Patterns for Web Applications[*]

Michael Weiss
Carleton University, Ottawa, Canada
`weiss@scs.carleton.ca`

Introduction

A conceptual framework for "small" web applications

This paper contains work in progress on a pattern language for "small" web applications. Its goal is to outline a conceptual framework for developing web applications. The pattern language has reached a certain stage where I would like to get the feedback of the pattern community.

Why a conceptual framework? Many web applications may not need a comprehensive application server framework, only micro-frameworks for specific tasks such as template processing. The main application can be developed on top of a standard open-source platform such as LAMP (Linux, Apache, MySQL, and Perl/PHP/Python) [11].

There is a trade-off between the complexity of a comprehensive framework, and the needs of your application. Frameworks often have a large feature set, and thus a steep learning curve, and can be difficult to deploy. For example, while some web applications may warrant a content management system, a template processor may be all that is required for most.

A conceptual framework, on the other hand, gives you a model for how to build a custom application server, and and tells you when to use task-specific micro-frameworks. The conceptual framework introduced in this paper is documented as a pattern language. These patterns are geared towards non-trivial, small to medium sized web applications. Our goal is to document common practices for typical design issues encountered.

[*]Copyright © 2003, Michael Weiss. Permission is granted to copy for PLoP 2003.

Common design issues

These design issues include the flow of interaction of the application with the user, processing user requests, and saving the state of the application between invocations. Other common problems covered are how to generate output for different types of browsers, interact with external data sources (including web sites), and make applications more interactive.

Web site and navigation design, on the other hand, are outside the scope of this pattern language. For example, the patterns documented by Welie [22], van Duyne et al [20], Lyardet and Rossi [17], Rossi et al [18], and Fernandez et al [12] address this area. Following Welie, these patterns can be classified into site, experience, task, and basic interaction patterns.

An example of a site pattern is a COMMERCE SITE, and an example of a task pattern is BUY/SELL. One of the patterns to support the buy/sell task is SHOPPING CART. Managing a shopping cart is an special instance of another task pattern, FAVORITES, which, in turn, is a special case of the basic interaction pattern LIST BUILDER. Another basic pattern is WIZARD, which suggests to step the user through the checkout process [22].

Overall context of this pattern language

The overall context for these patterns is that the web site and navigation design have been completed. Suppose we are building an e-commerce site, and have chosen to structure the basic user interaction around the SHOPPING CART and WIZARD patterns, how do we proceed with the design at the programming level? Scoping the pattern language in this way just reflects the typical division between web designers and programmers.

Synopsis of the Pattern Language

An annotated roadmap summarizes the rationale for using each pattern

This section summarizes the pattern language in an annotated roadmap (Figure 1). Arrows indicate how patterns refine or complement each other, and annotations on the arrows summarize the rationale for choosing to apply a particular pattern in the context of the pattern or patterns that precede it. The rationale is given as a major force that the pattern helps resolve.

Maintainability is a driving force for many of the patterns: TRANSITION TABLE (5), CENTRAL DISPATCHER (10), SEPARATE CONTENT FROM PRESENTATION (19), DATA SOURCE ADAPTER (33), and CHAIN OF APPLICATIONS (33). Other patterns are motivated by security, and pri-

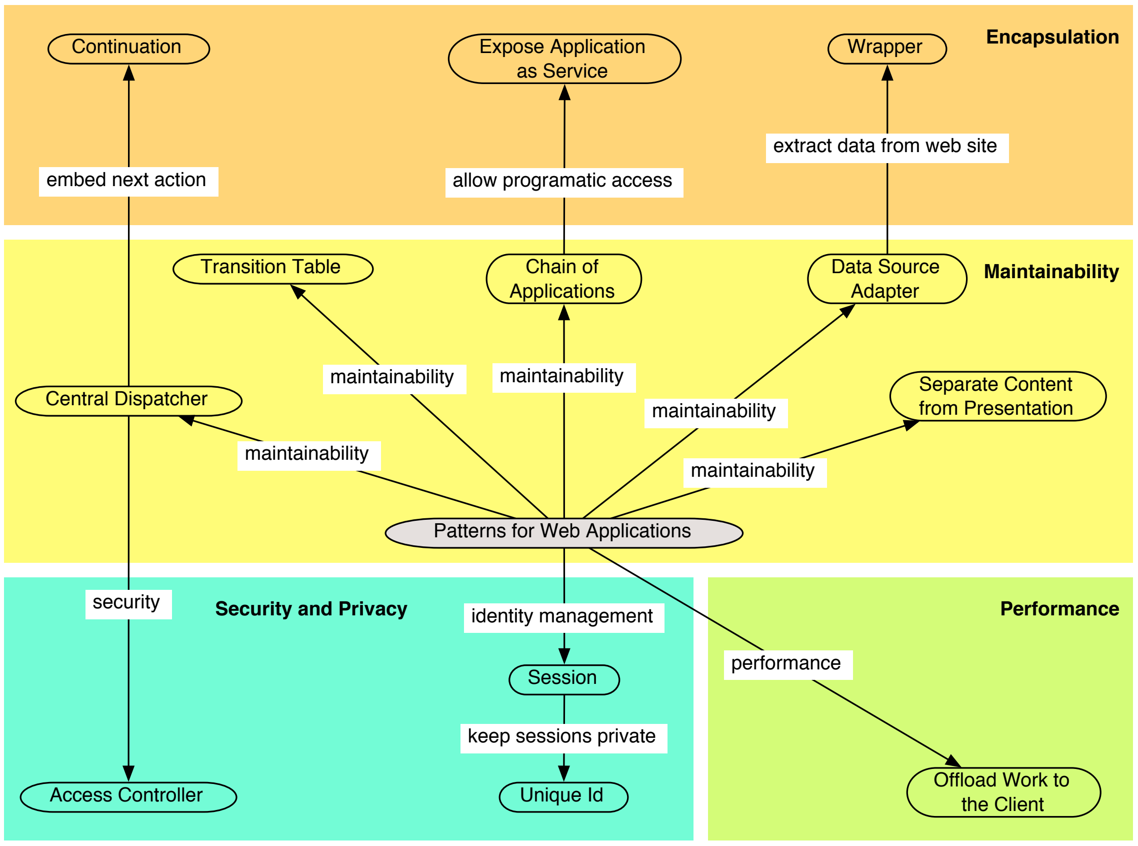


Figure 1: Roadmap to the patterns in this language

vacy concerns: ACCESS CONTROLLER (33), SESSION (24), and UNIQUE ID (29). Still other patterns focus on encapsulation: CONTINUATION (14), WRAPPER (33), and EXPOSE APPLICATION AS SERVICE (33). Finally, performance is addressed by OFFLOAD WORK TO CLIENT (30).

