The Skin Pattern

By Rani Pinchuk and Yonat Sharon

Abstract

The Skin Pattern is a technique to separate the presentation style of an application (its "skin") from its logic. Keeping the skins separate makes the application code closed to cosmetic changes in the UI, and the UI closed to bug fixes and changes in the application code. The code becomes more readable and easier to maintain. Further, the people who define and maintain the skins do not need to understand any programming language, since there is no code in the skins. Graphic tools may be used to generate and modify the skins, thus providing intuitive and friendly tools for UI designers.

SKIN

Intent

Separate the presentation style of an application from its logic.

Motivation

Problem

Web applications dynamically create HTML pages. One common technique to do this is to insert HTML code inside the application code, sometimes using special method calls, as shown in listings 1-2.

Another common technique is to write the application within the HTML pages by inserting program instructions in the HTML code, as shown in listings 3-4. When the application is run, the program code and the HTML code are read together, and generate the final HTML pages.

```
...
totalPrice = price * qty;
htmlHeading  h1( 1 );
h1 << "The total price is" << totalPrice << "$";
```

Listing 1 - C++ code using html++ (1) library

```
...
$total_price = $price * $qty;
$cgi->h1("The total price is $total_price\$");
```

Listing 2 - Perl code using CGI.pm (2) library

```
<CFSET total_price = (#qty# * #price#)>
<CFOUTPUT>
<H1>The total price is #total_price#$</H1>
</CFOUTPUT>
```

Listing 3 - ColdFusion (**Error! Reference source not found.**) code

```
<%...
total_price = qty * prices
%>
<H1>The total price is <%=total_price%>$</H1>
```

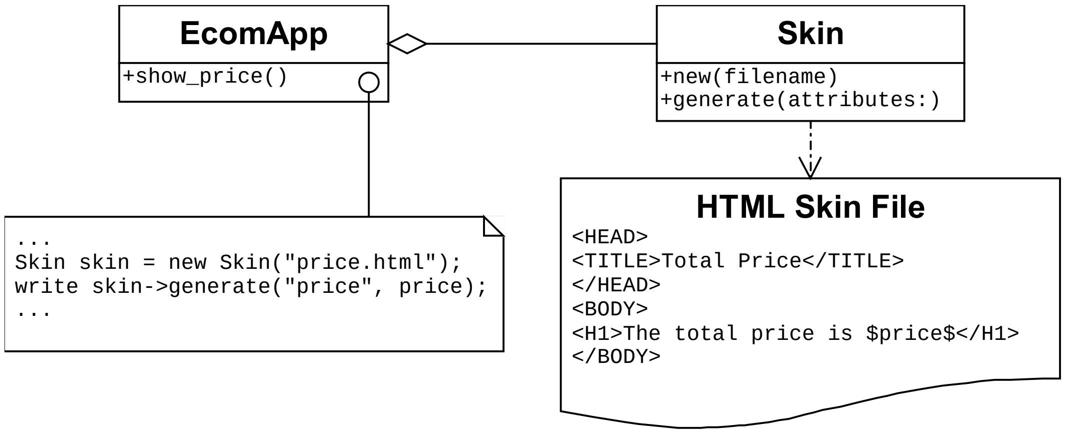
Listing 4 - ASP (4) code

Both techniques cause a situation where the application code suffers in its readability and its maintainability: It is difficult to see the logic of the application when the presentation code is mixed inside. Moreover, usually the programmer is not responsible for the look of the application, and the graphic designer will not want to see the program code.

Solution

We can separate the logic of the application from its presentation style by introducing **skin**s as shown in Figure 1. In the Web application example the skins will be pages written in HTML with some simple extensions that work mainly as placeholders for application-supplied data. A Skin object will parse one of those pages, and then the application can use the Skin object to generate the output HTML by supplying the actual data to fill in the placeholders. Note that although we add extensions to the HTML code, they are only used for displaying application-supplied data, not for manipulating that data. This way the application logic does not creep into the HTML page.

