Creating Reusable Components

As you've seen in previous chapters, a WebObjects application comprises a number of components. Each component is represented in the file system as a directory (for example, **MyComponent.wo**) that stores the component's template, declarations, and script files (**MyComponent.html**, **MyComponent.wod**, and **MyComponent.wos**, respectively) as well as other resource files. In the simplest applications, each component corresponds to an HTML page, and no two applications share components. However, one of the strengths of the WebObjects' architecture is its support of reusable components: components which once defined, can be used within multiple applications, multiple pages of the same application, or even multiple sections of the same page.

This chapter describes reusable components and shows you how to take advantage of them in your applications. The topics covered include:

- The benefits of reusable components
- How to use them in your applications
- How applications find these components at run time
- How to design components for reusability

Benefits of Reusable Components

Reusable components benefit you in two fundamental ways. They help you:

- Centralize application resources
- Simplify interfaces to packages of complex, possibly parameterized, logic and display.

The following sections explain these concepts in detail.

Centralizing Application Resources

One of the challenges of maintaining a web-based application derives from the sheer number of pages that must be created and maintained. Even a modest application can contain scores of HTML pages. Although some pages must be crafted individually for each application, many (for example, a page that gathers customer information) could be identical across applications. Even pages that aren't identical across applications can share at least some portions (header, footer, navigation bars, and so on) with pages in other applications. With reusable components, you can factor out a portion of a page (or a complete page) that's used throughout one or more applications, define it once, and then use it wherever you want, simply by referring to it by name. This is a simple but powerful concept, as the following example illustrates.

Suppose you want to display a navigational control at the bottom of each page of your application, something like this

Figure 1. Navigational Control

The HTML code for this control is:

```
<HTML>
<HEAD>
    <TITLE>World Wide Web Wisdom, Inc.</TITLE>
</HEAD>

<BODY>
Please come visit us again!

<!-- start of navigation control -->
<CENTER>
<TABLE BORDER = 7 CELLPADDING = 0 CELLSPACING = 5>
    <TR ALIGN = center>
        <TH COLSPAN = 4> World Wide Web Wisdom, Inc.</TH>
    </TR>
    <TR ALIGN = center>
        <TD><A HREF = "http://www.wwww.com/home.html"> Home <a></TD>
        <TD><A HREF = "http://www.wwww.com/sales.html"> Sales <a></TD>
        <TD><A HREF = "http://www.wwww.com/service.html"> Service <a></TD>
        <TD><A HREF = "http://www.wwww.com/search.html"> Search <a></TD>
    </TR>
</TABLE>
</CENTER>
<!-- end of navigation control -->

</BODY>
</HTML>
```

Thirteen lines of HTML code define the HTML table that constitutes the navigational control. You could copy these lines into each of the application's pages or use a graphical HTML editor to assemble the table wherever you need one. But as application size increases, these approaches becomes less practical. And obviously, when a decision is made to replace the navigational table with an active image, you must update this code in each page. Duplicating HTML code across pages is a recipe for irritation and long hours of tedium.

With a reusable component, you could define the same page like this:

```
<HTML>
<HEAD>
    <TITLE>World Wide Web Wisdom, Inc.</TITLE>
</HEAD>

<BODY>
Please come visit us again!

<!-- start of navigation control -->
<WEBOBJECT NAME="NAVCONTROL"></WEBOBJECT>
<!-- end of navigation control -->

</BODY>
</HTML>
```

The thirteen lines are reduced to one, which positions the WebObject named NAVCONTROL. The declarations file for this page binds the WebObject named NAVCONTROL to the component named NavigationControl:

```
NAVCONTROL: NavigationControl {};
```

All of the application's pages would have entries identical to these in their template and declarations files.

WebObjects Builder makes reusable components even more attractive by providing a graphical way to create and maintain them. With the builder, you can assemble these components graphically and then put them on a palette for later use. Adding a custom reusable component to you application becomes a simple matter of dragging it from Builder's palette into your application and binding its attributes to variables or methods just as you would a dynamic element you drag from Builder's palettes. See "Advanced WebObjects Builder Tasks" in the *WebObjects Builder Guide* for more information.