Suggested order of applying the patterns

Here is the suggested order of applying these patterns. You begin the design of your web application by defining the flow of interaction with the user in a TRANSITION TABLE (5). The flow of interaction is expressed in terms of the pages presented to the user, and the actions the user can perform on them. You then define how the application responds to user actions within a CENTRAL DISPATCHER (10). If you need to support multiple output formats, use SEPARATE CONTENT FROM PRESENTATION (19).

Most web applications will need to track user profiles, and session information — SESSION (24). DATA SOURCE ADAPTER (33) describes how your application can interact with external data sources (databases, and web sites). While advanced security techniques are outside the scope of this paper, basic protection of web site assets, as well as user privacy can be achieved using ACCESS CONTROLLER (33), and UNIQUE ID (29).

Eventually, you will find it convenient to separate your web application into multiple subapplications that can be composed to form new applications. CHAIN OF APPLICATIONS (33) describes the general approach, and EXPOSE APPLICATION AS SERVICE (33) describes a particular instance using web services. For reasons of performance and responsiveness, you may also be wanting to OFFLOAD WORK TO CLIENT (30).

A running example illustrates the patterns: BookFinder.com

We will illustrate these patterns with a running code example, an application for comparing the prices of books at online stores. This example will be gradually introduced with each pattern. We also show excerpts of the implemention in Perl, but try to stay away from its idiosyncracies which would make porting to another environment (PHP, Java) difficult.

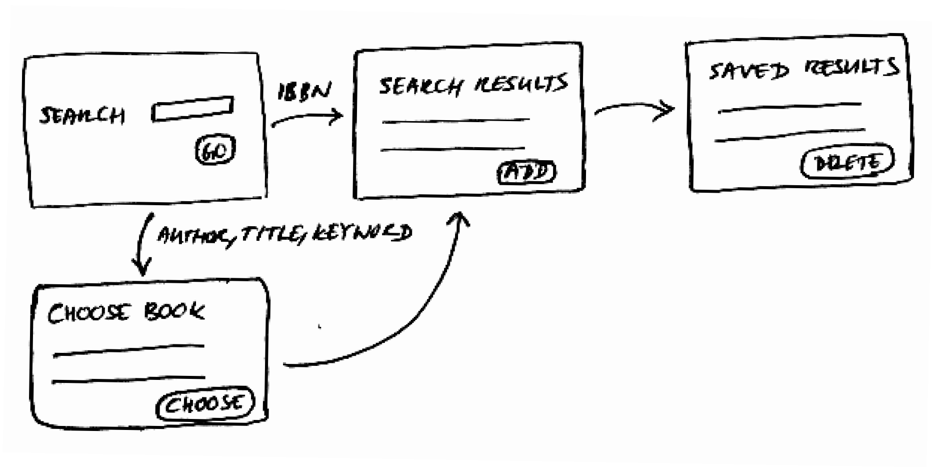
A final note on the format used to document the patterns. The patterns are described using the Alexandrian form [1]. That is, for each pattern we document its context, problem, problem details, solution, and resulting context. Problem and solution are highlighted by emphasis. For each pattern, a diagram summarizes the solution. The diagrams use the Use Case Map notation to indicate both structure and behavior of a solution [4].

Because the pattern language is quite long, it will be published in several parts. The following patterns will be presented in this paper: TRANSITION TABLE (5), CENTRAL DISPATCHER (10), CONTINUATION (14), SEPARATE CONTENT FROM PRESENTATION (19), SESSION (24), UNIQUE ID (29), and OFFLOAD WORK TO CLIENT (30). Thumbnails for the remaining patterns are provided in an appendix.

Transition Table

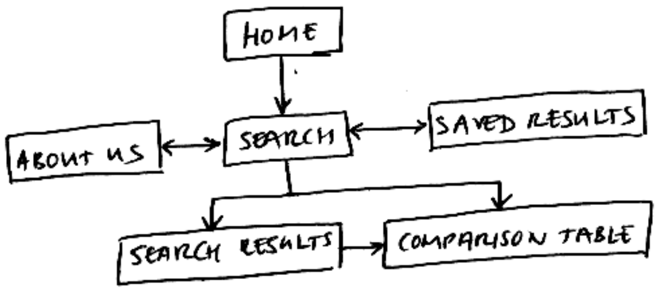
You are designing a web application. The navigation design of your web site is described by storyboards, a site map, and some wireframes for individual pages. Storyboards document likely interaction scenarios, and give a general sense of how the application will work [23]. However, they are only rough sketches of how users will interact with the site.

Storyboards document likely interaction scenarios



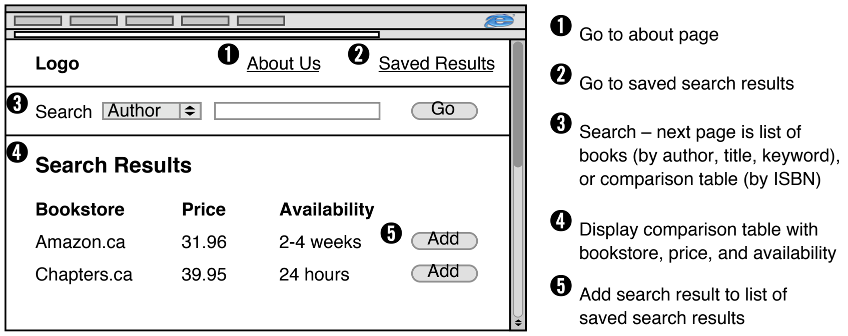
A site map documents the organization of your site, and may indicate other important information such as pages that are dynamically generated, and areas of the site that are access protected [23]. However, a site map does not indicate every single link between pages. Links hat break the hierarchy of the site map (such as shortcuts) are typically excluded.

Site map shows the major pages and their links



Wireframes document important page elements such as search boxes, and the layout of specific pages [23]. They may also indicate links between pages at a level of detail not shown in the site map, such as links to related pages, or backlinks. For instance, this wireframe shows details of the search result page for a price comparison site.

Wireframes provide details on invididual pages



Bookstore	Price	Availability
Amazon.ca	31.96	2-4 weeks
Chapters.ca	39.95	24 hours

❶ Go to about page

❷ Go to saved search results

❸ Search – next page is list of books (by author, title, keyword), or comparison table (by ISBN)

❹ Display comparison table with bookstore, price, and availability

❺ Add search result to list of saved search results

***

You need to specify the detailed navigation logic without getting lost in too many links, and without forgetting important navigation paths.

A smooth flow of interaction creates a positive user experience. Before you design the details of your application logic you need to plan in what sequence the pages of your application should follow each other.

While a site map documents the organization of your site, it only shows the key relationships that exist between its pages. A fully connected site map would not meet the goal of documenting the *typical* navigation paths through a site. When making the transition to the design of your web application, you need to make the navigation logic unambiguous.

A site map only shows key links between pages

Example: Consider the design of a site for comparing the prices of books at different online stores. The user should be able to search for a book by author, title, keyword, or ISBN. The result of a search should be a table with the price for the book at each store and its availability. The user should also be able to save the results of different searches on the site. Key

BookFinder.com

functions (about us, search, and saved results) should be accessible from every page. The navigation design produces the site map above.

