Web Developing: PLT Scheme vs. JSP

Sara Forghanizadeh

CPSC503 Course Project Final Report

1. Introduction

Based on my experience in developing Java Server Pages, when I first saw the "addition of two numbers" example in Scheme, I became surprised by its simplicity. So I decided to put Web-developing using Scheme servlets on test. The authors of PLT Scheme Web-server claim that their Web-server does not force an awkward decomposition of the program on the developers, and this fact results in cleaner, more modular and more maintainable application code; most of this is because of the power of continuations. Here are two of their exact sentences: *"Declarative languages win on the Web"* and *"The manual decomposition simply implements the continuation-passing style transformation!"*

The goal of this project is to investigate these claims by developing a Web-application in Scheme (using PLT Scheme Web-server) and comparing it to JSP. The equivalent JSP application was not implemented because I had some experience in JSP so I could compare the two technologies without developing the system using it; and besides in [7] I found a JSP application whose topic was a bit similar to the application I developed in Scheme, so I modified some parts of its code for the purpose of comparison.

This project compares Scheme servlets and JSP only considering readability, modularity and clean design. Issues like memory usage and performance are not covered in this report. Ease of development could not be a comparison factor for two reasons: First, creation of generators makes the task of developing Web-pages more automated; but even while using generators, readability, modularity and clean design are still important factors. Secondly, this is the first Scheme application that the author of this report has developed, and obviously dealing with Scheme is not an easy task in the first time; so I was not qualified to judge it.

2. Background Material

This section provides background material on Java Server Pages and PLT Scheme Servlets.

2.1 Java Server Pages

In the beginning, Java servlets were invented. In Java servlets, HTML code is included inside Java code. For writing each line of the HTML code you should use an "out.println()" method, e.g. "out.println("<HTML>"). But having the programmer write

an “out.println()” call per HTML line became a serious problem. Besides being a tedious and time-consuming task, it was difficult to have artists, content creators and programmers work on the same file.

Then Java Server Pages (JSP) technology was invented, and soon became popular. In this technology you could put Java code inside HTML pages, and the server would automatically create a servlet from the page. JSP was considered a "cheap" way for a Java programmer to write a servlet. But still interface developers and programmers had to work on the same file. So improvements were needed in order to separate interface from business logic.

One of the methods used to accomplish this goal is relying more on JavaBeans, so that much of the code inside the JSP page could be moved out of the page into the bean with only minimal hooks left behind where the page would access the bean. JavaBeans components are Java classes that can be easily reused and composed together into applications. Any Java class that follows certain design conventions can be a JavaBeans component [9]. Figure 1 shows an example of using JavaBeans. In Figure 1.a, a shopping cart object is created in simple JSP. In 1.b, the equivalent code is shown if the shopping cart object conforms to JavaBeans conventions.

```
<%
   ShoppingCart cart = (ShoppingCart)session.
      getAttribute("cart");
   // If the user has no cart, create a new one
   if (cart == null) {
      cart = new ShoppingCart();
      session.setAttribute("cart", cart);
   }
%>
```

Figure 1.a

```
<jsp:useBean id="cart" class="cart.ShoppingCart"
   scope="session"/>
```

Figure 1.b

In “Model 2 Design”, named after architecture laid out in the JSP 0.92 specification and based on the model-view-controller pattern, requests are submitted to a servlet "controller" that controls business logic, and generates an appropriate data "model" to be displayed. This data is then passed internally to a JSP page "view" for rendering, where it appears to the JSP page like a normal embedded JavaBean. The appropriate JSP page can be selected to do the display, depending on the logic inside the controlling servlet. The JSP page becomes a nice template view.

The controlling servlet in fact acts as a state machine. In this Model the logic that controls the flow of the program is less scattered across the JSP pages, and the design is

cleaner. The downside of this model is its complexity. The state machine is not easy to read, and in fact, implementation of user interaction logic is scattered across multiple action classes and a state machine implementation. The result is difficulties in mapping the code to the requirements and in the process of visual representation. The code becomes harder to maintain and understand.