NavigationControl is a component that's defined once, for the use of all of the application's pages. Its definition is found in the directory **NavigationControl.wo** in the file **NavigationControl.html** and contains the HTML for the table:

```
<CENTER>
<TABLE BORDER = 7 CELLPADDING = 0 CELLSPACING = 5>
    <TR ALIGN = center>
        <TH COLSPAN = 4> World Wide Web Wisdom, Inc.</TH>
    </TR>
    <TR ALIGN = center>
        <TD><A HREF = "http://www.wwww.com/home.html"> Home <a></TD>
        <TD><A HREF = "http://www.wwww.com/sales.html"> Sales <a></TD>
```

```
        <TD><A HREF = "http://www.wwww.com/service.html"> Service <a></TD>
        <TD><A HREF = "http://www.wwww.com/search.html"> Search <a></TD>
    </TR>
</TABLE>
</CENTER>
```

Since NavigationControl defines a group of static elements, no declaration or script file is needed. However, a reusable component could just as well be associated with complex, dynamically determined behavior, as defined in an associated script file.

Now, to change the navigational control on all of the pages in this application, you simply change the NavigationControl component. What's more, since reusable components can be shared by multiple applications, the World Wide Web Wisdom company could change the look of the navigational controls in all of its applications by changing this one component.

If your application's pages are highly structured, reusable components could be the prevailing feature of each page:

```
<HTML>
<HEAD>
    <TITLE>World Wide Web Wisdom, Inc.</TITLE>
</HEAD>

<BODY>

<WEBOBJECT NAME="HEADER"></WEBOBJECT>
<WEBOBJECT NAME="PRODUCTDESCRIPTION"></WEBOBJECT>
<WEBOBJECT NAME="NAVCONTROL"></WEBOBJECT>
<WEBOBJECT NAME="FOOTER"></WEBOBJECT>

</BODY>
</HTML>
```

The corresponding declarations file might look like this:

```
HEADER: CorporateHeader {};
PRODUCTDESCRIPTION: ProductTable {productCode = "WWWW0314"};
NAVCONTROL: NavigationControl {};
FOOTER: Footer {type = "catalogFooter"};
```

Notice that some of these components above take arguments, that is, they are parameterized. For example, the ProductTable component's **productCode** attribute is set to a particular product identifier, presumably to display a description of that particular product. The combination of reusability and customizability is particularly powerful, as you'll see in the following section.

Simplifying Interfaces

Another benefit of reusable components is that they let you work at a higher level of abstraction than would be possible by working directly with HTML code or with WebObjects' dynamic elements. You (or someone else) can create a component that encapsulates a solution to a possibly complicated programming problem, and then reuse that solution again and again without having to be concerned with the details of its implementation. Examples of this kind of component include:

- A menu that posts different actions depending on the user's choice.
- A calendar that lets a user input start and end dates.
- A table view that displays records returned by a database query.

To illustrate this feature, consider a simple reusable component, an alert panel:

Figure 2. Alert Panel

The panel is similar to the navigation table introduced above, but as you'll see, most of the component's attributes are customizable.

To use this component, you simply declare its position within the HTML page and give it a name:

```
<HTML>
<HEAD>
    <TITLE>Alert</TITLE>
</HEAD>
<BODY>

<WEBOBJECT NAME = "ALERT"></WEBOBJECT>

</BODY>
</HTML>
```

The declarations file specifies the value for each of the panel's attributes, either by assigning a constant value or by binding the attributes value to a value

determined by the script file (as with the **alertString** and **infoString** attributes below):

```
ALERT: AlertPanel {
    alertString = alertTitle;
    alertFontColor = "#A00000";
    alertFontSize = 6;
    infoString = alertDescription;
    infoFontSize = 4;
    infoFontColor = "#500000";
    tableWidth = "50%";
};
```

The script file defines the **alertTitle** and **alertDescription** instance variables or methods (see "Intercomponent Communication" below for more information about binding attributes to methods), which set the text that's displayed in the upper and lower panes of the alert panel. The **alertDescription** method could, for example, consult a database to determine the release date of the video.