The skin page defines a general HTML page, while the Skin object enables the application to generate a specific HTML page with specific data.



```
...
Skin skin = new Skin("price.html");
write skin->generate("price", price);
...
```

```
<HEAD>
<TITLE>Total Price</TITLE>
</HEAD>
<BODY>
<H1>The total price is $price$</H1>
</BODY>
```

Figure 1 - Generating HTML pages from a skin

Applicability

Use this pattern when

- The application generates views; and
- Views that are generated in different conditions have a similar structure ("form-views" or "reports") so we can define a generalization for them. A simple generalization can be specified when the views are identical in everything except for:
  - Specific data that may differ between views.
  - The number of times certain sections appears in each view (where this number may be zero).

The pattern deals only with the generation of these views. Specifying interactive behavior of the views after they are created is orthogonal to using Skin for view generation.

Structure

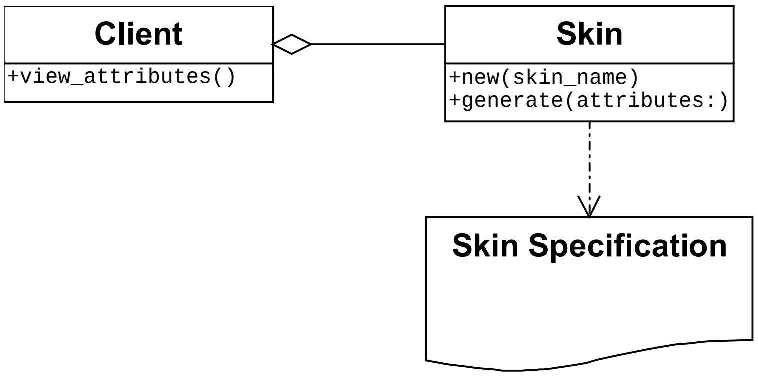


Figure 2 - Class structure of the Skin pattern

Participants

- **Skin Specification**
  - Defines a general view structure.
- **Skin (object)**
  - Generates a specific view according to the attributes that are provided by the client.
- **Client**
  - Calculates view-specific values and creates views using Skin objects.

Collaboration

1. The client calculates view-specific data.
2. The client creates a Skin object associated with the skin for presenting this data (if it does not yet exist).
3. The client passes view-specific data to the Skin object.
4. The Skin object generates the view.

Consequences

1. *Separation of domains.* The presentation skins are separate from the application logic so it is easier to change and manage them independently. There is less danger that a graphic redesign will introduce new bugs or that code re-factoring and bug fixing will affect the graphic presentation. This also makes the application code more readable.
2. *Easier graphic design*. It is not necessary for user interface specialists to dig the graphic definitions from within the code in order to maintain them. Instead they only need to manipulate the skins. Friendly graphical tools can be used for this purpose (like WYSIWYG HTML editors).
3. *Personalization and localization*. Since skins can be parsed at run-time, it is possible

to supply several skins for a view and allow the user to choose the one she prefers. In simple cases, this can be used for localization.

4. *Skin specification language.* A mechanism to specify the skin is needed. This can be an existing language, an extension of an existing language, or a new specification language altogether. See the Implementation section for more details.
5. *More complex design.* The extra level of indirection introduced by the Skin object adds to the complexity of the design.

Implementation

Implementation considerations regarding the specification language of the skins (hereafter *the language)*:

1. *Level of flexibility*. We can imagine a flexibility-scale of the language: At one extreme we will have a language that will only allow to substitute view-specific values in place of named elements in the skin (like substituting the actual title in `<h1>$title</h1>`). At the other extreme is a Turing complete language. We should try to keep the language as close as possible to the first extreme, since the second extreme allows application logic to creep into the skin, and this contradicts the intent of this pattern.

   A good limit to the flexibility of the language is to keep it without the ability to change data. The reason is that any algorithm has states that are described using variables, and state-transitions are made by changing these variables. If we cannot change variables, we cannot define algorithms, and so the skin cannot contain application logic. This limitation does not inhibit having constants and control structures like if-else blocks and while loops (where the loop condition is determined by a callback function), like the ones shown in the Sample Code section.

2. *Value placeholders vs. named values*. To substitute view-specific data into the skin, the language can either

   - Have special escape sequences to denote placeholders for application supplied values (for example, `The total price is $price$`); or
   - Have a mechanism to name specific elements so they can be changed by the application (for example, `The total price is <a name="price">0.00</a>$`).