So in general, the difficulties with Web-programming using Java servlets and JSP are as follows:

- Since HTTP is a stateless protocol, Web applications themselves must maintain conversational state with each client. Using JSP and Java servlets, the application is broken into a number of small pieces, each of which is capable of handling a small number of requests. Breaking the application into pieces may make it harder to modify the application, and besides, this is not the natural decomposition you would perform if this application was not a Web-application

- Legacy applications (the ones that have not been used as Web-applications before) are difficult to adapt to the Web, because they are decomposed in a different way

- Techniques to maintain state often require each piece of the Web application to manipulate objects in a ”session” hash table (or even worse, in hidden fields), resulting in side-effects and dependencies that can cause subtle bugs

- In Model 2 design, depending on the size of the state machine and the amount of data required to maintain a client's current state, the application logic could become unnecessarily cluttered and complex

- users can change the state by using the back-button or open multiple windows

The above problems are caused by asynchronous nature of Web applications, and inability of Java programming language to offer execution context as first class object that could be stopped and resumed at will – in other words, lack of support for “continuations”. There are frameworks like Cocoon that try to provide this feature for java, implementing some sort of serializable threads; but they use some extra files such as XML files, and using them is a complicated task.

2.2 Scheme Servlets

Since continuation is a first-class value in Scheme, continuations can be used to add state to a Web application. Whenever the Web application needs input from the user, the current continuation associated with that user is saved. When the user responds with some information, the saved continuation is restored, and the input provided by the user is returned as the value of the continuation. This structure also makes book-marking a page, and using the back button possible in a natural way.

The teachpack *servlet2.ss* provides a simplified API for PLT Scheme servlets. The teachpack does not require any understanding of HTML and higher-order functions. It provides the programmers a series of constructors as follows,

```
make-password ; asking for a password
make-number   ; asking for a number
make-yes-no   ; asking for a Yes-No style answer
make-boolean  ; asking for a true-false style answer
make-radio    ; requesting one choice
```

as well as data definition structures such as *Forms* and *Tables*, and a series of functions such as *single-query* for posting a single question, *extract/single* for extracting a response, or *inform* for posting some information on the Web and waiting for continue signal.

Here is an example of using Servlet2 teachpack for writing a Web-program using this teachpack:

```
(inform
 "Calculator Result"
 ( number--> string
   (+ (single-query (make-number "Enter a number"))
      (single-query (make-number "Enter another number")))))
```

Figure 2

But the Web-interface provided by these functions is really simple. They might be appropriate for an HtDP (how to design programs) course; or suitable when you need to develop a series of interactive pages without worrying about any kind of colors or headers on these pages! In fact for developing real Web-applications it is not possible to use the functions provided by this teachpack – though they make life seem so easy. They are not really flexible, and there is no way to combine them with graphics (at least I found no way to do that).

So in fact you need to use the teachpack *servlet.ss* for developing real Web-applications. You can put XHTML inside your code for developing the Web interface through X-expressions. The following figure 3 shows the equivalent codes in HTML and Scheme.

In *servlet.ss* the data definitions represent HTTP requests and Web page responses using *request* and *response* data structures. Some of the provided functions are *send/suspend* for sending the suspender's page to the browser and waiting for the browser's request, *send/finish* for sending the response to the browser and terminating the servlet, *extract-binding/single* and *extract-binding* for consuming the symbol of an HTML form field and a bindings environment.

```
<html>
  <head>
    <title>
      A Simple Web Page
    </title>
  </head>
  <body>
    <h1>My First Web Page</h1>
    <p>Hello World.</p>
  </body>
</html>
```

```
'(html ()
   (head ()
     (title ()
       "A Simple Web Page"))

   (body ()
     (h1 "My First Web Page")
     (p "Hello World")))
```

Figure 3

Scheme provides two mechanisms for writing modular programs: Modules and units, which are specified as follows:

```
(module a-module-servlet mzscheme
  (provide interface-version timeout start)
   …))
```

and

```
(require (lib "unitsig.ss")
         (lib "servlet-sig.ss" "Web-server"))
(unit/sig ()
  (import servlet^)
   ;;; ... servlet code ...
)
```

3. Example Application

The Scheme Web-application implemented in this project is named 511 Online Store. It contains the main flow for online shopping: users should login to the system, select a category in the first page, select a subcategory in the next page, select an item, add it to the shopping cart and view the contents of the cart; then either checkout or continue shopping and adding other items to the cart if desired. Checkout asks for the credit card number (and does not validate it in reality!).

