
Mi
```

**EXAMPLE 3.** Object methods.

Unfortunately in JavaScript, the visibility of an object's attributes or methods can not be controlled. All properties and methods of an object are public. This means

that any utility methods that are used within other methods of the object will also be visible to your program. There is no way to create private methods for an object. These private methods should not be callable from outside the scope of the object itself. This limitation will change in future versions of the JavaScript specification.

Inheritance in object-oriented design is very important. JavaScript provides a mechanism for inheritance through the prototype object. The prototype object is available for every object in the JavaScript language including user-defined objects. Setting the prototype object property of an object to the object you wish to inherit from creates a link that is used to lookup properties and/or methods that are associated with the new object. An example of this is given in Example 4.

```
function RectArea() {
   return (this.len * this.width);
}

function Rectangle(len, width) {
   this.len = len;
   this.width = width;

   this.area = RectArea;
}

function Square(edge_len) {
   this.len = edge_len;
   this.width = edge_len;
}
Square.prototype = new Rectangle;

var s = new Square(4);
write(s.len); // This would print 4.
write(s.width); // This would print 4 also.
write(s.area()); // This would print 16 since Square
uses the same area method that rectangle has defined
for it.
```

**EXAMPLE 4.** Object inheritance.

Another related issue that involves inheritance is the issue of overriding. In object-oriented systems, overriding a method of an inherited object is an essential capability that must be supported. JavaScript allows the user to easily define an overriding method for an object. Borrowing from Example 4:

```
function sq_area() {
   return (this.edge_len * this.edge_len);
}

function Square(edge_len) {
   this.len = edge_len;
   this.width = edge_len;
   this.edge_len = edge_len;

   this.area = sq_area; // area method override.
}
Square.prototype = new Rectangle;

