Objects and the Web

Alan Knight, *Cincom Systems*

Naci Dai, *ObjectLearn*

Applying software engineering principles, particularly object-oriented techniques, to the Web can be difficult. Many current Web technologies lend themselves to—or even encourage—bad practices. Scripting and server-page technologies can encourage cut-and-paste reuse, direct-to-database coding, and poor factoring. Component models such as the Component Object Model (COM) and Enterprise JavaBeans (EJB) seek to construct building blocks for application assembly, but in doing so they sacrifice many of the advantages of objects. XML emphasizes technology-independent reuse and sharing of content, data, and messaging but at the expense of encapsulation and the association of behavior with state, which is central to OO.

Good design practices are increasingly important in Web development. Here, the authors apply such practices using a framework for layered architectures based on the Smalltalk GUI development pattern of Model-View-Controller.

Scripting languages, common on the Web, are often optimized for rapidly creating simple functionality rather than for modular construction of large programs. Also, such languages typically lack the rich development environments of general-purpose languages. Some Web developers even deliberately disregard software engineering principles. They argue that if we're just writing scripts, they don't merit the kind of engineering techniques—object or otherwise—we apply to "real" systems.

However, software engineering and OO techniques are gaining importance in Web development as Web applications become more complex and integrated with traditional server-side applications. To motivate the need for these techniques, we examine some representative Web technologies and the issues they present in naive use. We describe a layered, OO architecture, based on the Model-View-Controller (MVC) pattern, which can overcome these issues to produce large, well-structured systems.

Motivations

If we consider scripts from an OO and layering perspective, the most immediate problem is that a single script has responsibilities spanning several layers (see the "Definitions" sidebar for an explanation of our terminology). The script must

- Accept input
- Handle application logic
- Handle business logic
- Generate output (presentation logic)

This couples all the layers together, making

Definitions

Here we define our terminology and goals. In the main text, we present a layered architecture, implemented using both scripts and server pages. Most of these techniques can be applied to any technology in these categories, but where the details of a specific technology are important, we use servlets and JavaServer Pages as representative technologies. Both, while nominally Java specifications, can easily be applied to other languages, and implementations in both Smalltalk and C++ are commercially available.

Layered architecture

A layered architecture is a system containing multiple, strongly separated layers, with minimal dependencies and interactions between the layers. Such a system has good *separation of concerns*, meaning that we can deal with different areas of the application code in isolation, with minimal or no side effects in different layers. By separating the system's different pieces, we make the software adaptable so that we can easily change and enhance it as requirements change. The layers we are concerned with here include input, application logic, business logic, and presentation.

Input

The input layer contains the code concerned with processing and syntactically validating input. In a Web context, this processing and syntactically validating input includes HTTP input parsing and extracting parameters from an HTTP request. In the Model-View-Controller framework, this corresponds to the input controller.

Application logic

The application logic code is concerned with the application's overall flow. We often refer to it as the glue layer, separating business logic from input and presentation logic and managing the interface between the two. This requires some knowledge of both layers. For example, this layer would be involved in converting between presentation-level inputs and outputs as strings and the corresponding business object messages or state. This layer might also manage a multipage Web interaction as a sequence of steps. In the Model-View-Controller framework, this corresponds to the application controller.

Business logic

The business logic code is concerned only with the underlying business functionality. This code should be entirely unaware of the presentation being used. We also refer to *business objects*, which implement the business logic. In a complex application, business logic is likely to be the largest component and can include code that accesses external systems such as databases, external components, and other services. In the Model-View-Controller framework, this corresponds to the model.

Presentation

This layer contains code and noncode resources (such as HTML text and images) used to present the application. It typically contains little code—code concerned only with formatting and presenting data. An example of this in a Web context is a server page's code fragments that print values into a dynamically generated Web page. In the Model-View-Controller framework, this corresponds to the view.