This site map is still ambiguous. It does not contain two important transitions: one from Comparison Table to Saved Results for adding a search result, and another from Saved Results to itself for deleting a search result. Note that these have not been forgotten, but were not included because they would break the hierarchy. Such links *can* be indicated in a wireframe as shown in the example above, but are typically not shown in a site map.

Unspecified navigation details may lead to inconsistencies in the implementation

Resolving ambiguities in the site map up front is essential to avoid inconsistencies in the implementation of the navigation logic. Detailed design decisions on navigation should not be left to the code that handles user requests, because the same part of a sitemap may be interpreted differently by different developers, as it is not fully specified.

It can be difficult to extend the code without a specification of the detailed navigation logic

Code and specification must be properly synchronized

Without a specification of the detailed navigation logic, the only documentation of how to get from the site map to the implemented navigation logic would be in the code. Anyone implementing an extension would have to find out about navigation design decisions from the code itself, potentially a very time consuming and error prone process. On the other hand, well-structured code can be more accurate than a specification, if the specification is not kept in synch with the actual implementation

This pattern suggests to define the acceptable succession of pages in a set of transition rules. Each page corresponds to a user action. The left side of each transition rule is the name of the user action. The right side contains the set of user actions that can legally be performed next.

```
<action> => <next action 1> | <next action 2> | ...
```

BookFinder.com

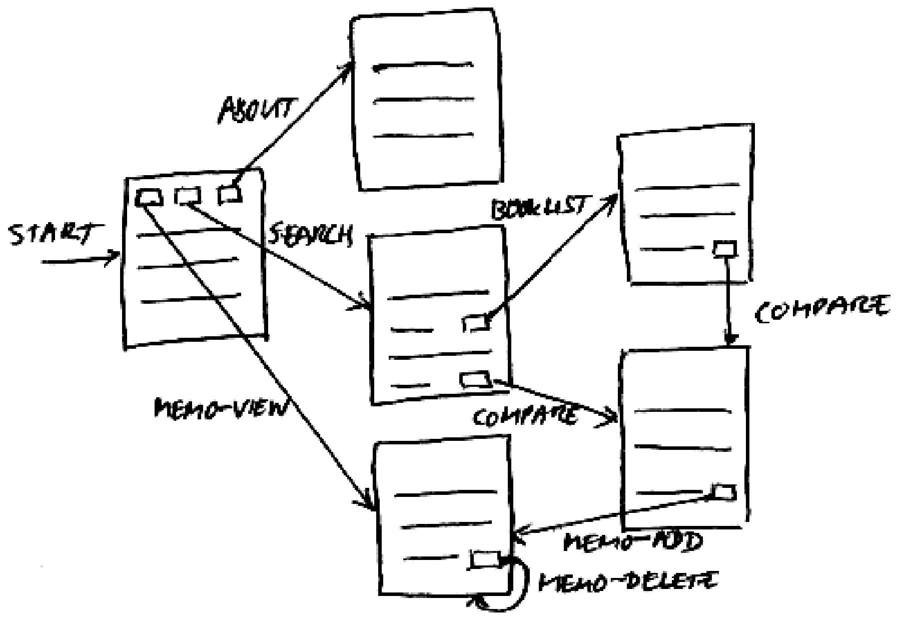
Example: The following transition rules provide a detailed navigation design for the price comparison site. From the `start` page, the user can perform these actions next: `about`, `search`, and `memo-view`. The `about` action causes information about the application to be displayed. The `search` action will result in a list of matching books (`booklist`) — from which the user has to select a book — for a query by anything else than the ISBN, and a comparison table (`compare`) otherwise. The `memo-add` and `memo-delete` actions update the list of saved results.

```
start    => menu
search   => booklist? | compare? | menu
booklist => compare? | menu
compare  => memo-add! | menu
memo     => memo-delete! | menu
about    => menu

menu == about? | search? | memo-view?
```

We also indicate whether an an action has side effects. Side-effect free query actions are decorated with a `?`, and update actions with a `!`.

Therefore:

Define rules for the legal transitions between pages. Each page corresponds to a user action, and a transition rule defines what actions can be performed next.



***

Defining the transition rules up front ensures that the details of the navigation logic left open by the information architecture are not up to interpretation later in the design. This helps ensure that the navigation logic is implemented consistently. Another benefit of this solution is that the transition rules can also be used to derive test cases for the web application.

Consistent navigation logic enhances maintainability

Transition rules can be used to test the application

Transition rules provide a point of reference for maintaining and extending the code, since the navigation logic is no longer only documented in the code. However, this requires that changes to the navigation logic during implementation must be reflected in the specification. A good way of achieving this is to use the transition rules for testing, as suggested.

Separating the specification from the code makes the application easier to extend

Also note that the application still needs to implement proper error handling for out-of-order requests. Requests can be received in any order, for example, if the request is submitted out of context from a bookmark, using the browser's back button, or due to caching.

However, this does not ensure requests are made in the specified order

The handling of user actions should be coded into the logic of a CENTRAL DISPATCHER (10). The next actions should be encapsulated using CONTINUATION (14). This pattern was inspired by the Tennis Shoppe case study in [14], and its discussion of design issues in session management.

Central Dispatcher

You have defined the detailed navigation logic in a TRANSITION TABLE (5). You now need to define how the application handles user requests. Some pages may also require special pre- and postprocessing (such as logging, authentication, and output transformation).

***

Unstructured evolution of your application, as code to handle new user actions is added, often results in application logic that is hard to understand and extend. Specifically, it is a challenge to ensure that pre- and postprocessing is done in a consistent manner.

Separate scripts for each type of request can result in duplicated common code and inconsistent navigation

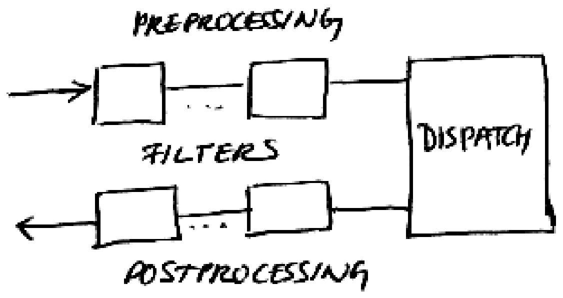
One way of handling user requests is to write separate scripts for each type of request, and rely on the web server to dispatch the request to the appropriate script based on script names. While this solution is simple to implement, it can result in duplicated code for common functions (such as logging, authentication, and output transformation), and inconsistent implementation of the navigation logic. The reason is that the scripts implement both the navigation *and* the presentation logic.

Centralized request handling can enhance maintainability at the cost of added complexity

Another option is to centralize request handling, and to separate navigation from presentation. In this approach, requests are processed by a single script that dispatches the request to the appropriate request handler. The request handlers can be implemented in the same script as the dispatcher, or in separate scripts, however, ones that are "invoked" by the dispatcher through redirection via the web server. A variation of same-script implementation is to factor out the code for request handlers into command objects (which can be implemented as separate packages/files in Perl).