var sq = new Square(4);
write(sq.area()); // Will print 16 using the sq_area
function.
```

**EXAMPLE 5.** Object method overrides.

As you can see, JavaScript provides all the basic ingredients for an object-oriented design. There are several distinctions that need to be made between JavaScript and class-based languages such as Java or C++. These differences are highlighted in Table 1.

**TABLE 1.** Comparison of class-based languages and JavaScript object frameworks. [jsobj]

| Class-based Languages | JavaScript |
|---|---|
| Class and instance are separate. | All objects are instances. Classes are not defined separately from their instances. |
| Class is defined by a class definition. A class is instantiated using constructor methods. | The constructor function is the class definition. An object is still instantiated by the constructor function. |
| Objects are instantiated using the *new* operator. | Same. |
| Object hierarchies are created by defining them as subclasses in the class definition. | Object hierarchies are created by assigning a *new* object to the prototype object property of the subclass in the subclass' constructor. |

**TABLE 1.** Comparison of class-based languages and JavaScript object frameworks. [jsobj]

| Class-based Languages | JavaScript |
|---|---|
| Properties/methods are inherited from the class chain. | Properties/methods are inherited from the prototype chain. |
| The class definition is static for all instances of a class. No properties/ methods can be added to a class at run-time. | The constructor function defines an initial set of properties/methods for an object. But, properties/ methods can be added dynamically, either to individual objects or to a class of objects. |

## CTBT R&D Web Site Object Design

In developing our object hierarchy, we started out designing objects that would represent the database tables we used to store the web site information. We had another set of objects that we used to represent the information used to display the List Page. These sets of objects were somewhat similar in that they had to represent database information in order to retrieve and manipulate database data. Since these sets were similar, an obvious hierarchy was created to support these objects.

This hierarchy divided into two areas. The first area contains abstract objects that are used to represent fields in a database table to access it via SQL. The second area contains abstract objects that are used to represent columns in a list for the List Page. These two areas are inextricably linked into the hierarchy, however, as shown in Figure 2.

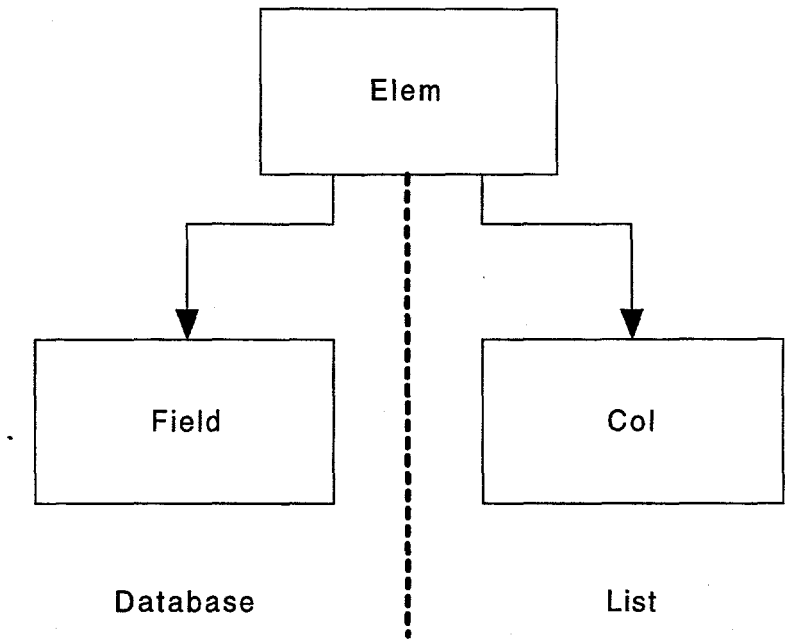


**FIGURE 2.** Component element object hierarchy.

The Elem object encapsulates a single piece of data. The purpose of this object is to maintain some information about that piece of data, such as its database name and descriptive label, and to provide methods for translating that piece of data back and forth between an application format and a database format. The Field object builds on the Elem object by including additional information such as the size of the field and methods to display the field within the context of a Data Page or a Form Page. The Col object builds on the Elem object by including additional information such as whether this field is checked by the user to be listed in the columns of the List Page and methods to create portions of SQL to be generated to display and format the column in the List Page.

The objects shown in Figure 2 are used as components for another set of objects that follow a similar division. An array of Field objects is actually a property of the Table object. An array of Col objects is actually a property of the Columns object. This new set of objects form the basis for the main objects referred to in the server-side code that is seen in the HTML pages. There is a set of abstract objects that represents the database tables by which data is accessed. There is also a set of abstract objects that represents the columnar list presented by the List Page. This hierarchy is shown in Figure 3.

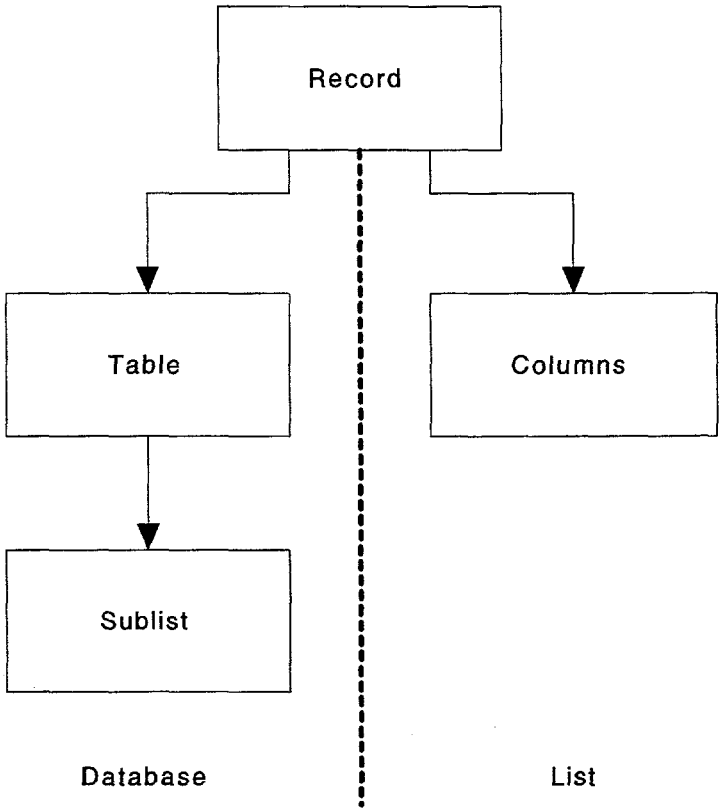


**FIGURE 3.** Major abstract object hierarchy.

The Record object has several properties common to Table and Columns objects. There are no methods defined for the Record object. The Table object considerably extends the Record object, mainly in the methods that it provides. These methods deal with adding, modifying, deleting, verifying, and displaying records in the database. The Columns object extends the Record object by including additional properties that define the elements of the List Page, particularly the search criteria available. The Columns object also provides methods for manipulating the elements on a List Page. The Sublist object extends the Table object to represent sublists of data. For example, these sublists can consist of persons on a contract or phone numbers for a person. There are some additional properties in the Sublist object that are needed to specify joining information for the main table. There are also complementary methods to the Table object in the Sublist object to add, delete, copy, get, and display sublist database records.

There is one object that stands alone in this architecture. It is actually a component of the Elem object. That object is the Role object. This object is not part of either hierarchy described above. The purpose of this object is to maintain the translation values for a particular Field or Col object instead of in a database lookup table. There is a method in the Elem object to translate a particular value to its corresponding Role description for display. There is also a method of the Field object to enumerate the different Role values for an HTML Select form element.

There is a set of objects that is not represented by the hierarchies discussed earlier. This set encompasses those objects that are created with declared values for the properties that belong to the associated abstract object. For example, there is an orgTable object that is a Table object with the property values set to represent the organization table in the database. We refer to these objects as concrete objects because they define concrete values for properties of an abstract object. There is a one-to-one mapping of concrete objects to their associated physical form, such as a database table or list page. The list type abstract objects also follow the same pattern. For every List Page that is needed, there is a corresponding concrete object that establishes concrete values for use by that List Page. One could argue that these objects are derived from their corresponding abstract class, but conceptually they are more like instances of a particular abstract class. This can be confusing to hard-core object-oriented developers, but the key thing to remember is that server-side JavaScript is a prototype-based object language, not a class-based one.

## CTBT R&D Web Site HTML Page Design

The objects described in the previous section are components of more complex objects used to put together the HTML pages and to access the data residing in the database. The database stores and manages most of the information presented at our web site. The key to accessing the database revolves around manipulating the objects in the context of the different HTML pages that we use. We use a List Page to view a group of records from the database that match some search criteria. We use a Data Page to view a single record and its related support information, i.e., person information and address information are stored in separate tables but are displayed together on one Data Page. We use a Form Page to edit a single record and its related support information similar to a Data Page. From just these three page types, a user is able to view and edit the database with a great deal of flexibility.

## The Data Page

The Data Page is the simplest form of HTML page we use. It basically takes a record ID in the query string of the URL and displays the information for that record ID in a two column format. A sample Data Page is shown in Figure 4.

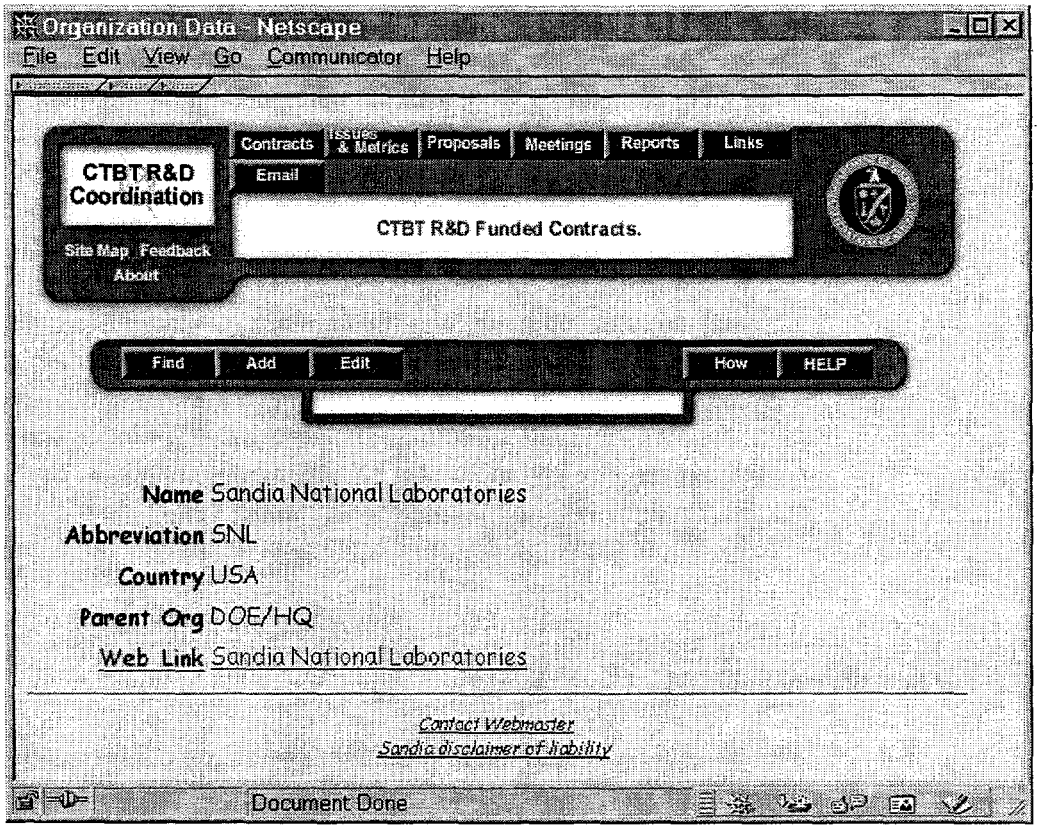


**FIGURE 4.** Sample Data Page.

The pseudo-code for a Data Page follows this pattern:

1. Instantiate the required table object to retrieve record values.
2. Instantiate any support table object(s) to list support values.
3. Get the values from the main table for the record specified.
4. Generate client-side information needed.
5. Instantiate and display the appropriate navigation bar.
6. Instantiate and display a function bar with the appropriate function buttons.

7. Create a form section to support the function buttons.
8. Loop through the fields of the main table object and display the fields.
9. Call the support table list function to get and list the support values for this record, if any.
10. Display the common footer.

Example 6 shows how this pseudo-code looks when implemented in JavaScript. It also shows the flexibility of using objects to control the flow of code on a page. This template can be reused with a different set of similar objects and very little of the code on the page has to change. The instantiation of the appropriate objects will change, but the methods needed to produce the same page for a different data set would not change.