Scripts

Many basic Web technologies can be grouped together in the category of *scripts*—small programs that perform HTTP processing. This term encompasses, among others, compiled CGI programs, files of scripting language code (such as Perl, Python, Ruby, and VBScript), and Java servlets. While there are significant differences among these technologies, all of them have the fundamental characteristic of being programs that accept an HTTP request and send back a response. In basic usage, each script is stateless and independent of all others. An important distinction within this category is whether scripts can share memory with other scripts (for example, servlets) or are entirely independent (for example, CGI programs). Shared memory allows more effective use of system resources through pooling, and a simpler programming model through persistent states at the cost of a more complex infrastructure.

Server pages

An alternative mode of HTML scripting is that of an HTML page annotated with small amounts of code. Many scripting languages support this kind of facility in addition to pure scripts, and representative examples include JavaServer Pages (JSP), Microsoft's Active Server Pages, PHP, and Zope. There are also variations on this approach in which the pages are annotated with application-specific HTML or XML tags, and developers can specify code to be run when these tags are encountered. Examples of this include JSP bean and custom tags and Enhydra's XMLC.

it harder to modify or test any particular aspect in isolation. In addition, there are significant issues related to handling these responsibilities, as described below. For server pages used alone, the same issues apply (because we can consider a server as a script with some additional embedded HTML), and mixing code with the HTML also presents code management and debugging issues.

Accepting input

When accepting input, a script receives either the raw HTTP input stream or a minimally parsed representation of it. HTTP supports several different mechanisms for passing parameters (encoding into the URL, query values, or form data), and all of these pass the data as simple strings. Each script must know or determine the parameter-passing mechanism, convert the parameters to appropriate types, and validate them. This causes code duplication between scripts.

Handling application logic

Another issue, which affects both input and application logic, is the lack of information hiding when accessing request and session data. The script must retrieve input data from the request by name. HTTP is a stateless protocol, so data used in multiple scripts must be either stored in a session identified with the user or reread from an external data source in each script requiring the data. For example, if a script passes login information as form data, the code to store that information in the session might be

```
password = request.getParameter
  ("passwordField");
decrypted = this.decode
  (password);
request.getSession().putValue
  ("password",decrypted);
```

Both storage in the session and storage in an external data source are effectively global in scope, and the application accesses them in dictionary-like fashion using strings as keys. Normal programming mechanisms for controlling variable access do not apply to this data, and any scripts or server pages that wish to use this data must be aware of the naming conventions. We cannot easily find all accesses to the variables using programming-language mechanisms, so modifications become more difficult. If the script does not encapsulate these conventions, knowledge of them and of the HTTP protocol's details can spread throughout the application, greatly hindering adaptation to new uses. Furthermore, this is a potential source of errors because of both spelling errors and different scripts using the same name for different purposes. As the number of scripts or server pages increases, these problems become overwhelming.

When using server pages for application logic, we are adding potentially significant amounts of code to the page. Code management techniques are not usually available for code inside server pages. With many technologies, debugging code inside the server pages is difficult. Features of modern development environments for code authoring and interactive debugging might not be available, and for compiled languages we might need to debug inside complex generated code. For these reasons, it is a good idea to minimize the amount of code in server pages and to keep application logic out of the pages.

It is a good idea to minimize the amount of code in server pages and to keep application logic out of the pages.

Handling business logic

Because all of the code is grouped together, it is difficult to isolate the business logic from the other layers, particularly application logic. Furthermore, unless we can run portions of the scripts separately, it is impossible to test the business logic (or any of the other layers) independently.

With server pages, business logic presents the same issues that we discussed for application logic. We can very quickly have too much code in the pages, and even pages with minimal code are difficult to manage and debug.

Generating output

In producing output, simple scripts mix the HTML encoding of the result with the dynamic data. This couples the page's look and feel with the other layers. Changing the Web site's look or adapting the application to multiple output devices becomes extremely difficult. The latter is becoming increasingly important as the Web expands to include devices such as WAP (Wireless Application Protocol)-connected mobile phones and other small devices.

Server pages help address this last issue by letting Web designers design and maintain the pages and by letting programmers provide annotations. This is generally considered the most appropriate use for server pages.