Common pre- and postprocesing also makes the application more robust against changes

In the centralized approach, the "entry" script acts as a central dispatcher. Besides dispatching the request appropriately, it is also a place for uniform pre- and postprocessing of the requests. Authentication is an example of preprocessing. Here, a central dispatcher can go beyond the simple capabilities offered by a web server. For instance, web servers provide a simple form of authentication to restrict access to specific areas of your site, however, the user interface is rather primitive (a popup dialog), and logins and passwords are submitted in cleartext. There is also no way of remembering user passwords beyond a single client session.



The basic idea is to test for the type of action in a user request, and then branch off to the appropriate part of the application. However, if the handling code is intermingled with the dispatching code, this can result in convoluted application logic. There must be a clear separation between the part of the application that determines the action to handle, and the handling of the request. The solution is to separate the dispatching block from the request handlers in the structure of your code.

BookFinder.com

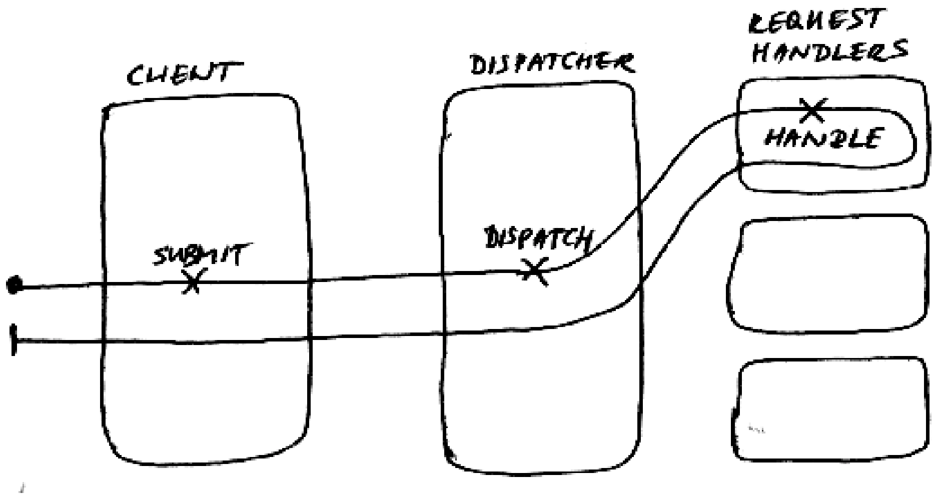
Example: Assuming that the requested user action is passed via a form parameter (`action`), the following code excerpt will dispatch the action to the correct request handler for our price comparison site. Note that this example also illustrates a simple case of preprocessing (logging).

```
my $q = new CGI;
my $action = $q->param("action") || "start";

# pre-processing
log($q);

# dispatching block
if ($action eq "start") {
  start($q, ...);
} elsif ($action eq "about") {
  about($q, ...);
} elsif ...
  .
  .
} else {
  error("don't understand");
}
```

All request handlers are passed a reference to the invocation context `$q`. Other data (eg a session id) may have to be passed to the handlers as well, as we combine this pattern with other patterns. Invoking `log()` is only a simple example of centralized pre- and postprocessing. More complex examples are discussed in ACCESS CONTROLLER (33), and SEPARATE CONTENT FROM PRESENTATION (19), respectively.

Therefore:

Centralize the processing of user requests. Handle all user requests in a single "entry" script that dispatches them to the appropriate handler. Locate code for pre- and postprocessing (such as logging, authentication, and output transformation) in this central dispatcher.



***

Using a central dispatcher is more complex than using individual scripts for each user action. While the individual script approach is more familiar to developers, there are certain advantages to be gained from using a central dispatcher. Centralizing request handling makes an application with more complex navigation logic easier to maintain. It cleanly decouples navigation from presentation. The central dispatcher receives requests, and dispatches them to the corresponding request handler.

Applications with more complex navigation logic are easier to maintain using a centralized dispatcher

Locating code for pre- and postprocessing in the central dispatcher ensures consistency of common behavior across the different parts of your application. For instance, when used in combination with ACCESS CONTROLLER (33), this pattern enforces that all requests to access-restricted areas of your site are checked for their admissability in a *single* location.

Centralizing common code ensures consistency

This pattern is most often used in combination with CONTINUATION (14), and SEPARATE CONTENT FROM PRESENTATION (19). The request handler creates its reply page using CONTINUATION (14) to embed context information required to process each user action that can be performed next. It is also responsible for rendering the page. SEPARATE CONTENT FROM PRESENTATION (19) helps with generating the output in the required format, in particular, if you need to support multiple types of browsers.

This pattern is widely used in the design of web applications. Examples of its use are documented in the Tennis Shoppe case study in [14], and in the design of the original Wiki [9]. The FRONT CONTROLLER (344) pattern in [13] has the same intent. A single object handles all user requests, and delegates them to the appropriate handler. Common code for logging, authentication, or output transformation can be added through filters.

Continuation

As suggested by CENTRAL DISPATCHER (10), you use request handlers to generate the next page in response to a user action.

***

Context information for processing a user request (an action, plus any required parameters) needs to be passed from one page to the next.

When a request handler in the CENTRAL DISPATCHER (10) generates a page in response to a user action, this page needs to contain all the information required to perform the next action (or set of actions) identified in the TRANSITION TABLE (5) for this action. The reason for this is that the HTTP protocol underlying the communication between web clients and the application is stateless. It does not preserve the application context between requests. This is something you need to do yourself.

Design for simplicity

In designing a solution for this problem you need to consider a number of trade-offs. On the one hand you want the solution to be simple. Often you can assume that for the duration of a short transaction (for example, the registration of a new user) the pages of your application will be accessed consecutively, that is, it is in synch with the designed navigation flow. Such a solution does not require cookies, which the user may have disabled, and will therefore work with all browser configurations. You also want to encapsulate the context so that adding code to embed the context for a new action does not affect the code for existing actions.

Goal is to encapsulate the context of user actions

Restricted to a series of consecutive pages

On the other hand, relying on receiving a series of consecutive page requests restricts the use of CONTINUATION (14) to short interaction sequences, and is brittle against any interruptions of the navigation flow, for example, caused by the user hitting the back button. Context information that is passed in unencrypted form can be read by a client, and thus be manipulated. (Allowing the user to read the data is a potential privacy concern, while allowing them to manipulate the data raises security concerns.)

Sending context data unencrypted raises privacy and security concerns

This pattern suggests to encode each next action and its parameters within form parameters. Form parameters are generally associated with a

`<form>` tag, but it does not have to be that way. For example, they can be embedded in an URL. Within a form, they can also be stored in hidden fields, or embedded within the `action` attribute of the form.