This application is implemented using MzScheme language which is included in DrScheme. The PLT Scheme Web-server has been used as the server.

4. Experience

4.1 Implementation – Scheme

You can find snapshots of this application in Appendix. There are three main modules in the implemented 511 Online Store Web application: interface, business and data.

Interface module: The procedure *make-page* shown in figure 4 taken from [] (but modified) behaves just like a template for producing the graphical part of the pages (The banner as well as the footer containing information about login). It is parameterized over title, user, and body-generating procedure. It returns a procedure that takes a continuation url and produces an xexpr.

```
(define (make-page title user body)
  (lambda (url)
    `(html (head (title ,title)
                 (link [(rel "stylesheet")
                        (href "/default.css")
                        (type "text/css")
                        (title "specified")]))
           (body
            (div [(style "background-color: #333300; color: white")]
                 (table [(width "100%")]
                        (tr
                         (td
                          (div [(style "color: white; text-align: center")]
                               (div [(style "font-size: x-large")]
                                    "511 Online Store")
                               (div
                                (span [(style "font-size: large")]
                                      "CPSC511 Course Project")
                                (span " using ")
                                (span [(style "font-size: large")]
                                      "Scheme ")))))))
            (div [(style "border-color: #336633; border-style: solid")]
                 (table [(width "100%")]
                        (tr [(style "text-align: center")]
                            (td ,(make-main-link 'about "About" url)))))

            (div [(style "padding: 4ex 8ex 4ex 8ex")]
                 ,(body url))
            (div [(style "background-color: #336633; color: white; font-weight: bold")]
                 ,(if user
                      `(span "Logged in as user "
                             (em ,user))
                      `(span "Not logged in")))))))
```

Figure 4

Other procedures in this module are used for producing bodies of different pages; most of them contain "forms". These procedures are passed to *make-page* later. One of the procedures is shown in figure 5. It produces the body of a page for showing the list of categories. The "map" operation is used for mapping the categories to radio list options:

```
;; categories-body : -> list -> xexpr
;; Generates the category selection body.
(define (radio-list-body category)
  (lambda (url)
    (let ((choice-num -1))
    `(form [(action ,url) (method "post")]
           `(p "Select a category from the following list:" )
           ,@(map (lambda (choice-x)
                    (set! choice-num (add1 choice-num))
                        `(p
                            (input ((type "radio")
                                    (name "choice")
                                    (value , (number->string
                                              choice-num)))
                            , choice-x))
                        category)
           (input ((type "submit")(value "Next")))))))
```

Figure 5

Business module: This module contains a send-page procedure for every page, which uses *send/suspend* and passes the corresponding body to the *make-page* procedure. It also extracts the response from *request-bindings* using *extract-binding/single* function, and returns it if it should be used by other pages in the flow of the program. If any control on the page is needed, it will be put in this function. Figure 6 is an example of one of these functions, used for sending the login page:

```
; send-login-page : -> string
(define (send-login-page)
  (let* [(bindings (request-bindings
                    (send/suspend
                     (make-page "Login" #f (login-body)))))
         (username (extract-binding/single 'username bindings))
         (password (extract-binding/single 'password bindings))]
           (if (login-invalid? username password)
               (send-login-page))
  username))
```

Figure 6

This function calls *make-page,* with parameter *login-body*. It extracts username and password from the bindings; then calls the *login-invalid?* from the data module. If the

username and password are not valid, it calls itself and the login page will be shown again. When the username and password are valid, the username will be returned.