Model-View-Controller

To overcome these difficulties, we can use a combination of scripts, server pages, and application code. The approach we present here is part of a family of possible approaches we have used[1,2] to properly partition responsibilities and overcome weaknesses in the underlying technologies (see the "A Related Approach" sidebar for a comparison). MVC strongly influences our approach, so we first examine its history.

MVC concepts

MVC originated in the Smalltalk-80 system to promote a layered approach when developing graphical user interfaces.[3] It emerged in the late 1970s and early 1980s, long before such

A Related Approach

There are many frameworks and design patterns influenced by layering, the Model-View-Controller pattern, and object principles. In general, using servlets and server pages together and keeping code out of the server pages as much as possible is referred to in Java circles as *model 2 Web programming* (see http://java.sun.com/j2ee). The most widely known framework using these principles is the open source Jakarta Struts (see http://jakarta.apache.org/struts).

Struts is a controller framework for Java to build JavaServer Pages-based views that are linked to business models using a single controller servlet. Struts is very close to many of our concepts and also uses a single input controller servlet, but it differs in some significant areas. Most notably, it does not distinguish a separate application controller, distinguishing only input, action, presentation, and model.

Application controller responsibilities are assigned to actions, model objects, or declarative aspects of the framework. Actions are command objects, and Struts suggests that they should delegate as much behavior as possible to the model, but coding examples include application control or even model behavior in some actions. The transition to the next page after an action is controlled by the developer by editing configuration files. One other difference is that Struts classifies model objects into "bean forms" that have only state, and more normal objects with both function and behavior. These bean forms are used to store data for input or presentation processing. We have no corresponding layer and are unclear as to the role of pure state objects in an OO framework.

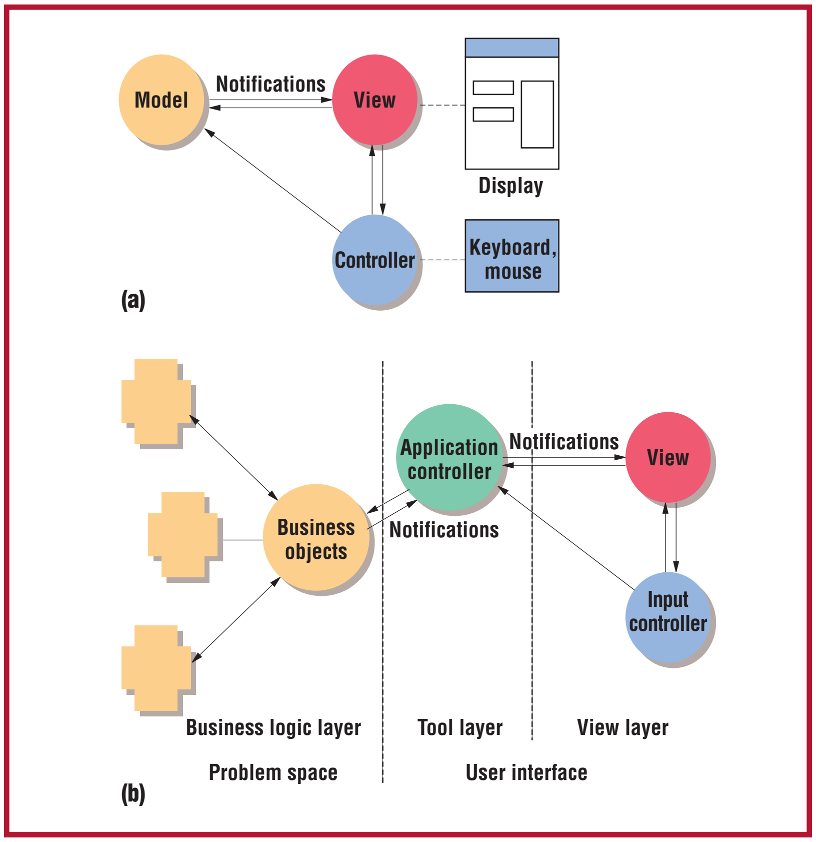

Figure 1. (a) The original Smalltalk-80 Model-View-Controller pattern and (b) the revised Model-View-Controller.

interfaces had become mainstream. It defined three different components (see Figure 1a):

- The *model* handles application and business logic.
- The *view* handles presentation logic.
- The *controller* accepts and interprets keyboard and mouse input.