WebObjects Builder makes working with reusable components such as AlertPanel even easier. For the component creator, WebObjects Builder lets you determine which of the reusable component's attributes will be "exported" to clients. You could, for example, export only the **alertTitle** and **infoString** attributes, but not allow clients to set font color, table width, and other attributes. Clients, on the other hand, can simply drag the AlertPanel from WebObjects Builder's palette window into their applications and use the Inspector to set the bindings. They don't need to manually edit the declarations file to set these bindings. See "Advanced WebObjects Builder Tasks" in the *WebObjects Builder Guide* for more information.

AlertPanel is one of several components included in the Reusable Components Examples. If you take a look at the source code for AlertPanel, you'll notice that it's moderately complicated and in fact relies on other reusable components for its implementation. However, WebObjects lets you think of the AlertPanel component as a black box. You simply position the component in your HTML template, specify its attributes in the declarations file, and implement any associated dynamic behavior in the script file.

Intercomponent Communication

Reusable components can vary widely in scope, from as extensive as an entire HTML page to as limited as a single character or graphic in a page. They can even serve as building blocks for other reusable components. When a reusable

component is nested within another component, be it a page or something smaller, the containing component is known as the *parent component*, and the contained component is known as the *child component*. This section examines the interaction between parent and child components.

In the AlertPanel example above, you saw how the parent component, in its declarations file, sets the attributes of the child component:

```
ALERT: AlertPanel {
    alertString = alertTitle;
    alertFontColor = "#A00000";
    alertFontSize = 6;
    infoString = alertDescription;
    infoFontSize = 4;
    infoFontColor = "#500000";
    tableWidth = "50%";
};
```

Each of the AlertPanel's attributes is set either statically (for example, alertFontSize = 6) or dynamically, by binding the attribute's value to a variable or method invocation in the parent's script file (for example, `alertString = alertTitle`). Communication from the parent to the child is quite straightforward.

But for reusable components to be truly versatile, there must also be a mechanism for the child component to interact with the parent, either by setting the parent's variables or invoking its methods, or both. This mechanism must be flexible enough that a given child component can be reused by various parent components without having to be modified in any way. WebObjects provides just such a mechanism, as illustrated by the following example.

Consider an AlertPanel component like the one described above, but with the added ability to accept user input and relay that input to a parent component. The panel might look like this:

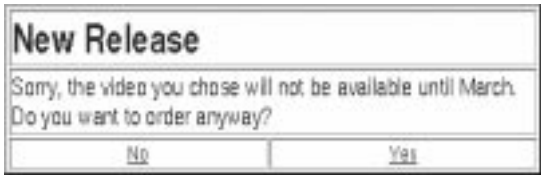

Figure 3. Alert Panel That Allows User Input

As in the earlier example, you use this component by simply declaring its position within the HTML page:

Parent's Template File

```
<HTML>
<HEAD>
    <TITLE>Alert</TITLE>
</HEAD>
<BODY>

<WEBOBJECT NAME = "ALERT"></WEBOBJECT>

</BODY>
</HTML>
```

The corresponding declarations file reveals two new attributes (indicated in bold):

Parent's Declarations File (excerpt)

```
ALERT: AlertPanel {
    infoString = message;
    infoFontSize = 4;
    infoFontColor = "#500000";
    alertString = "New Release";
    alertFontColor = "#A00000";
    alertFontSize = 6;
    tableWidth = "50%";
    parentAction = “respondToAlert”;
    exitStatus = usersChoice;
};
```

The **parentAction** attribute identifies a *callback* method, one that the child component invokes in the parent when the user clicks the Yes or No link. The **exitStatus** attribute identifies a variable that the parent can check to discover which of the two links was clicked. This attribute passes state information from the child to the parent. A reusable component can have any number of callback and state attributes , and they can have any name you choose.

Now let's look at the revised child component. The template file for the AlertPanel component has to declare the positions of the added Yes and No hyperlinks. (Only excerpts of the implementation files are shown here.)