Encode the context of an action in URL parameters

This suggests three approaches to encoding context information, each with their advantages and disadvantages. First, the context can be encoded in the parameters of an URL. Specifically, fixed parameters (such as the language to be used) are often embedded into a page in this way. The following URL encodes the context for the `about` action:

```
<a href="bookfinder.cgi?action=about">About Us</a>
```

This is a simple way of encoding a static context in a page. (At least, if you are creating an HTML page; WML pages can also contain variables, which you can use to embed dynamic context information in an URL.) However, you should avoid creating complex URLs in this manner, as they tend to be hard to read. Also, this approach is restricted to query actions.

Encode the context for each action in a separate form

Second, if more than one action can be taken, the context for each action can be encoded in a separate form within the same page. The form below contains the context for processing a `memo-add` request. The action is embedded into the form `action` attribute. The action parameters (`isbn`, `bookstore`, etc.) are stored in hidden fields.

```
<form action="bookfinder.cgi?action=memo-add" method="POST">
  <input type="hidden" name="isbn" value="0802117244">
  <input type="hidden" name="bookstore" value="Chapters.ca">
  <input type="hidden" name="price" value="39.95">
  <input type="hidden" name="availability" value="24 hours">
  <input type="submit" value="Save">
</form>
```

Using hidden fields is far more readable than embedding all form parameters into an URL. Some pieces of context information (for example, the action name), however, can still be URL-encoded within the `action` attribute of the form. This will more clearly indicate which context parameters have static values, and which values are dynamically generated by the application. Since "hidden" fields are, of course, quite visible to web clients, fields with sensitive information should also be encrypted to ensure that their content cannot be read, or manipulated.

Encode the context for each action using decorated input fields when using one form

Third, when using a single form, the different actions and their associated data can be distinguished by decorating the input field names with unique identifiers. The following form for the `memo-delete` action illustrates the approach. Each search result on the Saved Results page is given a unique name, reflected in the name of the corresponding submit button.

```
<form action="bookfinder.cgi" method="POST">
  <table border="0">
    <tr> ...
      <td><input type="submit" name="memo-delete.0802117244"/></td>
    </tr>
    <tr> ...
      <td><input type="submit" name="memo-delete.0393041530"/></td>
    </tr>
  </table>
</form>
```

This approach is often used when there is a single action (in this case, `memo-delete`) that can be performed on a list of items. It will result in a more compact representation of the page, since the preamble of the form (action and method) is not repeated multiple times. On the other hand, decorating input fields (including submit buttons) with unique identifiers is more complex than using multipe forms with a parameter that identifies the item each form applies to. If you want to ensure that the item id cannot be manipulated, it should also be encrypted as suggested earlier.

BookFinder.com

Example: In response to the `compare` action, the price comparison site extracts price and availability information for a given book from the sites of online bookstores. From this it generates a price comparison table that can be represented as a nested associative array like the following:

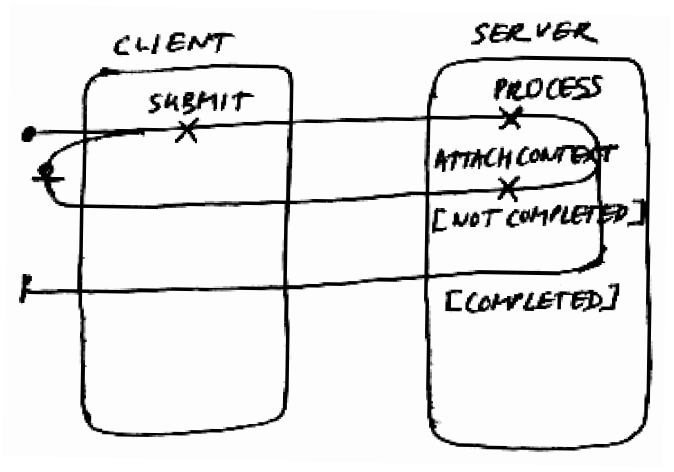
```
my $searchResult = {
  title => "Lucky Pierre",
  authors => [
    "Robert Coover"
  ],
  isbn => "0802117244",
  quotes => [
    {
      bookstore => "Amazon.ca",
      price => "31.96",
      availability => "2-4 weeks"
    },
    {
      bookstore => "Chapters.ca",
      price => "39.95",
      availability => "24 hours"
    }
  ]
};
```

This code excerpt shows how the request handler generates a separate form for each context in which the `memo-add` action can be invoked:

```
sub compare {
  ...
  foreach $quote (@{$searchResult->{quotes}}) {
  print <<END_OF_HTML;
<form method="POST" action="bookfinder.cgi?action=memo-add">
  <input type="hidden" name="isbn" value="$searchResult->{isbn}">
  <input type="hidden" name="bookstore" value="$quote->{bookstore}">
  <input type="hidden" name="price" value="$quote->{price}">
  <input type="hidden" name="availability" value="$quote->{availability}">
  <input type="submit" value="Save">
</form>
END_OF_HTML
  }
  ...
}
```

Therefore:

Pass context information for processing a user request by encoding each next action and its parameters within form parameters. These can be stored in either URLs, hidden fields, or submit buttons.



***

Encapsulating context data makes it easier to pass

Due to the way it encapsulates context, a solution in which the whole context of an action is passed within the returned page is easier to understand than one where session information is managed at the server side. On the other hand, such a solution may violate security and privacy concerns, since the context information could be passed in cleartext, and the application cannot be sure that the information has not been manipulated.

Passing context may violate security and privacy

Makes applications brittle against interruptions

For longer transactions, relying only on CONTINUATION (14) can also make an application brittle against interruptions of the flow. In this case, use SESSION (24) to store context information on the server. Only session ids need to be encoded using CONTINUATION (14), or, alternatively, client-side cookies. Often both forms of encoding session ids are used together, as we cannot always assume that clients have cookies enabled.

This approach is used by the Tennis Shoppe case study in [14]. The ASYNCHRONOUS COMPLETION TOKEN (65) pattern [6] is similar in intent. An asynchronous completion token is sent with a client request, and identifies how the response from the server should be processed. Similarly, a continuation provides context information for processing the next user action, but as a reminder for the server, instead of the client.

Separate Content From Presentation

The request handlers in CENTRAL DISPATCHER (10) are responsible for rendering the returned pages. The task is made more difficult since you need to support multiple types of web clients with very different presentation requirements (eg the screen resolution of a high-end LCD display vs that of a Internet TV), or accommodate frequent changes to the presentation that are independent of the functionality of your site.

***

By hardwiring the output of your application in the application logic, you will be faced with rewriting large parts of the application when changing its appearance, or duplicating code when adding support for new types of web clients. You need to reduce the effort required to accomodate changes to the target output format.

Support for multiple clients is often an afterthought

Separating the user interface from your application logic is accepted wisdom for conventional applications, but when it comes to writing web applications, it seems that we need to learn the drill all over again. There appear to be several reasons for this. One is that the initial version of a web application is often written for a specific browser on a single platform (eg Internet Explorer 5 on Windows 2000). Support for other types of web clients is thus an afterthought, and usually results in code cloat because of a duplication of application code, once for each output format.