The business module also contains the main flow of the program. Figure 7 shows the power of continuations in controlling the flow of the program:

```
;checkout-flow
(define (checkout-flow user shopping-cart)
  (send-checkout-page user shopping-cart)
  (send-payment-successful-page user))

;shopping-flow
(define (shopping-flow user shopping-cart)
    (let* (
           [cat-num (send-categories-page user)]
           [subcat-num (send-sub-categories-page user cat-num)]
           [item-num-and-action
              (send-items-page user cat-num subcat-num)]
           [item-num (string->number(list-ref item-num-and-action 0))]
           [action (list-ref item-num-and-action 1)])
           (if (eq? action (string->symbol "Add to Cart"))
              ((set! shopping-cart
                     (cons (get-item cat-num subcat-num item-num) shopping-cart))
               (set! action
                     (send-show-shopping-cart-page user shopping-cart))
               (if (eq? action (string->symbol "Checkout"))
                   ((checkout-flow user shopping-cart)))
               (shopping-flow user shopping-cart)))
           (shopping-flow user shopping-cart)))

;main-flow
(let [(user (send-login-page))(shopping-cart ())]
  (shopping-flow user shopping-cart))
```

Figure 7

The main flow would be like this:

- user name is returned by the send-login-page, and shopping-cart is initialized to an empty list
- the shopping-flow begins
- the user selects a category through the categories-page; cat-num is returned
- the user selects a subcategory; subcat-num is returned
- the user either chooses an item and adds it to the shopping cart; or continues shopping
- if asked for adding the item to the shopping cart, the list of the items already in the shopping cart is shown, along with item prices and the total price. user can either checkout, or continue shopping

- if checkout is chosen, the checkout-flow begins

Data module: In order to reduce complexity, files are being used as the database in this application. Loading files into lists is an easy task; it can be done just by a single “load” command like this:

```
(define *categories-data* "categories-data.ss")

(define (get-categories)
   (load *categories-data*))
```

The file “categories-data.ss” contains the list of all categories; in fact here is what it contains:

```
'("Musical Instruments" "Electronics" "Jewelry")
```

The file “sub-categories-data.ss” contains lists of subcategories for each category. Information about the items is stored in “items-data.ss”. It contains name, price and a link to the item’s picture; figure 8 shows a part of this file. This module provides services such as a function for retrieving an item given the category, subcategory and item number.

```
...
((("Floating Heart Pendant with Diamond   "
   "$39.99"
   "http://images.amazon.com/images/P/B0001KOBJO.01-A365Q7M5Z8FG4S._SCMZZZZZZZ_.jpg")

   ("14k Yellow Gold Round Diamond Stud Earings   "
    "$58.00"
    "http://images.amazon.com/images/P/B0002CEC1O.01._SCMZZZZZZZ_.jpg"))
...
```

Figure 8

4.2 Implementation – JSP

As mentioned before this application was not really implemented in JSP, but here is what it would look like in JSP:

There will be an Item class, probably a User class, a Controller servlet, as well as 7 JSP pages in order to implement this system. The shopping-cart items are kept in a Vector which will be put in the session. Figure 9 shows what the Controller servlet would look like.

```
public class ShoppingServlet extends HttpServlet {
    public void init(ServletConfig conf) throws ServletException {
        super.init(conf);
    }
    public void doPost (HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
            HttpSession session = req.getSession(false);
            if (session == null) {
            res.sendRedirect("http://localhost:8080/error.html");
    }
    User user = (User)session.getValue("user");
    Vector buylist= (Vector)session.getValue("shoppingcart");
    String action = req.getParameter("action");
        if (action.equals("log in")) {
            ...
        } else if (action.equals("Shopping")) {
            ...
        } else if (action.equals("Categories")) {
            ...
        } else if (action.equals("Sub-Categories")) {
        } else if (action.equals("items")) {
        } else if (action.equals("Add to cart") {
            buylist.addElement(newItem);
            session.putValue("shoppingcart", buylist);
            String url="/jsp/shopping/ShowShoppingCard.jsp";
            ServletContext sc = getServletContext();
            RequestDispatcher rd = sc.getRequestDispatcher(url);
            rd.forward(req, res);
        } else if (action.equals("Checkout") {
            ...
            req.setAttribute("amount",amount);
            String url="/jsp/shopping/Checkout.jsp";
            ServletContext sc = getServletContext();
            RequestDispatcher rd = sc.getRequestDispatcher(url);
            rd.forward(req,res);
        }
    }
}
```