   In the first case, the skin itself cannot be used as a view without putting all the view-specific values into it. In the second case, the skin itself defines a complete view and the application only needs to replace some of its named parts with view-specific values.

3. *Using an existing language*. If there is already an existing view specification language (like HTML), it may be possible to use it for defining skins as-is or with simple extensions. As a minimum, this language needs to allow naming specific elements, so that they can be replaced with view-specific values. If the existing language does not support all the desired specification instructions then it can be extended by putting these instructions inside escape sequences (as demonstrated in the Sample Code

section). Thus, the extended language (original language with extensions inside escape sequences) becomes the skin specification language.

If the original language does not include a flexible mechanism for inserting escape sequences, the skins may not be legal documents in the original language. The generated documents, however, will not include these escape sequences and can therefore be legal documents in the original view specification language.

Languages like PHP, ASP, JSP and CFML can be used for skin specification only if you restrain yourself from slipping application logic into the skins. To do that you should avoid using any feature that changes data, as explained in item 1.

4. *Creating a new language*. If there is no existing language for defining the views, or the existing language is too complicated (perhaps a more general presentation language) then a new language is needed. This language can be tailored to the needs of defining skins. This approach was chosen for several GUI application frameworks and operating systems, where skins are sometimes called "resource templates" or "window specs" (see Known Uses section for details).
5. *Nesting skins*. To support sub-forms (5) it should be possible to nest one skin inside another. There are two ways to do this:

One) The simplest way is first to generate the sub-skin, and then substitute it into the main skin as a parameter. The responsibility for selecting the sub-skin is on the main application code. However, if the views themselves are not text-based, then two generation mechanism will be needed: skin specification to skin specification (to insert sub-skins into the main skin) and skin specification to view.

Two) Another possibility is to have an "include" directive in the skin specification language. The responsibility for selecting sub-skins is then in the specification of the parent skin.

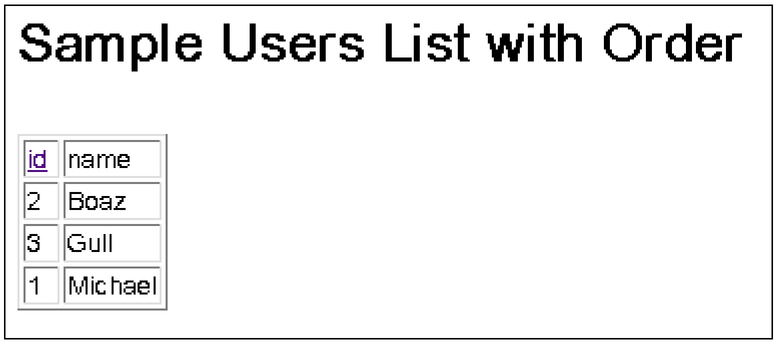
Implementation considerations regarding the application code:

1. *Parsing the skin*. To generate a view from the skin, the skin first needs to be parsed. This can be done every time view-specific data is passed to the skin object, or it can be done only once, when the skin object is initialized with a specific skin. In the second case, some abstract representation of the skin will be kept in the skin object and used every time it generates a view.
2. *Displaying the views*. After the views are generated, they may need to be displayed. In the simple case where the views are always displayed right after generation (and never stored, transferred, displayed on a different target etc.), the skin object may hold the responsibility to display the views. However, since the rule "always display after generation" is often likely to change, it is better to keep the responsibility for display out of the skin object.

Sample Code

Example 1 – Writing Web applications using the TextTemplate class

The following example shows a part of an application that generates an HTML page that presents a list of items in a table. Clicking on the title of one of the columns will sort the table according to that column:



Sample Users List with Order

id	name
2	Boaz
3	Gull
1	Michael

Figure 3 - Sample HTML page view

The Skin specification for this page is as follows:

```
<HTML>
  <HEAD>
    <TITLE> $title </TITLE>
  </HEAD>
  <BODY>
    <H1> $title <H1>
    <TABLE border="1">
      <TR>
        <TD>
          <IF condition="is_order_id">id</IF>
          <ELSE><A HREF="order.pl?order=id">id</A></ELSE>
        </TD>
        <TD>
          <IF condition="is_order_name">name</IF>
          <ELSE><A HREF="order.pl?order=name">name</A></ELSE>
        </TD>
        <WHILE condition="user_list">
          <TR><TD>$id</TD><TD>$name</TD></TR>
        </WHILE>
    </TABLE>
  </BODY>
</HTML>
```