The intention was to separate the model (meaning nonGUI) code from its presentation. The model code didn't contain any GUI information, but it broadcast notification of any state changes to dependents, which were typically views. This is similar to many current GUI component models, of which JavaBeans is perhaps the most widely known.

This scheme provided a good separation between these three layers but suffered from two weaknesses. First, it had a simplistic view of the model and did not account for any difference between application logic (for example, flow of control and coordination of multiple widgets in a GUI) and business logic (for example, executing a share purchase). Second, most GUI libraries and windowing systems combined the view and controller functions in a single widget, making the logical separation into view and controller less useful. Later versions of Smalltalk with operating system widgets chose not to use a separate controller. All of the Smalltalk versions eventually introduced an additional layer to handle application logic, distinct from the business objects. Perhaps the most elegant of these is Presenter, as used in, for example, Taligent and Dolphin Smalltalk.[4,5]

Together, these revisions changed the common understanding of the framework, such that the term controller now refers to the object handling application logic and the term model is reserved for business objects (see Figure 1b). To distinguish these two interpretations of MVC, we use *model* to refer to business objects, and we use *input controller* and *application controller* to refer to the two types of controllers.

MVC for the Web

To apply MVC to the Web, we use a combination of scripts, server pages, and ordinary objects to implement the various components in a framework we call *Web Actions*. In this context, both versions of the

MVC are relevant, particularly the dual uses of the term *controller*. For HTTP applications, input and presentation are entirely separate, so an input controller distinct from the view is useful. For applications of any complexity, we also need an application controller to separate the details of application flow from the business logic.

Figure 2 shows the framework's basic object structure.

Input controller. We implement the input controller as a script. One of the framework's important features is that there is a single input controller for all pages in a Web application. The input controller parses input, determines the parameter-passing mechanisms, extracts any necessary information from the request, cooperates with the application controller to determine the next action, and invokes that action in the correct context.

By having a single script as an input controller, we localize any knowledge of HTTP or naming conventions at the request level, reduce code duplication and the total number of scripts, and make it easier to modify any of the input processing, because there is a single point of modification.

Note that the input controller class is shown as an abstract class, with one implementation for accessing the applications over HTTP with a regular Web browser and another implementation for accessing the applications using a WAP-enabled device.

Application controller. We implement the application controller as a regular object—not as a script or server page. It coordinates logic related to the application flow, handles errors, maintains longer-term state (including references to the business objects), and determines which view to display. We store the application controller in the session using a key known to the input controller. This relies on a naming convention, with the disadvantages described earlier, but because this is the only thing stored directly in the session, the impact is minimized.

A single application controller is responsible for multiple Web pages. In a simple application, it might be responsible for all pages; in a complex application, there are multiple application controllers for different areas of the application.

By using a single, well-encapsulated object as the central point of reference for any persistent information, the application controller resolves the issues of information hiding and naming conventions. Rather than storing isolated pieces of information in the session, we store them in business objects and access them using messages from the application controller. Programming-language mechanisms let us track use of the application controller and business objects and more easily modify our code. If we are using a statically typed language, we also get static type checking as an additional validation of data usage.

Action. The single input controller will invoke one of many possible actions on each request. One of its responsibilities is to determine which one. This depends on both the input from the user and on the application's current state, so it must be determined in conjunction with the application controller. We represent the result of this determination as an Action object (an implementation of the Command pattern described elsewhere[6]).