```
<html>
<head>

<title>Organization Data</title>
<server>
// orgdata.html - Display the data for a selected
record.
//
// Author: Randall W. Simons
// Copyright Sandia National Laboratories, 1998
//
// Instantiate the required table object to retrieve
record values.
var table = new OrgTable();

// Instantiate any support table object(s) to list
support value.
var linkTable = new LinkTable();
var rec_id = "";

if(request.orgid != null)
   rec_id = request.orgid;

// Get the values from the main table for the record
specified.
var fieldVal = new Array();
if(rec_id != "") {
   connectDB();
   fieldVal = table.getRecord(rec_id);
   disconnectDB();
}
else {
   showError("orgdata.html: No ID given for record to
```

```
display");
}

// Generate client-side information needed.
table.initClientInfo(fieldVal[0], "Data");
</server>

<script language="JavaScript" src="/coordination/
coordnavbar.js">
</script>

</head>

<BODY bgcolor="#eedfcc" text="#000000" link="#cc0000"
alink="#3366cc" vlink="#3366cc">

<SCRIPT>
<!--
// Instantiate the appropriate navigation bar.
var mainNavBar = new CoordNavBar("contracts");

// Display the navigation bar instantiated.
mainNavBar.displayNavBar("mainNavBar");


// Instantiate a function bar with the appropriate
function buttons.
var buttonList = new makeArray(3);
buttonList[0] = "find";
buttonList[1] = "add";
buttonList[2] = "edit";
var mainFunctBar = orgFunctBar(buttonList, null,
rec_id);

// Display the function bar instantiated.
mainFunctBar.displayFunctBar("mainFunctBar", true);
//-->
</script>

<!-- Add left margin of 20 pixels. -->
<table CELLPADDING="0" CELLSPACING="0" BORDER="0">
<tr><td width=20> </td>
<td>

// Create a form section to support the function
buttons.
<form method="post" name="mainform"
action="orgform.html">
<input type="Hidden" name="submitFunc">
```

```
<input type="Hidden" name="orgid" value=`fieldVal[0]`>
</form>

<table border="0">
<server>

connectDB();

// Loop through the fields of the main table object and
display the
// fields.
for(var ii=1; ii<table.field.length; ii++) {
   if(!fieldVal[ii]) continue;
   table.field[ii].showDataField(fieldVal[ii]);
}

// Call the support table list function to list the
support values
// for this record.
linkTable.listSublistFromDB(fieldVal[0]);
disconnectDB();
</server>
</table>

<!-- End left margin of 20 pixels. -->
</td></tr>
</table>

<script>
<!--
// Display the common footer.
displayFooter();
//-->
</script>

</body>
</html>
```

**EXAMPLE 6.** Data Page code.

Another thing to notice about the Data Page and most of the other pages as well is the function bar. This is the button bar that contains buttons that link to the help system and the other functions that can be launched from this page. Most Data Pages have the options [Find] to go to the List Page, [Add] to go to a Form Page to add a new record for this record type, and [Edit] to go to a Form Page to edit the record being shown. The [How] button takes the user to a HTML page of frequently asked questions (FAQ) about the record type shown. The [HELP] button takes the user to a HTML page that is the help for that particular page. As noted in

the Data Page pseudo-code, the function bar is itself an object. It is a client-side (browser) object that has been defined with all the possible sets of functions it may contain. The code maintainer simply provides the constructor function with the set of buttons (functions) s/he would like to see available on the function bar. The [How] and [HELP] buttons are always available. The function bar can display up to six (6) buttons not including the [How] and [HELP] buttons. This provides consistency across the applications in terms of look-and-feel and maintainability. Example 7 shows how the basic FunctBar object definition looks in code.

```
// Author: Jeff Hampton
// Copyright Sandia National Laboratories, 1998
//
// FunctBar - Object Constructor
// This object defines info and methods needed to
display and
// operate a function bar for any application.

function FunctBar(buttonList, highlight, rec_id,
customText) {

   // Define some base properties.
   this.imageroot = "/app/lib/nav_images/";
   this.buttonList = buttonList;
   this.highlight = highlight;
   this.defaultTxtName = "bar_text";
   this.defaultTxtFile = this.imageroot +
"default_txt.gif";

   // Define the button names to use in the function
bar.
   // Change this array to define the buttons possible.
   this.buttonName = new makeArray(14);
   this.buttonName[0] = "add";
   this.buttonName[1] = "cancel";
   this.buttonName[2] = "delete";
   this.buttonName[3] = "edit";
   this.buttonName[4] = "find";
   this.buttonName[5] = "help";
   this.buttonName[6] = "how";
   this.buttonName[7] = "profile";
   this.buttonName[8] = "quit";
   this.buttonName[9] = "save_as";
   this.buttonName[10] = "save";
   this.buttonName[11] = "reg";
   this.buttonName[12] = "review";
   this.buttonName[13] = "add-page";
```

```
   // Define the corresponding parameters to the
menubar function
   // for those buttons defined.
   // Be sure to make the corresponding changes in this
array also
   // when changing the buttonName array above.
   this.menubarParm = new
makeArray(this.buttonName.length);
   this.menubarParm[0] = "'New'";
   this.menubarParm[1] = "'Close', document.mainform";
   this.menubarParm[2] = "'Delete', document.mainform";
   this.menubarParm[3] = "'Form', " +rec_id;
   this.menubarParm[4] = "'List'";
   this.menubarParm[5] = "'This Page'";
   this.menubarParm[6] = "'How Do I...'";
   this.menubarParm[7] = "'AddProfile'";
   this.menubarParm[8] = "'Close'";
   this.menubarParm[9] = "'Save-Add',
document.mainform";
   this.menubarParm[10] = "'Save-Modify',
document.mainform";
   this.menubarParm[11] = "'Register',
document.mainform";
   this.menubarParm[12] = "'ReviewMine',
document.mainform";
   this.menubarParm[13] = "'New-Page'";

   // If the user's browser supports the images object
then
   // build up the images to be used for the image
roll-overs.
   If (document.images) {
   this.buttonImageOn = new
makeArray(this.buttonName.length);
   this.buttonImageOff = new
makeArray(this.buttonName.length);
   this.buttonImageText = new
makeArray(this.buttonName.length);

    for (var i=0; i<this.buttonName.length; i++) {
     this.buttonImageOn[i] = new Image();
     this.buttonImageOn[i].src = this.imageroot +
      this.buttonName[i] + "_btn_on.gif";
     this.buttonImageOff[i] = new Image();
     this.buttonImageOff[i].src = this.imageroot +
      this.buttonName[i] + "_btn_off.gif";
     this.buttonImageText[i] = new Image();
```

```
        if(customText && customText[i]) {
          // Use supplied custom text message.
          this.buttonImageText[i].src = customText[i];
        }
        else {
          // Use standard text message.
          this.buttonImageText[i].src = this.imageroot +
            this.buttonName[i] + "_txt.gif";
        }
      }

    }

    // Methods - defined in /lib/NavBarMethodLib.js
    this.showFunctButton = showFunctButton;
    this.displayFunctBar = displayFunctBar;
    this.turnButtonOn = turnButtonOn;
    this.turnButtonOff = turnButtonOff;
}
```

**EXAMPLE 7.** FunctBar object.

The same principles apply when you talk about the navigation bar. The navigation bar is a client-side object. It is more statically defined in terms of the buttons that are visible. However, the button that is highlighted is determined by the context of the page, and is passed in to the constructor. When the displayNavBar method is called, that button is highlighted on the navigation bar. Each button's name, link, and help text image are defined within the object. The displayNavBar method puts these together in a uniform way. If buttons need to be added, deleted, or rearranged, the code maintainer simply 1) Edits the NavBar object, 2) Changes the array that controls the order and kind of buttons to be shown, and 3) Makes the corresponding changes to the array representing the button links. The displayNavBar method simply interates over these arrays and displays the appropriate buttons and associates the proper link and help text for each button defined. Example 8 shows how the CoordNavBar object definition looks in code.