Child Component's Template File (excerpt)

```
    <TD>
        <WEBOBJECT name=NOCHOICE></WEBOBJECT>
    </TD>
    <TD>
        <WEBOBJECT name=YESCHOICE></WEBOBJECT>
    </TD>
```

The corresponding declarations file binds these declarations to scripted methods:

Child Component's Declarations File (excerpt)

```
NOCHOICE: WOHyperlink {
    action = rejectChoice;
    string = "No";
};

YESCHOICE: WOHyperlink {
    action = acceptChoice;
    string = "Yes";
};
```

And the script file reveals the implementation the **rejectChoice** and **acceptChoice** methods:

Child Component's Script File (excerpt)

```
id exitStatus;
id parentAction;

- rejectChoice
{
    exitStatus = NO;
    return [self performParentAction:parentAction];
}

- acceptChoice
{
    exitStatus = YES;
    return [self performParentAction:parentAction];
}
```

Note that **exitStatus** and **parentAction** are simply component variables. Depending on the method invoked, **exitStatus** can have the values YES or NO. The **parentAction** variable stores the name of the method in the parent component that will be invoked by the child. In this example **parentAction** identifies the parent method named **"respondToAlert"**, as specified in the parent's declarations file. **Note:** You must enclose the name of the parent's action method in quotes, as in the example above.

Now, looking at the **rejectChoice** and **acceptChoice** method implementations, you can see that they are identical except for the assignment to **exitStatus**. Note that after a value is assigned to **exitStatus**, the child component sends a message to itself to invoke the parent's action method, causing the parent's **respondToAlert** method to be invoked. Since the parent's **usersChoice** variable is bound to the value of the child's **exitStatus** variable (see the parent's declaration file above), the parent script can determine which of the two links was clicked and respond accordingly. The following diagram illustrates the connections between the child and parent components.

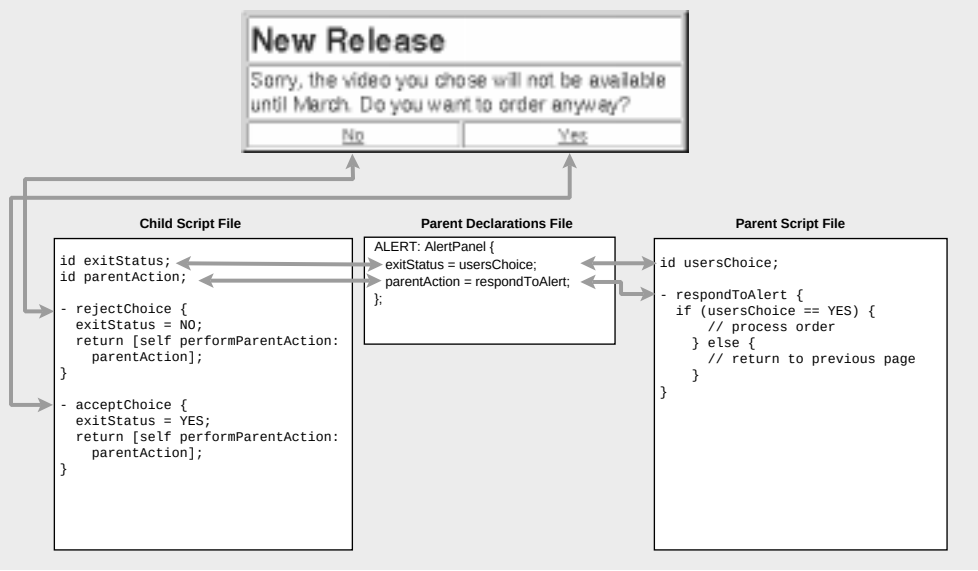

Figure 4. Parent and Child Component Interconnections

The child component's **parentAction** attribute provides a separation between a user action (such as a click on a hyperlink) within a reusable component and the method it ultimately invokes in the parent. Because of this separation, the

same child component can be used by multiple parents, invoking different methods in each of them:

Parent1's Declarations File (excerpt)

```
ALERT: AlertPanel {
    ...
    parentAction = “respondToAlert”;
    exitStatus = usersChoice;
};
```