Application and presentation logic are often intertwined

Another reason is an unclear division of roles between web designers and application programmers. It should be possible to change the presentation of a web site independently from its functionality. However, in practice, application and presentation logic are often intertwined. Changes to the presentation may affect many parts of the application logic.

Cascading Stylesheets

The separation between application and presentation logic can be accomplished in different ways. Probably the simplest approach is to use Cascading Stylesheets (CSS). Certain presentation decsisions such as the fonts to use with particular HTML tags, text color, or margin settings, can be separated from the page content by defining stylesheet rules. The content can now be marked up in a formatting-free subset of HTML.

However, stylesheet rules are limited to defining how your content should be formatted, and cannot be used to reorganize it, filter it against certain criteria (eg show only quotes below a certain price, and have them sorted in ascending order), or to map it into another type of markup.

The next two approaches can deal with these limitations of CSS. Both templates and XSLT stylesheets involve the definition of an intermediate representation for the content of a web page, from which the desired output format can be generated. Templates are written in the target output format, and contain placeholders where values from the application should be inserted by the template processor. Changes to a template can be made without touching the code that populates the template with application data.

Templates

However, templates are still tightly coupled to the application code, and the scripting language. Passing values between the application and the template processor is often limited to key-value pairs. Nested data structures can only be expressed in a manner specific to the scripting language. For example, HTML::Template supports hashes, arrays of hashes, etc.

XSLT

A more general approach is for the application logic to produce its results as XML, and to use XSLT stylesheets to translate the data to the desired output format. XSLT stylesheets can be processed on both server and client. Server-side stylesheets are often preferable, as they allow more powerful transformations without revealing the underlying data to the client. When using client-side stylesheets, the data is sent to the client.

XML as an interface between presentation and application logic

By encoding content as XML, we use XML as an interface between presentation and application logic. The XML representation preserves the meaning of the content in the markup. The tags used to mark up the content reflect the names of underlying business objects and properties, database tables and columns. The presentation logic now resides in stylesheets, and can be changed without touching the application logic.

However, while XSLT stylesheets provide a more general way of mapping from the content to the target output format, processing stylesheets is often more time-consuming than using templates. However, as XSLT processing technology becomes more efficient, XSLT processors may well soon be comparable in performance to template processors [10]. On the other hand, an XML representation of the content has the additional advantage that it is independent of the scripting language, and can be exchanged with other applications — see CHAIN OF APPLICATIONS (33).

BookFinder.com

Example: The internal representation of the price comparison table introduced in the discussion of the CONTINUATION (14) pattern can be mapped to XML in a straightforward manner. The hierarchical structure of the nested associative array translates one-to-one into XML.

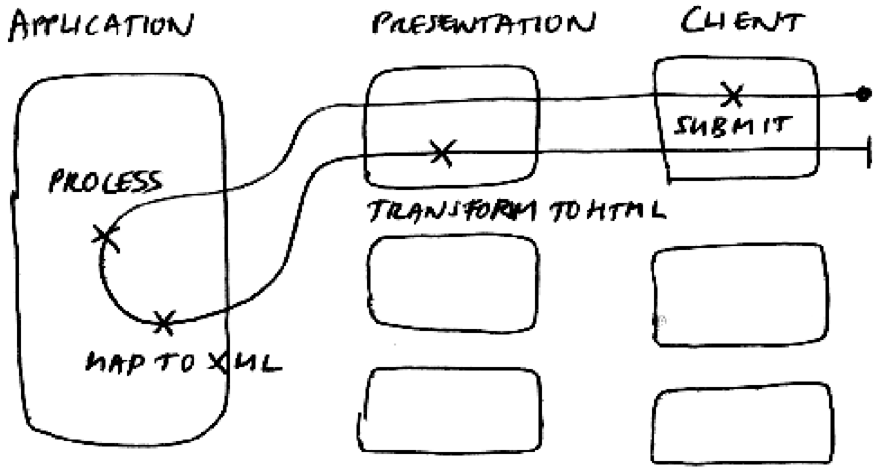
```
<searchResult>
  <title>The Adventure of Lucky Pierre</title>
  <authors>
    <author>Robert Coover</author>
  </authors>
  <isbn>0802117244</isbn>
  <quotes>
    <quote>
      <bookstore>Amazon.ca</bookstore>
      <price>31.96</price>
      <availability>2-4 weeks</availability>
    </quote>
    <quote>
      <bookstore>Chapters.ca</bookstore>
      <price>39.95</price>
      <availability>24 hours</availability>
    </quote>
  </quotes>
</searchResult>
```

An XSLT stylesheet consists of templates that match against different parts of the input document. For example, the following template defines that a quote representing the price and availabilty for a specific store should be shown as a row in a table. This stylesheet template generates the same output that was hardcoded in the discussion of CONTINUATION (14).

```
<xsl:template match="quote">
  <tr>
    <td><xsl:value-of select="store"/></td>
    <td>$<xsl:value-of select="price"/></td>
    <td><xsl:value-of select="availability"/></td>
    <td>
      <form action="bookfinder.cgi?action=memo-add" method="POST">
        <input type="hidden" name="isbn" value="{../../isbn}"/>
        <input type="hidden" name="bookstore" value="{store}"/>
        <input type="hidden" name="price" value="{price}"/>
        <input type="hidden" name="availability" value="{availability}"/>
        <input type="submit" value="Save"/>
      </form>
    </td>
  </tr>
</xsl:template>
```

Therefore:

Define an intermediate content representation as an interface between presentation and application logic of your application. Use CSS, templates, or XSLT to convert the content to the target output format.



***

This pattern proposes three solutions to separating application and presentation logic, each with their own benefits and liabilities. In summary, CSS supports a separation of content and formatting, but not changes to the structure of the content. In particular, it is impossible, using CSS, to map content from one markup format to another (eg from XML to WML). Templates and XSLT support the generation of *any* target output format.

The XML representation of the content can be mapped to the output format required by the browser, even if it is structurally different, for example, to WML for use on a mobile browsers, or to VoiceXML for use on a voice browser. Although the specifics of mapping to these markup languages are different, the basic approach is the same as generating HTML output. The main difference, not addressed here, is that mobile and voice browsers each require other dialog models than desktop web browsers.

Defining an intermediate representation adds some initial overhead. Choosing an internal data structure to be used with a template processor makes the solution specific to the scripting language and the template processor. A representation based on XML, on the other hand, is independent of the scripting language, and can be processed by either CSS or XSLT.

Using an intermediate representation also adds some performance overhead, which needs to be traded off for the flexibility when adding support for new web clients. Between templates and XSLT, templates are considered to be faster to process. However, more efficient implementations of XSLT processors have closed the performance gap to template processors.