Listing 5 - Sample HTML skin specification

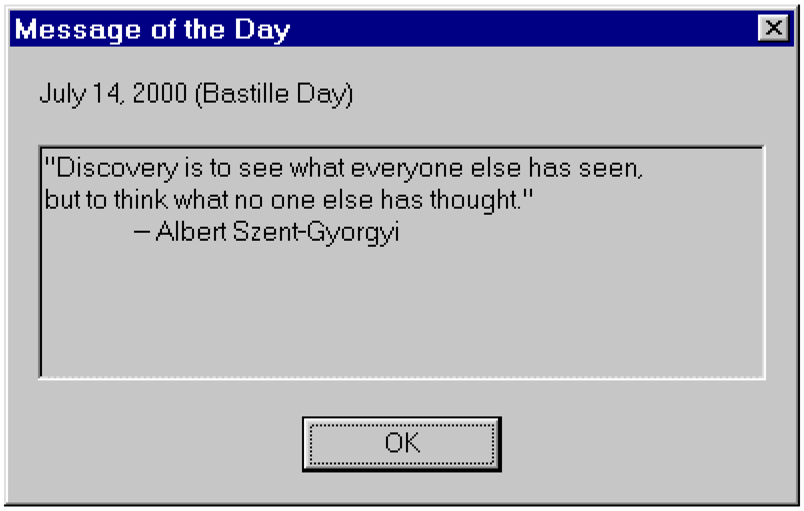
The code that uses this Skin specification to show the above view is as follows:

```
#!/usr/local/bin/perl

use TextTemplate;
use CGI;
```

```
# a list of sample users:
my @List = ( { id => 1, name => "Michael" },
             { id => 2, name => "Boaz" },
             { id => 3, name => "Gull" } );

my $title = "Sample Users List with Order"; # the page title
my $i = 0; # the counter i of the list - we start with 0
my $id; # the id of the current row
my $name; # the name of the current row

# the sorting order:
my $cgi = new CGI; # create the CGI object
print $cgi->header(); # print the HTTP header
my $order = $cgi->param("order") || "id"; # the default order is by "id"

# create new TextTemplate object
my $template = new TextTemplate("users_list.tmp");

# read the template, parse it and print the generated results. we send
# in an anonymous hash all the variables that we would like to use in
# the template, or that we should use in the call back function.
print $template->generate({ title => $title,
                            order => $order,
                            is_order_id => $order eq "id",
                            is_order_name => $order eq "name",
                            i => $i,
                            id => $id,
                            name => $name,
                            user_list => \&list_call_back_func });

# update $id and $name to the next item in the list
sub list_call_back_func {
    my $variables = shift; # always a reference to the anonymous hash
    # that we send with the parse method is
    # send to the callback functions.
    my $i = $variables->{i}; # the counter
    if ($i < scalar(@List)) {
        # create an ordered list from the global list
        my @list;
        if ($order eq "id") {
            @list = sort {$a->{id} <=> $b->{id}} @List;
        }
        elsif ($order eq "name") {
            @list = sort {$a->{name} cmp $b->{name}} @List;
        }

        # get the name and the id for that counter value from the
        # global List, when the list is ordered by the order.
        $variables->{id} = $list[$i]->{id};
        $variables->{name} = $list[$i]->{name};

        $variables->{i}++; # increment the value of id
        return 1; # continue to loop
    }
    else {
        return 0; # we finished
    }
}
```
Listing 6 - Sample Perl code using Skin

Example 2 - Message of the Day application using MFC

The MFC GUI framework (6) uses skins for dialog boxes. The skins are called " dialog templates" and are parsed by objects of class `CDialog` that serves as the Skin class. Here is a simple template for the "Message of the Day" dialog box shown in Figure 4:



July 14, 2000 (Bastille Day)

"Discovery is to see what everyone else has seen,
but to think what no one else has thought."
— Albert Szent-Gyorgyi