Parent2's Declarations File (excerpt)

```
ALERT: AlertPanel {
    ...
    parentAction = “okCancel”;
    exitStatus = result;
};
```

Parent3's Declarations File (excerpt)

```
ALERT: AlertPanel {
    ...
    parentAction = “alertAction”;
    exitStatus = choice;
};
```

In summary, parent and child components communicate in these ways:

A parent component can, in its declarations file, set child component attributes by:

- Assigning constant values
- Binding an attribute to the value of a variable declared in the parent’s script file
- Binding an attribute to the return value of a method defined in the parent’s script file

A child component can communicate actions and values to a parent component by:

- Invoking the parent’s callback method
- Setting variables that are bound to variables in the parent, as specified in the parent’s declarations file

Synchronizing Attributes in Parent and Child Components

WebObjects treats attribute bindings between parent and child components as potentially two-way communication paths and so synchronizes the values of the bound variables at strategic times during the request-response loop. This synchronization mechanism has some implications for how you design components.

For the sake of illustration, consider a page that displays a value in two different text fields--one provided by the parent component and one by the child:

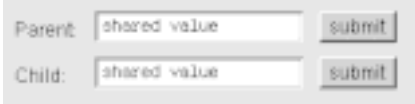

Figure 5. Synchronized Components

Setting the value of either text field and submitting the change causes the new value to appear in both text fields.

The parent's declarations file reveals the binding between the two components:

```
CHILDCOMPONENT: ChildComponent {
    childValue=parentValue;
};
```

When a value is entered in a field and the change submitted, WebObjects will, if needed, synchronize the value in the parent (**parentValue**) and child (**childValue**) at each of the three stages of the request-response loop:

- Before and after the components receive the **takeValuesFromRequest:inContext:** message.
- Before and after the components receive the **invokeAction:inContext:** message.
- Before and after the components receive the **appendToResponse:inContext:** message.

Synchronization is accomplished through key-value coding, a standard interface for accessing an object's properties either through methods designed for that purpose or directly through its instance variables. (The key-value coding mechanism is declared in the Enterprise Objects Framework, in **EOKeyValueCoding.h**. See the *Enterprise Objects Framework Developer's Guide* for more information.) Key-value coding always first attempts to set properties through accessor methods, only reverting to accessing the instance variables directly if the required accessor method is missing.

Given that synchronization occurs several times during each cycle of the request-response loop and that key-value coding is used to accomplish this synchronization, how does this affect for the design of reusable component? It has these implications:

- You rarely need to implement accessor methods for your component's instance variables. For instance, it's sufficient in the example above to simply declare a **childValue** instance variable in the child component and a **parentValue** instance variable in the parent. You only need to implement accessor methods (such as **setChildValue:** and **childValue**) if the component must do some calculation (say, determine how long the application has been running) before returning the value.
- If you do provide accessor methods, they should have no unwanted side effects and should be implemented as efficiently as possible since they will be invoked several times per transaction.
- If you bind a component's attribute to a method rather than to an instance variable, you must provide both accessor methods: one to set the value and one to return it. Let's say the parent component in the example above doesn't have a discrete **parentValue** instance variable but instead stores the value in some other way (for example, as an entry in an NSDictionary object). In that case, the parent component must provide both a **parentValue** method (to retrieve the value) AND a **setParentValue:** method (to set it). During synchronization, WebObjects expects both methods to be present and will raise an exception if one is missing.

Search Path for Reusable Components

When WebObjects encounters the name of a reusable component at run time:

```
NAVCONTROL: NavigationControl {};
```

it must find a WOComponent object to represent the component and then find the component's resources (HTML template file, image files, etc.).

To find an object to represent the component, WebObjects looks in the Objective-C run time for a subclass of WOComponent with the same name as the component ("NavigationControl" in the example above). For compiled reusable components this search should succeed, but for scripted ones it should fail. For scripted components, WebObjects provides its own private subclass of WOComponent.

Next, WebObjects looks within the application directory for the reusable component's resources. For example, if you manually start an application that resides in ***Doc_Root*/WebObjects/MyWOApps/Fortune.woa**, the **Fortune.woa** directory will be searched.