The use of this pattern is documented in [7] as the basis of a portal architecture, and in [5] as a strategy for supporting multiple devices in web applications. Its application to decoupling the roles of application developers and web designers is extensively explored in [21]. Content management frameworks such as Cocoon [2] and AxKit [3] directly support the pattern.

Session

You need to maintain the state of the web application between user requests. However, passing context information between pages, as suggested by CONTINUATION (14), is not always adequate. The user may have arrived at a page by using the back button, or a bookmark so that their next action is no longer in synch with the designed navigation flow.

***

Passing context information between server and client for each request raises a number of issues. The context can be lost when the designed navigation flow is interrupted. The ability of clients to read the context data may also pose a considerable privacy and security risk.

Passing the full context for each request is not robust, if navigation is interrupted

A design that maintains application state by passing context information between server and client for each request is not robust against interruptions of the navigation flow. This can be as simple as users switching to another web site while they are using your application, ending their browser session, or as a result of using the back button. In each case, the context information required to process the next user action is lost.

For example, users often don't immediately proceed to the checkout when ordering products online. They expect the contents of the shopping cart to be stored for a certain duration even though they may interrupt their shopping process temporarily to perform other tasks. (Amazon.com advertise that they keep the content of a shopping cart for 90 days.)

Need to persist the context across browser sessions

You may also want to collect information about a user that persists across multiple uses of the same application. This can be information that you gather implicitly from the user's interaction with the application (eg a history of recently viewed products), or data entered by the user to customize the application (eg a list of stock symbols to monitor).

Contexts with different lifespans

As these examples show, the state of the application is something that we will want to maintain over time intervals of different length. Some information only needs to be kept for the short term, such as a user name entered into a registration form. Such information does not need to persist across browser sessions, and only needs to be passed between contiguous forms. This can be achieved by using CONTINUATION (14).

Some information we want to keep for the medium term (eg the contents of a shopping cart), and some for the long term (eg a user profile). Medium term coincides with a single, possibly interrupted, transaction (the contents of a shopping cart are no longer relevant, once the user has completed the checkout stage), while long term implies that the information should be accumulated over repeated interactions with the application.

Need for cookies

Medium or long term application state that persists across browser sessions can only be stored using client-side cookies. If the state information is small (eg the last time the user accessed the application), it can be stored directly in the cookie. In the general case, however, only an identifier uniquely associated with the user session (session id) should be stored in the cookie. The actual application state (session information) should be stored at the server, either in a database or a file.

Storing context information thus involves a number of trade-offs. Passing the full context with each request and reply simplifies the design, but may violate privacy and security concerns. Sensitive information such as passwords, or credit card numbers should not be passed in the context. It is visible to the user. Unauthorized users may also gain access to the information due to browser caching. Finally, somebody monitoring the network may capture the HTTP requests and replies, and extract the information.

You also need to be concerned about users manipulating the context. A well-known example is that of a user who submits a forged request to add more products to a shopping cart *after* having paid. Also, interruptions of the navigation flow cause the context to be lost. While in some cases this is of no concern (you don't really need to remember which article a user last read on a news site), in others it does matter (you do need to remember the contents of a user's shopping cart beyond browser sessions).

Split context into session id and session data

This pattern suggests to split the context into a session id to be shared with the client, and session data to be kept on the server. The session information is stored in a database, using the session id as a database key. When a user first accesses the application, no session id is sent with the request.

The server therefore creates a new session id, and a corresponding record in the database. On subsequent requests, the client includes the session id, and the server retrieves the session information from the database.

Embed the session id in an URL or a hidden field

There are different approaches for storing the session id, and passing it to the server. One approach is to embed the session id in an URL. The approach requires that the session id is added to all URLs in the content of a reply. While some web servers already provide rewrite engines to automate this task, their use is rather complex. Another restriction of this approach is that it only works with a series of contiguous pages. Using hidden fields to store the session id shares that same drawback.

Store the session id in a cookie

Another approach is to store the session id in a cookie. Cookies are automatically sent by the browser to the server that set them. Thus they are the only way that we can implement contexts that persist across browser sessions. However, cookies also have their drawbacks. Users may choose to disable cookies for privacy reasons, and cookies can be accidentally deleted. In order to make the application as robust as possible, you may thus want to opt to combine both solutions by encoding the session id in a form parameter, and storing the session id in a cookie.

Encrypt the session id

In both approaches, the session id should be encrypted to ensure that it cannot be manipulated, whereby one user could "hijack" the session of another by guessing the session id. This involves computing a tamper-proof hash of the session id — see UNIQUE ID (29) for more details.

Timeout sessions on inactivity

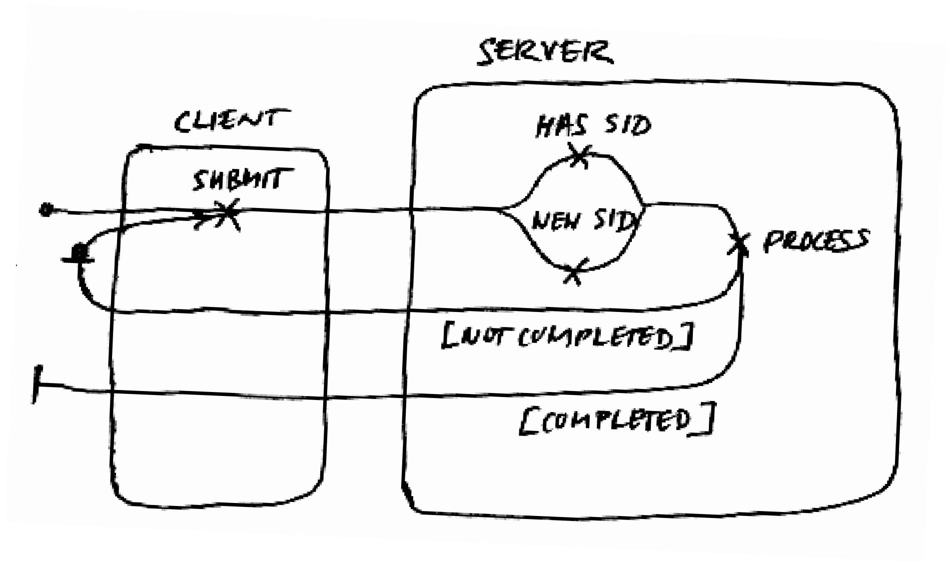
Another consideration is that a session should be terminated after a certain amount of inactivity, and the session id become invalid. This avoids that a session is kept open indefinitely, if the connection is prematurely closed. If the site is access controlled, this also ensures that the application is still used by the authenticated user, and not by somebody else.