Figure 4 - Sample dialog box view

The corresponding dialog template is:

```
IDD_MOTD_DIALOG DIALOGEX 0, 0, 196, 113
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
EXSTYLE WS_EX_APPWINDOW
CAPTION "Message of the Day"
FONT 8, "MS Sans Serif"
BEGIN
    DEFPUSHBUTTON   "OK",IDOK,73,92,50,14
    LTEXT           "date",IDC_DATE,7,7,182,8
    EDITTEXT        IDC_MESSAGE,7,24,182,59,ES_MULTILINE |
                    ES_AUTOVSCROLL | ES_AUTOHSCROLL | ES_READONLY |
                    NOT WS_TABSTOP
END
```

Listing 7 - Sample dialog template skin

The template starts with the ID for the resource template (`IDD_MOTD_DIALOG`), its type (`DIALOGEX`) and coordinates. It is followed by lines specifying attributes of this dialog box and a BEGIN-END block that specifies its contents: a button, a static text area labeled `IDC_DATE` (for the date), and an editable text area labeled `IDC_MESSAGE` (for the message).

To create a dialog box using this template, all we need to do is:

```
CDialog dlg(IDD_MOTD_DIALOG);
Dlg.DoModal();
```

Listing 8 - Sample code to display the dialog box

However, this will not allow us to manipulate the values of the dialog contents (specifically, the date and message). To do this, we derive a new class from `CDialog`, that will serve as the Client:

```
class CMotdDlg : public CDialog
{
public:
	CMotdDlg() : CDialog(IDD_MOTD_DIALOG) {}
protected:
	virtual BOOL OnInitDialog();

	// ...
};
```

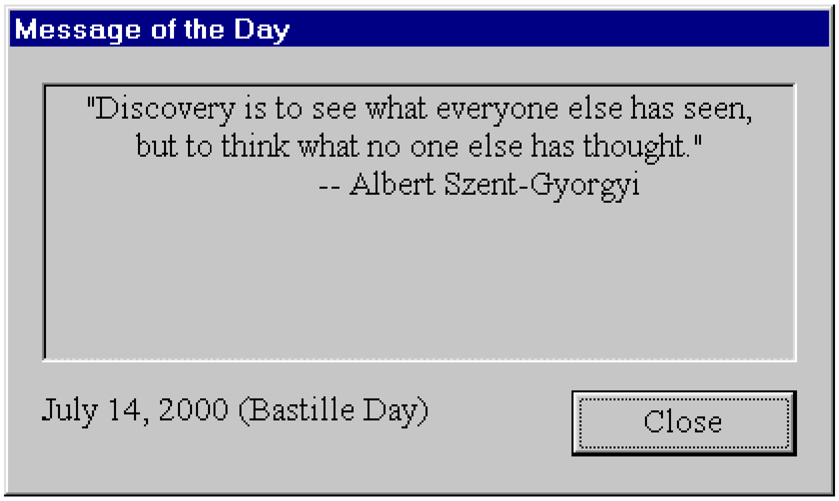
Listing 9 - Sample dialog box class

The `OnInitDialog()` member function is where we put the application supplied data into the date and message parts of the dialog box:

```
BOOL CMotdDlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	SetDlgItemText(IDC_DATE, theApp.GetDay());
	SetDlgItemText(IDC_MESSAGE, theApp.GetMessage());

	return TRUE;
}
```

Listing 10 - Sample code to substitute application data into skin placeholders

We can easily change the appearance of the dialog box without touching application code. For example, to make the dialog box look like Figure 5



"Discovery is to see what everyone else has seen, but to think what no one else has thought."
-- Albert Szent-Gyorgyi

July 14, 2000 (Bastille Day)

Figure 5 - Same view with a different skin

we changed its template to:

```
IDD_MOTD_DIALOG DIALOGEX 0, 0, 182, 93
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION
EXSTYLE WS_EX_APPWINDOW
CAPTION "Message of the Day"
FONT 10, "Times New Roman"
BEGIN
    DEFPUSHBUTTON   "Close",IDOK,125,72,50,14
    LTEXT           "date",IDC_DATE,7,72,107,14
    EDITTEXT        IDC_MESSAGE,7,7,168,59,ES_CENTER | ES_MULTILINE |
                    ES_AUTOVSCROLL | ES_AUTOHSCROLL | ES_READONLY | NOT
                    WS_TABSTOP
END
```
Listing 11 - Sample of a different dialog resource skin

The dialog templates themselves are usually manipulated using graphic editors, so the user interface designers do not even need to learn the dialog templates format.