```
// coordnavbar.js - Coordination Nav Bar object
definition
//
// Author: Jeff Hampton
// Copyright Sandia National Laboratories, 1998
//
// CoordNavBar - Object Constructor
function CoordNavBar(highlight) {
   // Define some base properties.
   this.imageroot = "/coordination/nav_images/";
```

```
   this.linkroot = "/coordination/";
   this.highlight = highlight;
   this.defaultTxtName = "nav_text";

   // Define the links for the feedback, sitemap, and
about areas.
   this.feedback =
"mailto:webmaster@"+location.hostname;
   this.sitemap = "";
   this.about = this.linkroot + "about.html";
   this.home = this.linkroot;

   // Define the button names to use in the nav bar.
   // Change this array to define the buttons and their
order.
   this.buttonName = new makeArray(7);
   this.buttonName[0] = "contracts";
   this.buttonName[1] = "metrics";
   this.buttonName[2] = "proposals";
   this.buttonName[3] = "meetings";
   this.buttonName[4] = "reports";
   this.buttonName[5] = "links";
   this.buttonName[6] = "email";

   // Define the corresponding links for those buttons
defined.
   // Be sure to make the corresponding changes in this
array also
   // when changing the buttonName array above.
   this.buttonLink = new
makeArray(this.buttonName.length);
   this.buttonLink[0] = "/contractApp";
   this.buttonLink[1] = this.linkroot + "metrics.html";
   this.buttonLink[2] = this.linkroot +
"proposals.html";
   this.buttonLink[3] = this.linkroot +
"meetings.html";
   this.buttonLink[4] = this.linkroot + "reports.html";
   this.buttonLink[5] = "/linkApp";
   this.buttonLink[6] = this.linkroot + "email.html";

// If the user's browser supports the images object
then
// build up the images to be used for the image roll-
overs.
if (document.images) {
  this.buttonImageOn = new
makeArray(this.buttonName.length);
```

```
  this.buttonImageOff = new
makeArray(this.buttonName.length);
  this.buttonImageText = new
makeArray(this.buttonName.length);

  for (var i=0; i<this.buttonName.length; i++) {
    this.buttonImageOn[i] = new Image();
    this.buttonImageOn[i].src = this.imageroot +
       this.buttonName[i] + "_btn_on.gif";
    this.buttonImageOff[i] = new Image();
    this.buttonImageOff[i].src = this.imageroot +
       this.buttonName[i] + "_btn_off.gif";
    this.buttonImageText[i] = new Image();
    this.buttonImageText[i].src = this.imageroot +
       this.buttonName[i] + "_txt.gif";
  }
}

// Make the sure the highlighted button's help text is
the default.
if (this.highlight != null && this.highlight != "") {
   this.defaultTxtFile = this.imageroot +
this.highlight +
      "_txt.gif";
}
else { // Otherwise, just use the regular default.
   this.defaultTxtFile = this.imageroot +
"coor_title.gif";
}

// Methods to support the object.
this.displayNavBar = displayNavBar;
this.turnButtonOn = turnButtonOn;
this.turnButtonOff = turnButtonOff;

}
```

**EXAMPLE 8.** CoordNavBar object.

One handy user interface element on the Data Page is the ability to navigate to other Data Pages for any of the elements of a data record that are supported by their own applications. Using the sample Data Page, clicking on the "Web Link" text link would take the user to another Data Page that would display the full record for the value associated with that web link. This gives the user a familiar look-and-feel when crossing the different applications within our web site.

## The Form Page

The Form Page is more complicated than the Data Page. The Form Page must include sections of code to add, modify, or delete a particular record type. It initially takes a record ID in the query string of the URL and displays the information for that record ID in a columnar format for editing. A blank Form Page will be displayed if no record ID is passed on the query string. A sample Form Page is shown in Figure 5.

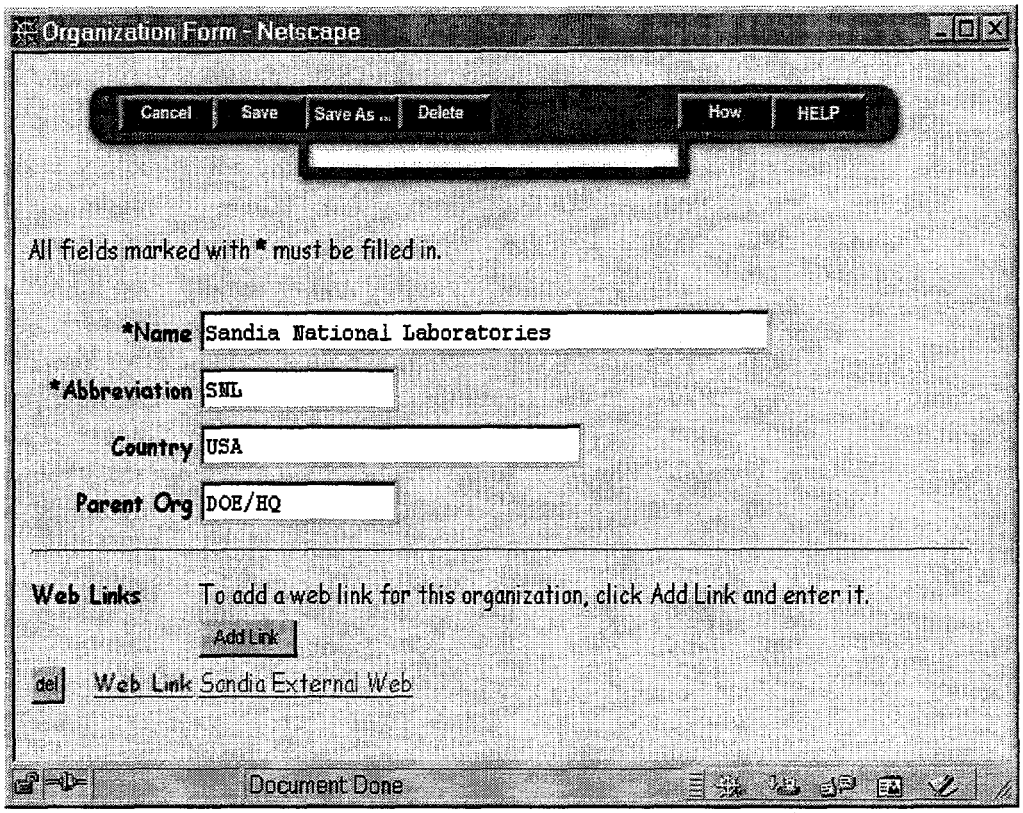


**FIGURE 5.** Sample Form Page.

A Form Page is treated differently by the application because it is always presented in a separate window by itself without all the ornamentation of a normal browser window. This is because we don't want the user navigating anywhere else while they are editing/adding a record. This helps to maintain the state of the window so that when the user is ready to submit the record to the database, all the data is true to what the user entered or was displayed.

The pseudo-code for a Form Page follows this pattern:

1. Instantiate the required table object to retrieve record values.
2. Instantiate any support table object(s) to list support values.
3. Get the values from the database or the request object for the record specified.
4. Generate client-side information needed.
5. Do the function chosen when form was submitted, if any.
6. Provide functions to open subordinate applications for support data, if any.
7. Provide the necessary validation checks in the isReady function for all required fields.
8. Instantiate and display a function bar with the appropriate function buttons.
9. Create a form section to support the function buttons and the input fields.
10. Loop through the fields of the main table object and display the fields.
11. Build the interface to edit support table values, if any.
12. Call the support table list function to list the support values for this record, if any.

Example 9 shows how this pseudo-code is implemented in JavaScript. Just like the Data Page framework, it shows the flexibility of using objects to control the flow of code on a page. The template can be reused by another application that needs to edit a database record. Similar to the Data Page, the instantiation of appropriate objects will change, but the methods to implement the page would not have to change. The learning curve for new developers using the system is also reduced since they are familiar with the pattern of development for each type of page. This means they can be more productive faster than if the pages were individualized and didn't follow the pseudo-code patterns shown for the Data Page and the Form Page.