Example: The price comparison site allows the user to save comparison results from multiple searches. This information needs to persist across browser sessions. We thus decide to use sessions, and store the saved results in the session information. In the code below, the function `getSid()` checks for the cookie named `sid`, and generates a new session id if necessary. We then return a new cookie to the client with an updated expiration date of 30 days from the current date.

```
my $q = new CGI;
my $action = $q->param("action") || "start";

my $sid = getSid($q);
my $cookie = $q->cookie(-name => "sid", -value => $sid,
  -expires => "+30d");
print $q->header(-cookie => $cookie, ...);

if ($action eq "start") {
  start($q, $sid, ...);
} elsif ($action eq "about") {
  about($q, $sid, ...);
} elsif ...
  .
  .
} else {
  error("don’t understand");
}

sub getSid {
  my $q = shift;
  my $sid = $q->cookie("sid");
  unless ($sid) { $sid = newSid(); }
  return $sid;
}
```

Therefore:

For maintaining medium or long term context, or if your site uses sessions to restrict access, store a session id at the client, and the associated session information at the server. The session information is stored in a database, using the session id as a database key.



***

Storing session information on the server rather than embedding it in a web page has several benefits. One is that he solution is more robust against interruptions of the navigation flow. However, out-of-order requests still need to be handled appropriately. Using DIRECTED SESSION [16] you can enforce that users visit the pages of your site in a certain order.

Also, keeping the session information on the server ensures the privacy of sensitive information. However, the input provided by a client still needs to be validated. Otherwise input can include embedded scripts that end up tricking other users of the application into releasing potentially sensitive information (cross-site attack). One pattern to use for validating input from an untrusted client is CLIENT INPUT FILTER [16].

The encryption of session ids is discussed in UNIQUE ID (29). The use of sessions to control access to web sites is further explored in ACCESS CONTROLLER (33). The use of this pattern has been documented in the TennisShoppe case study [14], and in Karkkhainen's tutorial [15]. Two related patterns that discuss HTTP session handling are AUTHENTICATED SESSION [16], and FRONT DOOR (17) [19].

Unique Id

You are using SESSION (24) to maintain the state of your web application between user requests. You need to encrypt session ids to ensure that they cannot be manipulated, preventing users from “hijacking” other sessions.

***

If session ids are sent in unencrypted form, users can gain access to another user’s session (session stealing). You need a way of ensuring that a session id has not been tempered with.

TBD

Therefore:

Create sufficiently unique session ids using a cryptographic hash (eg MD5) to ensure that sessions ids cannot be easily guessed. The hash should include some identification of the request source (eg source address, source port, or user agent) for validating the session id.

***

TBD

Offload Work to Client

You want to provide immediate feedback to the user for certain functions without the delay caused by requesting a new page from the server.

***

In the standard thin client approach, forms simply accept input and pass it to the server for processing. Validation requires a roundtrip to the server. You would like to make the application more responsive.

TBD

Therefore:

Offload functions such as input validation, or change propagation to client-side scripts. You still need to validate the input at the server, but the validation logic no longer needs to provide user feedback.

***

The solution suggested by this pattern is to provide feedback to the client without actually submitting the data to the server. However, the server-side application still needs to validate that the data is correct, regardless of what the client code does. On the other hand, the application logic will be simpler, because it does not need to give sophisticated error messages.

Conallen [8] documents a similar pattern called THICK CLIENT.

Acknowledgements

Thanks to my shepherd Peter Sommerlad for his fruitful feedback.

References

[1] Alexander, C., A Pattern Language, Oxford University Press, 1977

[2] Apache Software Foundation, Cocoon, *cocoon.apache.org/2.0*, last accessed in May 2003

[3] Apache Software Foundation, AxKit, *axkit.org*, last accessed in May 2003

[4] Buhr, R., Use Case Maps as Architectural Entities for Complex Systems, IEEE Transactions on Software Engineering, 1131-1155, 1998

[5] Burke, E., Java and XSLT, O'Reilly, 2001

[6] Buschmann, F., and Henney, K., A Distributed Computing Pattern Language, European Conference on Pattern Languages of Programming (EuroPLoP), 2002

[7] Carlson, D., Modeling XML Applications Using UML: Practical e-Business Applications, Addison-Wesley, 2001

[8] Conallen, J., Building Web Applications with UML, Addison-Wesley, 2003

[9] Cunningham, W., Wiki in HyperPerl, *c2.com/cgi/hp?WikiInHyper Perl*, 1995

[10] Daum, B., and Mertens, U., System Architecture with XML, Morgan Kaufmann, 2003

[11] Dougherty, D., LAMP: The Open Source Platform, O'Reilly, www.onlamp.com, 2001

[12] Fernandez, E., Liu, Y., Pan, R., Patterns for Internet Shops, Conference on Pattern Languages of Programming (PLoP), 2001

[13] Fowler, M., Patterns of Enterprise Application Architecture, Addison-Wesley, 2003

[14] Guelich, S., Gundavaram, S., and Birznicks, G., CGI Programming with Perl, O'Reilly, 2000

[15] Kärkkäinen, S., Unix Web Application Architectures, *http://webapparch.sourceforge.net/#23*, 2000

[16] Kienzle, D., Elder, M., et al, Security Patterns Repository, Networks Associate Technologi, *http://www.securitypatterns.com*, 2002

[17] Lyardet, F., and Rossi, G., Patterns for Dynamic Websites, Conference on Pattern Languages of Programming (PLoP), 1998

[18] Rossi, G., Lyardet, F., and Schwabe, D., Patterns for e-Commerce Applications, European Conference on Pattern Languages of Programming (EuroPLoP), 2000

[19] Sommerlad, P., Reverse Proxy Patterns, European Conference on Pattern Languages of Programming (EuroPLoP), 2003

[20] van Duyne, D., Landay, J., and Hong, J., The Design of Sites: Patterns, Principles, and Processes for Crafting a Customer-Centered Web Experience, Addison-Wesley. 2003

[21] Wallace, D., Ragget, I., and Aufgang, J., Extreme Programming for Web Projects, The XP Series, Addison-Wesley, 2003

[22] Welie, M., Interaction Design Patterns, *www.welie.com/patterns/index.html*, last accessed June 2003

[23] Wodtke, C., Information Architecture: Blueprints for the Web, New Riders, 2003

Appendix: Pattern Thumbnails

DATA SOURCE ADAPTER

Create a uniform interface for accessing data sources of the same type. Access a data source only through this interface. Changes to how the data source's implements the interface will not impact the code that accesses it.

WRAPPER

When extracting data from a web site, augment the data with metadata that indicates not only the type of each data element, but also its relationship to other elements. The added verbosity results in a processing overhead, but leads to a more reusable and extensible data representation.

ACCESS CONTROLLER

If you need to restrict access to some areas of your web site, authenticate the user in the central dispatcher. This ensures that authentication is implemented consistently, and eliminates potential "loopholes" allowing access to your application by other users than authenticated ones.

CHAIN OF APPLICATIONS

Instead of creating a complex application consider building smaller, more manageable applications that can be chained together in a pipe-and-filter fashion to provide the same functionality in a more flexible manner.

EXPOSE APPLICATION AS SERVICE

Factor out reusable functionality as a web service. This enables direct integration with other applications. Using this web service as input to your own application you can continue to provide a browser-based interface.