Known Uses

The TextTemplate class (shown in the Sample Code section) is used by EM-TECH group for several Web application projects. One example is the DHL Millenium Information Center that was used by DHL world wide in the first days of the millenium in order to have real time status reports from all over the world.

Both Apple MacOS (7) and Microsoft Windows (8) use skins for dialog boxes. The skins are called *dialog templates* and are specified in *resource files* (several skins can be defined in one resource file).

Several open source projects implement the skin pattern, among them:
- HTML::Template in Perl (https://sourceforge.net/project/?group_id=1075)
- MacroSystem in C++ (https://sourceforge.net/project/?group_id=2146)
- FreeMarker in Java (http://freemarker.sourceforge.net/).

XSL (9) can be used to specify skins. The generated views are in XML or HTML. The XSL processor acts as the Skin object. The client generates documents by marshalling the view-specific data to an XML document and invoking the XSL processor on this document.

Related Patterns

Sometimes the similarities between different views are not simple, and so a common structure cannot be easily formulated. In these cases it may be better to construct views from common elements using Builder (10) – where each view is constructed element-by-element, or using Phrasebook (11) – where each element is an expression in the foreign language phrasebook.

The Skin can use the Phrasebook pattern (11) to access the skins from the application code. The skin specification language is the foreign language and each skin is equivalent to a foreign language phrase.

The View part of MVC (12) or Model-View can use a skin. The View, acting as Observer (10), is then responsible for filling in the dynamic parts of the presentation into the static parts that are determined by Skin.

The Subform and Reusable Subform patterns (5) can be used in concert with Skin. Each subform has its associated skin, and the nesting of skins mirrors the subforms hierarchy. (See Implementation note 5.)

Acknowledgements

We would like to thank our PLoP2000 shepherd Mark Bradac and our meta-shepherd Dwight Deugo for their helpful comments, and the writers' workshop participants for their valuable feedback.

We thank EM-TECH group for providing the environment where the TextTemplate class could be created, and for providing resources that helped in writing the pattern.

References

1. DC Micro Development. *html++ CGI Class Library Online Manual*. DC Micro Development, 1998. http://www.dcmicro.com/htmlpp/manual.html.
2. L. Stein. *CGI.pm - a Perl5 CGI Library*. Cold Spring Harbor Laboratory, 2000. http://stein.cshl.org/WWW/software/CGI/.
3. Allaire Corporation. *Developing Web Applications with ColdFusion*. Allaire Corporation, 1999. http://www.allaire.com/developer/documentation/ColdFusion.cfm.
4. Microsoft. *Active Server Pages Tutorial*. Redmond, MA: Microsoft, 2000. http://msdn.microsoft.com/workshop/server/asp/asptutorial.asp.
5. M. Bradac and B. Fletcher: "A Pattern Language for Developing Form Style Windows." In R.C. Martin, D. Riehle, and F. Buschmann (eds.), *Pattern Languages of Program Design 3*. Reading, MA: Addison-Wesley, 1998.
6. MSDN Library. *Microsoft Foundation Class Library.* Redmond, MA: Microsoft, 1998. http://msdn.microsoft.com/library/devprods/vs6/visualc/vcmfc/mfchm.htm.
7. Apple Computer, Inc. *Inside Macintosh: Macintosh Toolbox Essentials*. Apple Developer Connection, 1996. http://developer.apple.com/techpubs/mac/Toolbox/Toolbox-2.html.
8. MSDN Library. *Platform SDK: Windows User Interface.* Redmond, MA: Microsoft, 1998. http://msdn.microsoft.com/library/psdk/winui/resource_1inn.htm.
9. M. Froumentin. *Extensible Stylesheet Language (XSL)*. The World Wide Web Consortium, 2000. http://www.w3.org/Style/XSL.
10. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.
11. R. Pinchuk and Y. Sharon. *The Phrasebook Pattern*. In this volume.
12. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern Oriented Software Architecture*. New York: John Wiley & Sons, 1996.