```
<html>
<head>
<title>Organization Form</title>

<server>
// orgform.html - Display a form to add a record to the
database,
//    or modify a record selected from the database.
//
// Author: Randall W. Simons
// Copyright Sandia National Laboratories, 1998
//
```

```
// Instantiate the required table object to retrieve
record values.
var table = new OrgTable();

// Instantiate any support table object(s) to list
support values.
var linkTable = new LinkTable();

// Get the values from the database or the request
object for the
// record specified.
var tmpId = request.tmpId;// Temporary negative ID.
var fieldVal = new Array();
var dataSrc = "none";
fieldVal = table.getFieldVal();

// Generate client-side information needed.
table.initClientInfo(fieldVal[0], "Form");

// Do the function chosen when form was submitted, if
any.
if(request.submitFunc) {
  connectDB();
  switch (request.submitFunc) {
  case "Save-Add":
    // Check that "similar" record doesn't already
exist.
    table.checkDupRecord(fieldVal);
    fieldVal[0] = -tmpId;
    table.addRecord(fieldVal);

    // Copy and Add associated links.
    for(var ii=0; ii<request.linkCount; ii++) {
      var action = eval("request.linkAction" +ii);
      if(action != "del")
        linkTable.addSublist(fieldVal[0],
eval("request.linkId" +ii));
    }
    request.dataSource = "database";
    request.change = "add";
    break;
  case "Save-Modify":
    table.modifyRecord(fieldVal[0], fieldVal);

    // Add and delete associated links.
    for(ii=0; ii<request.linkCount; ii++) {
```

```
        var rec_id = eval("request.linkId" +ii);
        var action = eval("request.linkAction" +ii);
        if(action == "add")
          linkTable.addSublist(fieldVal[0], rec_id);
        else if(action == "del")
          linkTable.deleteSublist(fieldVal[0], rec_id);
      }
      request.dataSource = "database";
      request.change = "mod";
      break;
    case "Delete":
      // Delete associated links.
      linkTable.deleteAllSublist(fieldVal[0]);

      table.deleteRecord(fieldVal[0]);

      disconnectDB();
      redirect(table.prefix+
"form.html?dataSource=none&change=del");
      break;
    case "Close":          // Cancel
      // Tell client to close the window.
      redirect("/closeWindow.html");
      break;
    case "changeSublist":
      // No-op case when change a sublist element.
      break;
    default:
      disconnectDB();
      showError("Logic Error in orgform.html:
request.submitFunc != valid value");
    }
    disconnectDB();
}

table.reloadOpener(fieldVal[0]);

if(request.closeOnAdd && request.closeOnAdd == "T" &&
request.change == "add") {
  // Tell client to close the window.
  write(expandTpl("windowClose", null));
}
</server>

<SCRIPT LANGUAGE="JavaScript" SRC="/formval/
FormChek.js"></SCRIPT>
```

```
<SCRIPT LANGUAGE="JavaScript">
<!--
// Provide functions to open subordinate applications
for support
// data, if any.
function openLinkApp(link_id, owner_id) {
  if(link_id != "") {
    window.open("/linkApp/
linkform.html?dataSource=database&link_id=" +
      link_id+ "&app=org", "linkFormFromOrg",
appWinSettings);
  }
  else {
    window.open("/linkApp/
linkform.html?dataSource=none&owner_id=" +
      owner_id+ "&app=org", "linkFormFromOrg",
appWinSettings);
  }
}

// Provide the necessary validation checks in the
isReady function
// for all required fields.
function isReady(theForm) {
if(!checkString(theForm.abbr, "Abbreviation", false))
return false;
  return true;
}
//-->
</SCRIPT>
</head>

<BODY bgcolor="#eedfcc" text="#000000" link="#cc0000"
alink="#3366cc" vlink="#3366cc">

<script>
<!--
// Instantiate a function bar with the appropriate
function buttons.
if (rec_id < 0) {
  // No rec_id means empty form, so can't modify or
delete.
  var buttonList = new makeArray(2);
  buttonList[0] = "cancel";
  buttonList[1] = "save_as";
}
```

```
else {
  var buttonList = new makeArray(4);
  buttonList[0] = "cancel";
  buttonList[1] = "save";
  buttonList[2] = "save_as";
  buttonList[3] = "delete";
}
var mainFunctBar = orgFunctBar(buttonList, null,
rec_id);

//Display the function bar instantiated.
mainFunctBar.displayFunctBar("mainFunctBar", true);
//-->
</script>

// Create a form section to support the function
buttons and the
// input fields.
<FORM METHOD="Post" name="mainform"
ACTION="orgform.html">
<input type="Hidden" name="dataSource"
value="request">
<input type="Hidden" name="submitFunc">
<p>
All fields marked with <B>*</B> must be filled in.
<p>
<table border=0 width="100%">

<server>
if(request.closeOnAdd && request.closeOnAdd == "T")
  write('<INPUT TYPE="Hidden" NAME="closeOnAdd"
VALUE="T">\n');
write('<INPUT TYPE="Hidden" NAME="tmpId" VALUE='
+tmpId+ '>\n');

// Loop through the fields of the main table object and
display the
// fields.
for(var ii=0; ii<table.field.length; ii++) {
  if(ii==0) {
    // Don't show internal IDs.
    write('<INPUT TYPE="Hidden" NAME="'
+table.field[ii].name+ '" VALUE=' +
      fieldVal[ii]+ '>\n');
  }
  else {
```

```
    // Provide text fields for input.
    table.field[ii].showFormField(fieldVal[ii]);
  }
}
</server>

// Build the interface to edit support table values, if
any.
<tr>
  <td colspan=3><hr></td></tr>
<TR>
  <TH colspan=2 ALIGN=Left>Web Links</th>
  <TD>To add a web link for this organization, click
Add Link and enter it.</td>
</tr>
<tr>
  <td colspan=2 align="right">
  </td>
  <td><INPUT TYPE="Hidden" name="linkAddId" value="">
  <INPUT TYPE='button' VALUE='Add Link'
onClick='openLinkApp("", rec_id)'></td>
</tr>

<server>
connectDB();

  // Call the support table list function to list the
support values
   // for this record, if any.
   var recs = new Array();
  if(dataSrc == "database" && fieldVal[0] > 0) {
    recs = linkTable.getSublist(fieldVal[0], false);
  }
  else if(dataSrc == "request") {// Even if count=0,
check for add.
    var actions = new Array();
    recs = linkTable.reqGetSublist(request, actions);
  }
  linkTable.listSublist(recs, "del", false, actions);
disconnectDB();
</server>