Figure 9

Figure 10 shows what a part of the show-shopping-cart or checkout page would look like.

```
<center>
<table border="0" cellpadding="0" width="100%" bgcolor="#FFFFFF">
<tr>
       <td><b>Name</b></td>
       <td><b>Price</b></td>
       <td><b>QUANTITY</b></td>
</tr>
<%
  Vector buylist = (Vector) session.getValue("shopping.shoppingcart");
  String amount = (String) request.getAttribute("amount");
  for (int i=0; i < buylist.size();i++) {
       Item anOrder = (Item) buylist.elementAt(i);
%>
<tr>
       <td><b><%= anOrder.getName() %></b></td>
       <td><b><%= anOrder.getPrice() %></b></td>
</tr>
<%
  }
  session.invalidate();
%>
<tr>
       <td><b>TOTAL</b></td>
       <td><b>$<%= amount %></b></td>
       <td>     </td>
</tr>
</table>
```

Figure 10

4.3 The advantages observed

The authors of PLT Web-server claim that the power of continuations results in a cleaner, more modular and more maintainable design. According to the example application developed, I believe this is true. Figure 11 shows the component diagram for Online Store application in Scheme and JSP:

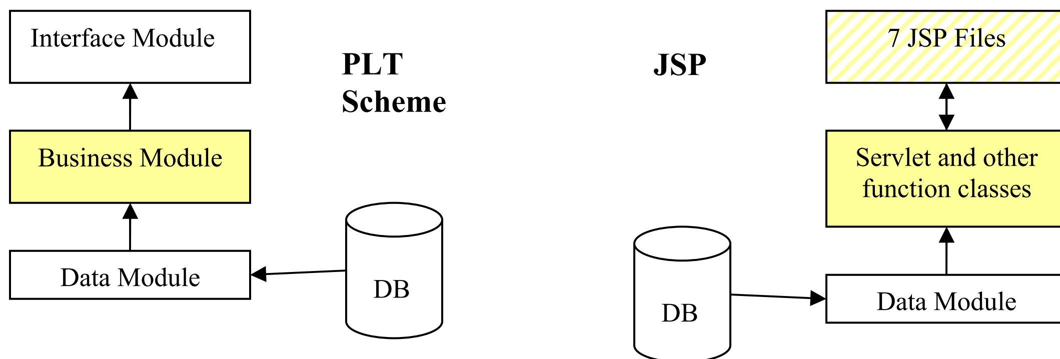


Figure 11

The yellow color in these diagrams shows where the business logic is focused. Suppose you wish to see the flow of the program in these two systems by looking only at the source code for a debugging purpose. In the Scheme system, the only thing you need to do is to look at the business module; the flow can be clearly seen in the part shown in figure 7 in section 4.1. But in the JSP system, first you should look at the state machine in the controller servlet, and guess what the beginning state is. If the first page is an HTML file (not a JSP one) then finding it would be even harder. Then you should open the first HTML or JSP file, look for the actions of the forms or possible hidden fields or values in the sessions, to find out what the request is. Then you should get back to the state machine. You find the current state, open the next JSP file, find the next request, and so on. In brief, you will have to investigate seven JSP files and the controller servlet.

Of course, as it is possible to specify several targets in a page in more complex systems, the flow in Scheme might also look like a state machine, but at least a state machine like this:

```
case (action)
        action1
                flow1
        action2
                flow2
        …
```

This comparison shows how Scheme could be more readable than JSP thanks to the power of continuations But besides continuations, there are other mechanisms and properties in scheme that make it a higher-level language for developing Web applications compared to JSP.

One property is that procedures are *first class* objects in Scheme, and also the macros are really powerful. This makes using templates for creating user interface very easy, just like what you saw in section 4.1 in the interface module. Templates can be used in JSP as well, but the interesting point is, this feature didn't exist in JSP in the beginning and was added into it later; but scheme naturally supports creating and using templates – there was no need for extensions.