WebObjects pages and reusable components can be located in subdirectories within the application directory. For example, assuming you use different navigation controls for different parts of your application, you might specify the navigation control for your application's catalog pages as:

```
NAVCONTROL: CatalogPages/ReusableComponents/NavigationControl {};
```

This causes WebObjects to search the application directory's **CatalogPages/ReusableComponents** subdirectory for the NavigationControl's resources. You'll find that grouping reusable components within subdirectories like this helps keep your application directories organized.

Designing for Reusability

Here are some points to consider when creating reusable components.

Make sure that your reusable component generates HTML that can be embedded in the HTML of its parent component.

A reusable component should be designed to be a "good citizen" within the context in which it will be used. Thus, for example, the template file for a reusable component should not start and end with the <HTML> and </HTML> tags (since these tags will be supplied by the parent component). Similarly, it is unlikely that a reusable component's template would contain <BODY>, <HEAD>, or <TITLE> tags.

Further, if you intend your component to be used within a form along with other components, don't declare the form (<FORM...> ... </FORM>) within the reusable component's template file. Instead, let the parent component declare the form. Similar considerations pertain to submit buttons. Since most browsers allow only one submit button within a form, putting a submit button in a reusable component severely limits where it can be used.

Guard against name conflicts.

Reusable components are identified by name and location. (see "Search Path for Reusable Components"). Those that reside within a particular application's application directory are only available to that application. Those that reside in ***Doc_Root*/WebObjects** are available to all applications on that server. Since no two

component directories can have the same name in ***Doc_Root*/WebObjects**, shared reusable components must have unique names. Consider adding a prefix to component names to increase the likelihood that they will be unique.

Provide attributes for all significant features.

The more customizable a component is, the more likely it is that people will be able to reuse it. For example, if the AlertPanel component discussed above let you set the titles of the hyperlinks (say, to OK and Cancel or Send Now and Send Later), the panel could be adapted for use in many more applications.

Provide default values for attributes wherever possible.

Don't require people to set more attributes than are strictly required by the design of your reusable component. In your component's **init** method, you can provide default values for optional attributes. When the component is created, the attribute values specified in the **init** method are used unless others are specified in the parent's declarations file.

For example, the AlertPanel's **init** method could set these default values:

```
- init {
    [super init];
    alertString = @"Alert!";
    alertFontColor = @"#ff0000";
    alertFontSize = 6;

    infoString = @"User should provide an infoString";
    infoFontColor = @"#ff0000";
    infoFontSize = 4;

    borderSize = 2;
    tableWidth = @"50%";
    return self;
}
```

Then, in a declarations file, you are free to specify all or just a few attributes. This declaration specifies values for all attributes:

Complete Declaration

```
ALERT: AlertPanel {
    infoString = message;
    infoFontSize = 4;
    infoFontColor = "#500000";
    alertString = "New Release";
    alertFontColor = "#A00000";
```

```
    alertFontSize = 6;
    tableWidth = "50%";
};
```

This declaration specifies a value for just one attribute; all others will use the default values provided by the component's **awake** method:

Partial Declaration

```
ALERT: AlertPanel {
    alertString = "Choice not available.";
};
```

Consider building reusable components from reusable components.

Rather than build a monolithic component, consider how the finished component can be built from several, smaller components. You may be able to employ these smaller components in more than one reusable component.

Take, for example, the AlertPanel example (see the Reusable Components Examples to view the source code for this component). The AlertPanel lets you not only set the message displayed to the user, but the message's font size and color. These font handling features aren't provided by the AlertPanel itself but by an embedded reusable component, FontString. FontString itself is a versatile component that's used in many other components.

Document the reusable component's interface and requirements.

If you plan to make your components available to other programmers, you should provide simple documentation that includes information on:

- What attributes are available and which are required
- What are the default values for optional attributes
- What context needs to be provided for the component. For example, does it need to be embedded in a form?
- Any restrictions that affect its use. For example, is it possible to have a submit button in the same form as the one that contains this component?

In addition, it's helpful if you provide an example showing how to use your component.