</table>
</form>

</body>
```

```
</html>
```

**EXAMPLE 9.** Form Page code.

One thing to notice about this page's code is the iterative nature of the form submission process. As its action destination in the <FORM> tag, the page defines itself. This is where the *switch* statement at the beginning of the page becomes useful. For a particular page, you can define any number of functionality designators, i.e. Save-Add, Save-Modify, Delete, etc. These designators can be linked to buttons on the page, functions on the function bar, etc. The code provided in each section of the *switch* statement performs a designated function based on the context in which the form was submitted. By setting the *submitFunc* variable prior to submitting the form, the context of the form submission is designated.

You may notice there are some client-side JavaScript functions supporting the Form Page. These functions serve two purposes: 1) Provide methods to open new windows in order to input data using supporting applications. 2) Provide a method to validate the form data before it is sent to the server. The methods for opening new windows provide a way to reuse application code that was built to maintain other types of database data. By allowing the user to link to existing applications, code reuse and maintainability increase. The *isReady* function provides a central point of validation for the form. Any required fields should be checked and any field data checks should be called within this function. If any of these checks fail the *isReady* function will return a value of *false*. This will keep the form from being submitted to the server until all checks have been validated to *true*. For this web site, all of our client-side validation functions reside in a single file called "FormChek.js". Since every Form Page uses this file, there is only one place where the code maintainer has to go to add, remove, or change any form validation functions. This helps to increase the maintainability of the web site overall.

In addition to the design patterns used for the code, there are some user interface patterns that we use, especially on the Form Page. These patterns relate to entering multi-data information. Multi-data information consists of information relevant to a record that has a many-to-one relationship. For example, an organization can have more than one web link associated with it. That web link information is a type of multi-data information for the organization record. When the user enters this type of information, they first will click on a button that takes them to another Form Page or List Page inside a separate window. This allows the user to enter or select from a list the relevant data for that record. Generally the button will start with the word "Add" to take the user to the appropriate Form Page. Buttons that start with the word "Select" will take the user to a list where they can enter search criteria to find data in a list to associate with the given record. The results of these associations are listed below the "Add" or "Select" button. For each row of this

data, there is a [del] button, the name of the type of data associated, and the value of that type of data associated. The [del] button allows the user to disassociate that row's value from the main record. The type of the data associated is always clickable and opens a new window that will allow the user to edit that piece of information. Sometimes the values themselves are clickable if it makes sense for those values. In the organization Form Page sample, the value "Sandia External Web" is clickable because it is a link to a web site. This serves to provide the user with consistency in the user interface and helps to bring down the learning curve of users.

**The List Page**

The List Page is the most complicated of the three types of general application pages supported by our web site. The List Page can display predetermined views of data, sort the data on a given field, match data on a given field, display only the user selected columns of information, display the data in multiple formats, and perform actions on a list of checked records. It has two states: 1) The start of a search, and 2) displaying search results. The first state is represented by Figure 6. The second state is represented by Figure 7. The user may cycle through these states as much as needed to get the list of data they desire.

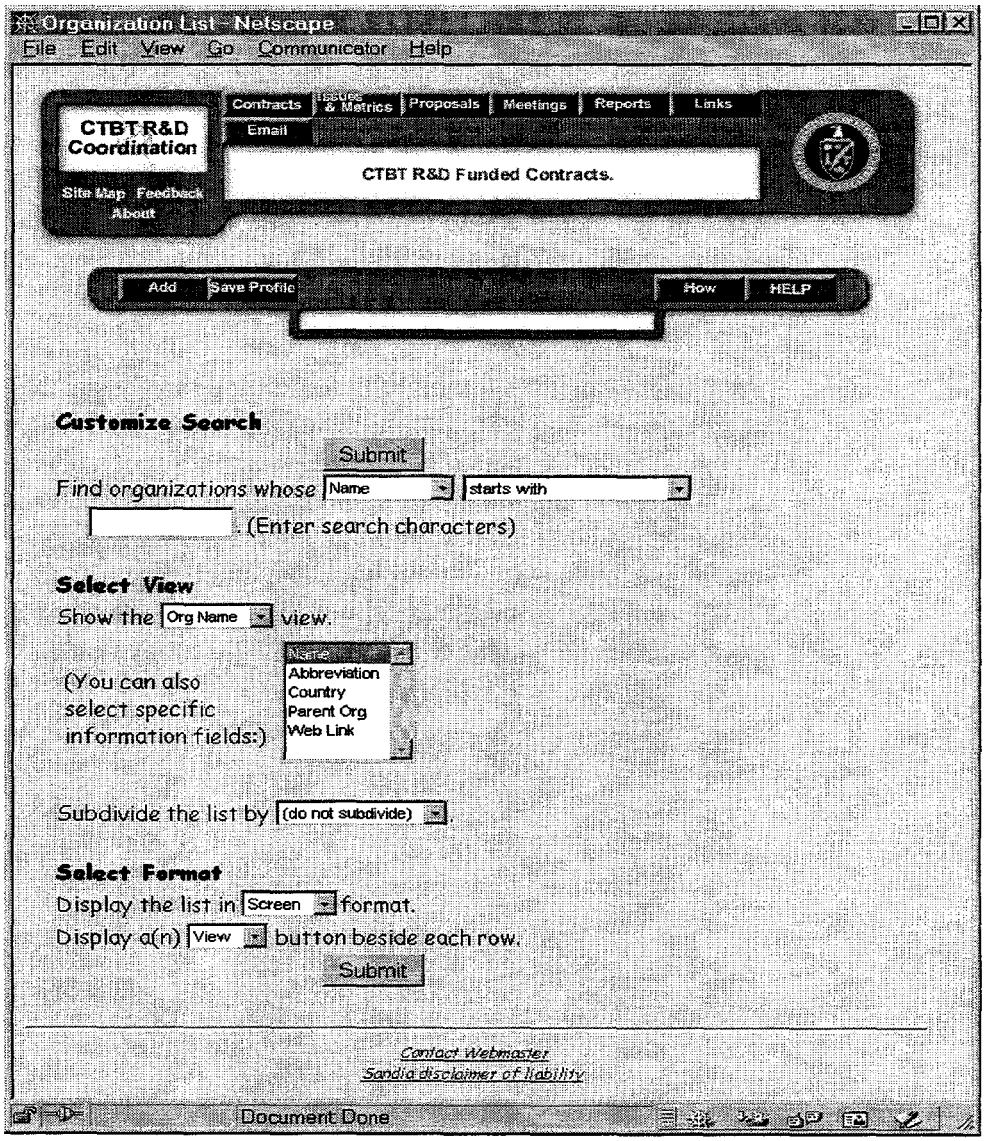


**FIGURE 6.** Sample List Page Start State.

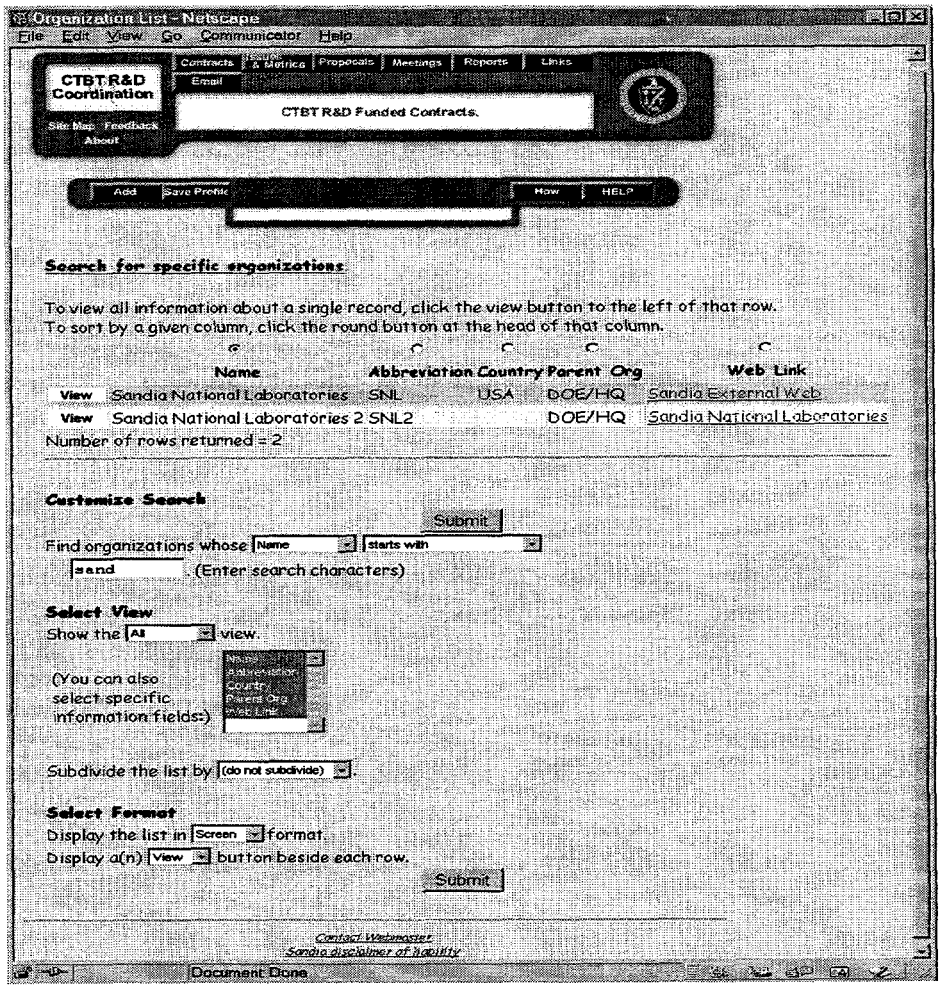


**FIGURE 7.** Sample List Page Result State.

The pseudo-code for a List Page follows this pattern:

1. Instantiate the required list object that describes the List Page.
2. Set the current format for the List Page.
3. Set the current view for the List Page.
4. Generate client-side information needed.
5. Instantiate a client-side JavaScript object to control the view area.
6. Decide whether to show a Navigation Bar.
7. Decide whether to show a Function Bar.

8. Create a form section to support the function buttons and search criteria inputs.
9. Initialize the list of columns to display.
10. Initialize the field to match against.
11. Initialize the column to subdivide against.
12. Show list only if have already shown user a page with query on it.
13. Determine whether to display the search criteria.
14. Determine whether to display the common footer.

Example 10 shows an example HTML file that follows this pattern. Just like the Data Page and the Form Page framework, it shows the flexibility of using objects to control the flow of code on a page. The template can be reused by another application that needs to have the capabilities of a List Page. Similar to the Data Page and the Form Page, the instantiation of appropriate objects will change, but the methods to implement the page would not have to change. The learning curve for new developers using the system is also reduced since they are familiar with the pattern of development for each type of page. This means they can be more productive faster than if the pages were individualized and didn't follow the pseudo-code patterns shown for the Data Page, Form Page, or the List Page.