Business objects. We implement business objects as normal objects that contain only business logic, so they should have no knowledge of any other layers. The application controller is the only thing that manipulates the business objects, and for this reason they are not shown in Figure 2 but are inside the application controller. Both of these attributes make it much easier to develop and test the business logic in isolation from the Web infrastructure. Because the business objects

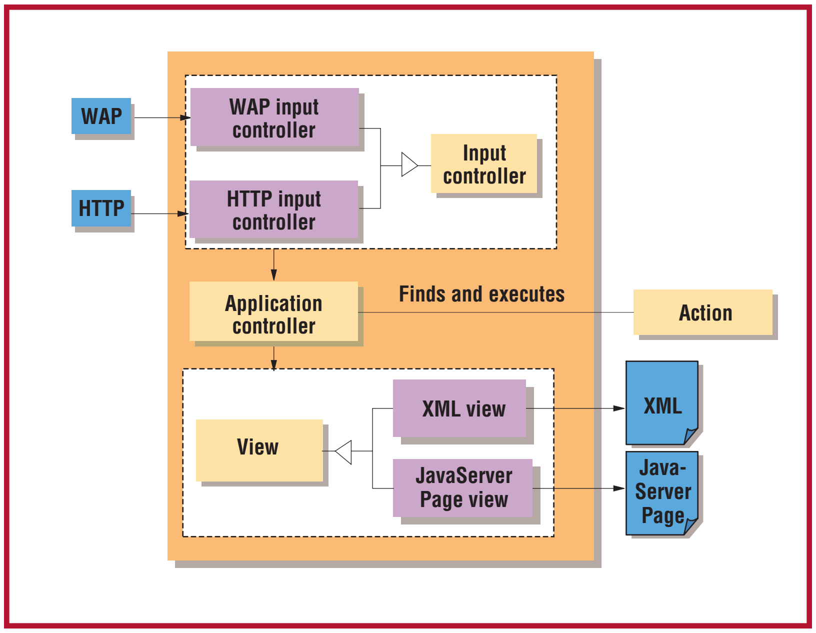

Figure 2. The Web Actions framework.

Figure 3. Java code for an input controller servlet.

```
public void service( HttpServletRequest req,
     HttpServletResponse res )
  throws ServletException, IOException {

  ApplicationController controller = this.appControllerFor(req);
  this.setValuesIn(controller, request);
  String actionID = req.getPathInfo();
  Action action = controller.actionFor(actionID);
  appController.performAction(action);
  View view = this.viewFor(req);
  view.forwardPage(controller.nextPage());
}
```

might be isolated, we should be able to use the same implementation for a thin-client Web application, a more rich-client implementation, or even a traditional GUI.

View. We implement views as server pages, which can access the application controller and business objects. Views should contain as little code as possible, delegating most functionality to the application controller or business objects. Only code directly related to presentation in the current page should be used in a page. If supported, we prefer to use a tag mechanism such as JavaServer Pages (JSP) custom tags to remove code from the pages altogether.

Figure 2 shows two different view mechanisms. The first uses a server page implementation, appropriate for a Web browser or WAP device. The second generates the same information in an XML format, which could be sent to an applet, an ActiveX control, or a GUI application.

Web actions: Control flow

By organizing the scripts, server pages, and regular objects as we've described, we've overcome many of the issues associated with simple Web development. We have minimized code duplication and reliance on the protocol's details or naming conventions by using a single input script. We have also achieved good information hiding by maintaining the data in objects rather than as flat session data. We have confined our use of server pages to the view layer, maximizing the amount of code we can manage and debug using standard programming tools and methods. By keeping each layer separate, we can test each in isolation. Overall, our application remains well-factored and easy to maintain and extend.

To see how this works in more detail, let's examine the flow of control from a single Web request in an online banking application. First, we see how the input controller accepts input, finds and sets up for the action, executes the appropriate application controller code, and then delegates to the view. Figure 3 shows sample code for this, and here we examine each of the steps.

Find the controller

The input controller must determine which application controller is responsible for the current request. Active application controllers are stored in the session. We assume that we can determine the controller using a lookup based on the request's path. For example, in our banking application, we might have an account maintenance application controller, which is used for any pages related to account maintenance.