```
<html>
<head>
<title>Organization List</title>

<server>
// orglist.html - Display a list of records selected
from the
// database.
//
// Author: Randall W. Simons
// Copyright Sandia National Laboratories, 1998
//
// Instantiate the required list object that describes
the List
// Page.
var list = new OrgCol();

// Set the current format for the List Page.
var format = "Screen";
if(request.format)
   if(request.format != "undefined" && request.format
!= "")
      format = request.format;

// Set the current view for the List Page.
var mode = "view";
```

```
if(request.mode)
   if(request.mode != "undefined" && request.mode !=
"") {
      mode = request.mode;
   }

// Generate client-side information needed.
list.initClientFromColumns(mode);
</server>

<script language="JavaScript" src="/coordination/
coordnavbar.js">
</script>

</head>

<BODY bgcolor="#eedfcc" text="#000000" link="#cc0000"
alink="#3366cc" vlink="#3366cc">

<server>
write("<script>\n");
write("<!--\n");

// Instantiate a client-side JavaScript object to
control the view
// area.
write("var orgColView = new OrgColView();\n");

// Decide whether to show a Navigation Bar.
if(mode != "select" && mode != "multi-select" && format
== "Screen") {
   write('var mainNavBar = new
CoordNavBar("contracts");\n');
   write('mainNavBar.displayNavBar("mainNavBar");\n');
}

// Decide whether to show a Function Bar.
if(format == "Screen") {
   write(expandTpl("functBarOnListPage", list));
}

write('//-->\n');
write('</script>\n');
</server>

<!-- Add left margin of 20 pixels. -->
<table CELLPADDING="0" CELLSPACING="0" BORDER="0">
<tr><td width=20> </td>
<td>
```

```
// Create a form section to support the function
buttons and search
// criteria inputs.
<FORM method="Post" name="showlistform"
ACTION="orglist.html">
<INPUT TYPE="Hidden" NAME="table"
VALUE="organization">
<input type="Hidden" name="submitFunc">

<server>

// Initialize the list of columns to display.
list.initColumnList();

// Initialize the field to match against.
list.initQueryList();

// Initialize the column to subdivide against.
var colSubdiv = new Array();
colSubdiv = list.initSubdivList();

// Show list only if have already shown user a page
with query on
// it.
if(request.isAfterQuery) {
   if(mode == "view") {
      list.showList("orgdata.html", "view-square.gif");
   }
   else if(mode == "edit") {
      list.showList("orgform.html", "edit-square.gif");
   }
   else if(mode == "select") {
      list.showList("orgdata.html", "select-
square.gif");
   }
}

// Determine whether to display the search criteria.
if(request.format != "Printer") {
   write('<br><font size=+1><B>Customize Search</
B><font><br>\n');
   write(expandTpl("submitButton", null));
   list.showQueryList();
   list.showViewList("orgColView");
   write('  \n');
   list.showColumnList();
   list.showSubdivList(colSubdiv);
   if(mode != "select" && mode != "multi-select")
        list.showFormatList(format);
```

```
        write(expandTpl("submitButton", null));
}
</server>

</FORM>

<!-- End left margin of 20 pixels. -->
</td></tr>
</table>

<server>

// Determine whether to display the common footer.
if(mode != "select" && mode != "multi-select" && format
== "Screen") {
   write(expandTpl("displayFooter", null));
}
</server>

</body>
</html>
```

**EXAMPLE 10.** List Page code.

Similar to the Form Page, the List Page code is iterative in nature during the form submission process. As its action destination in the <FORM> tag, the page defines itself. This is why in pseudo-code step 12 it is necessary to determine whether to show the list or not. Also due to the iteration process, there are condition statements that determine whether to display certain aspects of the List Page, such as the Navigation Bar, the search criteria, or the footer, etc. based on what was chosen by the user for formatting the results returned. There is only one context in which a List Page can be submitted, i.e., view results, which is why there is no need for a *switch* statement similar to the one found in the Form Page.

You may notice that the section of code that displays the search criteria is divided into multiple method calls. This allows the developer to arrange the search criteria in any order they wish. They also have the choice of showing only the search criteria they would like to allow for a particular List Page. Breaking the display of the search criteria into multiple method calls allows the developer more flexibility than if the search criteria were displayed using a single method call.

The use of presentation templates [Long98] is another aspect of the List Page that is interesting. It is used to display the "Submit" buttons for the List Page. Presentation templates allow us to separate some of the presentation from the server-side JavaScript code. Instead of having to use a number of *write* statements to produce HTML, we can define a template that contains HTML with some placeholders.

These placeholders are replaced with parameters passed to the template engine. Using this technique means that much of the HTML that would have been embedded in code can now be changed independent of the code. This means the application does not have to be recompiled in order to make these changes become visible. These templates are in fact used throughout our web site applications. Most of the time the calls to the template engine are buried within an object's methods. But the List Page also uses the template engine directly.

## Future Directions: Three-Tiered Architecture

A three-tiered architecture similar to the three-tiered client/server model [Edwards97] splits an application into three domains: data, function, and presentation. Using this architecture as a basis for design often improves the modularity, maintainability, and extensibility of the code. The database and all the code to interface to it comprise the data domain. The HTML data representation and the code to generate it make up the presentation domain. Although generated by the server in the presentation domain, the data represented in the request object is used in the function domain. All of the transformations between database, HTML, and request object are also part of the function domain. The implementation described previously uses JavaScript objects to collect related functions and data together, but falls short of always cleanly separating code that deals with different data representations. For example, the heart of the list page code constructs a database query based on request object properties, processes the retrieved data, and generates the HTML. This code would have to be changed if any one of the three data representations was modified.

Using JavaScript's object capability, we are currently implementing the three-tiered architecture by defining objects that encapsulate connections to each domain. Table objects are defined to hold the environment and the code to interact with the database. Page objects have sole control over generating HTML. We also add methods to the already existing request object to connect it to the Table objects.

We considered adding another "neutral" data representation that Table, Page, and request objects would all use to communicate with each other. This would have the advantage of truly isolating the database from the HTML and request object, such that one could completely replace the database without needing to modify any code dealing with HTML or the request object, and vice versa. (Note that HTML and the request object can not be completely isolated, since the request object is generated automatically from the HTML by the web server process.) The drawbacks to this approach include an increase in the complexity of the design and

code, since you would be adding another data representation and transformations to and from it, and a probable decrease in performance as these extra transformations are executed.

As an alternative, we observed that Table objects are the "smallest" unit of data we deal with, that is, Page and request objects may contain data that came from multiple tables. So, instead of inserting yet another data representation and requiring more transformations, we use the Table objects as the common data representation that all other objects understand. This naturally leads to making conversions of Table object data to Page object data into methods of Page objects, since they need to understand Table objects. Likewise, transformations of request object data to Table object data reside in methods of the request object. This will give us the architecture shown in Figure 8.

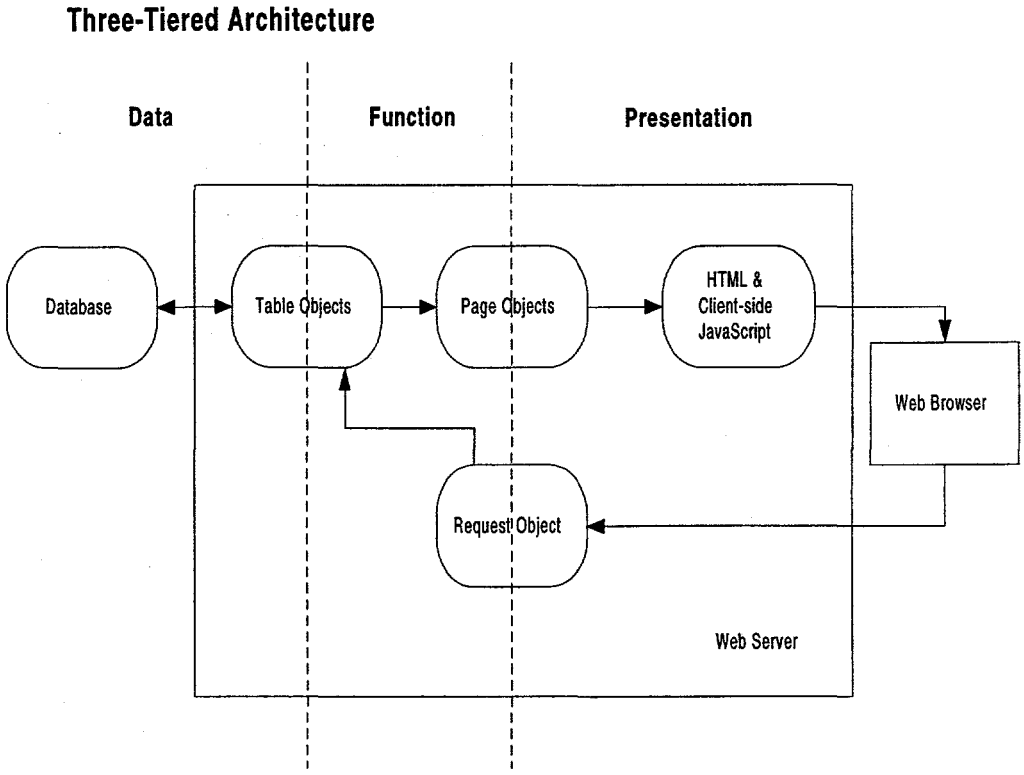


**FIGURE 8.** Three-Tiered Architecture Diagram.

## Conclusions

There are major differences between JavaScript and mainstream object-oriented programming languages. These include being prototype-based instead of class-based, and missing support for information hiding. Despite these differences, we have found it not only possible, but advantageous to use JavaScript (both server-side and client-side) as an object-oriented language to implement an object-oriented design. We think that the object-oriented approach we have developed has enhanced our ability to build complex applications using SSJS, made the code we write easy to read and understand, as well as, given us a framework that is extensible and easy to use. This framework can be more easily maintained and extended than an equivalent function library implementation. Our experience with supporting a web site has convinced us that maintainability and extensibility are crucial to the success of that site.

## References


[csjs] Netscape Communications Corporation, *JavaScript Guide*, 1996.

[Edwards97] Jeri Edwards, *3-Tier Client/Server at Work*, John Wiley & Sons, 1997.

[jsobj] Netscape Communications Corporation, "Object Hierarchy and Inheritance in JavaScript", December 1997, http://developer.netscape.com/docs/manuals/communicator/jsobj/contents.htm.

[jsref] Netscape Communications Corporation, *JavaScript Reference*, 1997.

[Long98] Glen Long, "Using Presentation Templates with Server-Side JavaScript", View Source, 1998, http://developer.netscape.com/viewsource/long_ssjs/long_ssjs.html.

[Shafer10/97] Dan Shafer, "surprise! JavaScript *is* object-oriented", Master Builder Column of CNet's web site builder.com, October 6, 1997, http://builder.cnet.com/Programming/Shafer/100697.

[Shafer11/97] Dan Shafer, "I meant what I said: JavaScript really *is* object-oriented", Master Builder Column of CNet's web site builder.com, November 17, 1997, http://builder.cnet.com/Programming/Shafer/111797.

[ssjs] Netscape Communications Corporation, *Writing Server-Side JavaScript Applications*, 1997.

## Further Reading


Kaveh Gh. Bassiri, *Programming Applications for Netscape Servers*, Addison-Wesley Pub. Co., 1998.

David Flanagan, *JavaScript: The Definitive Guide, 2nd Edition*, O'Reilly & Associates, 1997.

Nick Heinle, *Designing with JavaScript: Creating Dynamic Web Pages*, O'Reilly & Associates, 1997.

JJ Kuslich and Robert Husted, *Server-Side JavaScript, A Developers Guide*, Addison-Wesley Pub. Co., 1999.

Frank Manola, "Technologies for a Web Object Model", (to be published in IEEE Internet Computing, Jan/Feb 1999), http://www.objs.com/survey/wom-ieee.htm.

## DISTRIBUTION:

1 DOE Office of Research and Development/NN-20
Attn: Leslie A. Casey
1000 Independence Ave. SW
Washington, DC 20585-0420

1 Lawrence Livermore National Laboratory
Attn: John (Jay) Zucca, MS L205
PO Box 808
Livermore, CA 94551-0808

3 Los Alamos National Laboratory
Attn: Wendee M. Brunish, EES-DO
Fran C. Chavez, D460 NIS-RD
Dr. Jill Warren, B230 NIS-8
PO Box 1663
Los Alamos, NM 87545

1 Pacific Northwest National Laboratory
Attn: Ray A. Warner, MS K6-48
PO Box 999
Richland, WA 99352

1 Air Force Technical Applications Center
Attn: David R. Russell
HQ/AFTAC/TTR
1030 S. Highway A-1A
Patrick AFB, FL 32925-3002

1 Center for Monitoring Research/SAIC
Attn: Robert G. North
1300 N. 17th St./Suite 1450
Arlington, VA 22209

1 PTS/PrepCom/CTBTO
Attn: Mary L. Girven
Vienna International Centre, Room E0711
PO Box 1200
Vienna A-1400
AUSTRIA

| | | |
|---|---|---|
| 1 | MS 0661 | Patrick B. Milligan, 4825 |
| 1 | MS 0979 | Larry S. Walker, 5704 |
| 1 | MS 1137 | Debbie Kernan, 6534 |
| 1 | MS 1138 | Larry J. Ellis, 6531 |

## DISTRIBUTION (continued):

| | | |
|---|---|---|
| 1 | MS 1138 | Ralph G. Keyser, 6531 |
| 1 | MS 1138 | Marjorie T. McCornack, 6531 |
| 1 | MS 1138 | Matthew N. Chown, 6532 |
| 1 | MS 1138 | Tony L. Edwards, 6532 |
| 1 | MS 1138 | Bruce N. Malm, 6532 |
| 1 | MS 1138 | Randall W. Simons, 6532 |
| 1 | MS 1138 | Sharon K. Chapa, 6533 |
| 6 | MS 1138 | Jeffery W. Hampton, 6533 |
| 1 | MS 1138 | Bret M. McGowen, 6533 |
| 1 | MS 1140 | James K. Rice, 6500 |
| | | |
| 1 | MS 9018 | Central Technical Files, 8940-2 |
| 2 | MS 0899 | Technical Library, 4916 |
| 1 | MS 0619 | Review & Approval Desk, 15102<br>For DOE/OSTI |
| 1 | MS 0161 | Patent and Licensing Office, 11500 |