Accept input

Once we have determined the application controller, we must extract the appropriate information from the request and transfer that information to the application controller. Most notably, we need to find any input parameters. These can be encoded as part of the URL, as query parameters listed after the URL, or as form data. Regardless of the transmission mechanism, this input consists entirely of strings and must be converted into the appropriate types and validated. The input controller extracts the information and, in cooperation with the application controller, performs basic syntactic validation and informs the application controller of the values.

For example, we might have a user request to transfer funds such as

```
  https://objectbank.com/
InputController/transfer?from=
123&to=321&amt=$50.00
```

The string `/transfer` identifies the action to the input controller. The remainder of the string holds query parameters for the two account numbers and the amount to transfer.

On the other hand, a request to update account holder data might be submitted from an HTML form and would carry many pieces of data as form data, invisible in the URL:

```
  https://objectbank.com/
InputController/updateAccountData
```

Find the action

The application controller can keep track of a set of acceptable sequences of operations and the previous steps the user has taken. On the basis of this and the information submitted in the request, we can determine which action to take and whether this action is legal in the current context. This logic is particularly important because Web users can use the Back button to throw off the sequence of operations in a way that is not possible in a GUI.

In the simplest version, we might define actions for login, logout, transfer between accounts, account update, and bill payments. Any account activity is allowed once the user has logged in, but actions transferring money have a confirmation page. We only let the user confirm a request if the immediately previous operation was the request. We also detect the situation in which the user backs up and resubmits the same request or hits the Submit button repeatedly while waiting for the first submission to be processed.

Perform the action

Once we have determined which action to take, we must execute it. The exact procedure for this varies depending on which implementation we choose. We might choose to have heavyweight actions, implemented as objects inheriting from a class Action and implementing a trigger method.

```
public void trigger(Controller
      controller){
   BankController ctrlr=
      BankController)controller;
   Account acct=ctrlr.readAccount
      (context.getAccountNo());
   ctrlr.setCurrentAccount(account);
   ctrlr.setNextPage("account
      Summary");
}
```

This lets us separate the different actions, but it does not give us as much encapsulation in the application controller as we might like. Rather than the standard OO usage of multiple methods on a single object, we have many action objects that manipulate the controller's state, providing little encapsulation.

As an alternative, we can represent the action simply as an indicator of the application controller method to be performed. This is particularly easy in a system such as Smalltalk, with simple reflection techniques. Given a Smalltalk servlet and JSP framework,[7] we can simply set the action

```
action := #deposit.
```

and then execute

```
controller perform: action.
```

In this case, the action is simply the action method's name, and we invoke it with the standard `perform:` method. No action class or trigger operation is necessary.

Which of these choices is preferred depends on the situation. Keeping code and data together in the application controller provides better encapsulation but raises potential team issues of having many people collaborating on a single object. In a GUI presentation, it might also be desirable to use the full Command pattern to better support Undo functionality.

Keeping code and data together in the application controller provides better encapsulation but raises potential team issues of having many people collaborating on a single object.

Forward the request

We use forwarding to divide responsibility among different components. Once the action is complete, we determine the URL of the next page to be displayed and forward the request. In a simple system, the actions can directly determine the next page. In a more complex system, the application controller might coordinate this using internal state management such as a state machine.

Variations

We have described one particular implementation of this type of framework. Many others are possible, and in our uses of this framework, we have made significant variations depending on the application's precise needs.

Complex views

We have described views as mapping directly to server pages, but in more complex applications, this can become more sophisticated. First, we can assemble larger views from multiple smaller views. So, we might have a main page with subsections in frames or with subsections determined by including the results of other Web requests. We could model this structure either by explicitly issuing new requests or by using an internal mechanism to forward the request to another partial page.

We might also want to interpose another layer of objects at the view level. For example, if handling multiple possible forms of output, we might need an explicit View object that handles the output details on a particular device. For example, we might store a key that lets the View identify the appropriate output. This might delegate to different server pages for HTML and WAP presentation and to a screen identifier for a GUI. Using a View object helps isolate the presentation of an action's results from the action itself. It is also an appropriate place to implement functionality such as internationalization of the result.

Action context

We have described an architecture with very lightweight actions in which state is associated directly with the application controller. In some situations, it might be useful to dissociate some of this state from the controller. In particular, if we do not have a simple mapping from the URL to the application controller, we might need to extract more of the request state to determine the correct controller. We can do this by introducing an ActionContext object, which stores the request state in a standardized form. In this case, we would create the context, use it to find the appropriate controller, and then apply actions to the combination of controller and context.

Business logic and components

We have not devoted a great deal of discussion to the business logic layer, because we consider it to be a normal OO program. However, it can be quite complex and involve any number of other technologies. One that frequently comes up is the relationship between this layer and components, particularly Enterprise JavaBeans. Some frameworks (when using Java) use entity EJBs as a standards-based solution for the business object layer.

We do not generally consider this to be a good solution. Although the business logic layer might have good reasons for accessing components—most notably session beans encapsulating access to transactional or legacy systems—using entity beans to represent the business logic seems ill-advised. Entity beans offer some automation potential for issues such as transactions, security, and persistence, but they impose serious limits on our design and have significant performance constraints, even relative to normal Java performance. In particular, the absence of inheritance is a burden, and other frameworks exist to deal with transactions, security, and performance in the context of normal business objects. Finally, it is harder to test EJBs in isolation because they require a container to run; this imposes increased overhead on development. Using session beans to wrap business objects or forgoing EJBs altogether are both reasonable alternatives.

Using our framework, developers primarily focus on writing application code rather than dealing with servlets, requests, or session variables. We have used this in a variety of different Web applications, using both Smalltalk and Java. In this, we have achieved better software quality because good OO principles were not dictated but followed naturally from the structure of the framework. Paradoxically, it also allowed developers inexperienced with OO techniques to produce high-quality code without knowing the details of the framework too well. These frameworks could be extended in several different areas, including customization to particular domains, conversion into a full-fledged framework rather than a set of patterns, and more detailed adaptation to particular technologies.

Acknowledgments

The authors thank Michael Ellis and Martin Fowler for their contributions to the ideas expressed here.

References

1. M. Ellis and N. Dai, "Best Practices for Developing Web Applications Using Java Servlets," OOPSLA 2000 tutorial; www.smalltakchronicles.net/papers/Practices.pdf.
2. N. Dai and A. Knight, "Objects versus the Web," OOPSLA 2001 tutorial; www.smalltalkchronicles.net/papers/objectsXweb10152001.pdf.
3. E. Gamma et al., *Design Patterns*, Addison-Wesley, Reading, Mass., 1994.
4. G.E. Krasner and S.T. Pope, "A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System," *J. Object-Oriented Programming*, vol. 1, no. 3, Aug. 1988, pp. 26–49.
5. M. Potel, *MVP: Model-Viewer-Presenter*, tech. report, IBM, 1996; www-106.ibm.com/developerworks/library/mvp.html.
6. A. Bower and B. MacGlashan, "Model-View-Presenter Framework," 2001, www.object-arts.com/EducationCentre/Overviews/ModelViewPresenter.htm.
7. *VisualWave Application Developer's Guide*, tech. report, Cincom Systems, 2001.

About the Authors

Alan Knight is a senior software developer at Cincom Systems, where he works on Smalltalk Web tools. He was also recently the technical lead on a high-performance Smalltalk Web application server supporting full implementations of ASP, JSP, and servlets. His research interests include Web development, object-relational mapping, and team programming systems. He received his BS and MS in computer science from Carleton University in Ottawa, Canada. He is a member of the ACM. He coauthored *Mastering ENVY/Developer* (Cambridge Univ. Press, 2001). Contact him at 594 Blanchard Cr., Ottawa, Ontario, Canada K1V 7B8; knight@acm.org.

Naci Dai is an independent mentor and an educator. He teaches object technology, Java, design patterns, and distributed computing. He leads and mentors Web development projects. He has a background in applied engineering and computational physics. He has received his PhD in mechanical engineering from Carleton University. He is a member of the ACM. Contact him at Acarkent C75, 81610 Beykoz, Istanbul, Turkey; nacidai@acm.org.