Another feature in Scheme that seems to be very useful in Web-programming is lists and mappings. There is an example in [6] which shows the shortcoming of JSP in implementing the MVC pattern, where the Model, View and Controller should be distinct. Suppose you'd like to display the existing orders in the shopping cart. There are two choices in JSP for doing this:

1. Using a JSP 1.0-like mechanism, you have a Vector containing order objects. This Vector is available to the JSP as a JavaBean:

```
<table>
   <%
      orderVector = customer.getOrders();
      for( int i=0; i<orderVector.size(); i++){
         Order o = orderVector.elementAt(i);
   %>
   <tr>
      <td><%= o.getDate() %></td>
      <td><%= o.getOrderID() %></td>
   </tr>
   <% } %>
</table>
```

Figure 12

2. Using a JSP 1.1-specific mechanism, you have a custom tag similar to <javaworld:orderlist/>. You should configure the tag library file and import the correct tag library at the top of the JSP file. The code will probably look like this:

```
...
BodyContent body = this.getBodyContent();
JspWriter writer = body.getEnclosingWriter();
writer.print("<table>");
      orderVector = customer.getOrders();
      for( int i=0; i<orderVector.size(); i++){
         Order o = orderVector.elementAt(i);
         writer.print("<tr>");
         writer.print("<td>" + o.getDate() + "</td>" );
         writer.print("<td>" + o.getOrderID() + "</td>" );
         writer.print("</tr>");
      }
      writer.print("</table>");
      ...
```

Figure 12

Both these options violate the MVC pattern. The first requires elements of the model in the view; if the structure of Order or Customer objects changes, it would be needed to make changes to the Java elements present in the view. The second option requires the elements of the view in the model. If you change the HTML table (the view) to an unordered list, the custom tag would require modifications. This is even worse than the first option, because it forces recompilation of the code, regression testing, stopping the servers, and new or updated installation.

But I noticed that in Scheme, lists and the mapping operation solve this problem! Here is the part of XHTML code I implemented for showing the list of orders in the shopping cart:

```
;; show-shopping-cart-body : -> list string -> xexpr
;; generates the body for showing the items in the shopping cart, their prices, and
;; the total price (that is passed to it as a string)
(define (show-shopping-cart-body shopping-cart total)
  (lambda (url)
    `(form [(action ,url) (method "post")]
           (table
            (tr (td "Item")
                (td "Price"))
            ,@(map(lambda (shopping-cart-item)
                    `(tr (td , (list-ref shopping-cart-item 0))
                         (td , (list-ref shopping-cart-item 1))))
                  shopping-cart)
            ,`(tr (td , "Total Price:")
                  (td , total))
            ,`(tr (td "")
                  (td (input [(type "submit") (name "action")
                                              (value "Checkout")]))
                  (td (input [(type "submit") (name "action")
                                              (value "Continue Shopping")]))))))))
```

Figure 13

The mapping function here only knows that a list of pairs will be given to it, and just maps this list to table rows. There is no element from the View in the Model, and vise versa. The power of lists and mappings makes the separation of business logic from the interface much easier in Scheme in comparison to JSP. As you can see in the example codes, there is really little non-XHTML code inside the Web pages in comparison to JSP; they are free from objects names, the code dealing with session, hidden inputs or custom tags.

One more thing I liked about writing the Web-interface in scheme was the XHTML code in scheme is more compact than regular XHTML using tags inside "<>". First I thought it will be somehow confusing to see too many parentheses in the code, but then I found out that it makes my mind more focused because in fact I can concentrate more on the contents rather than the long tags. DrScheme editor makes dealing with parentheses easy as when you move on any parentheses, it puts some shadow on the part of the code that begins or ends with it.

4.3 Trade-offs

Of course, if there are advantages in using Scheme, there must be some trade-offs. Issues like performance, memory usage and garbage collection that were not covered in this report are as important as – or even more important than – code readability and modularity. Though maintaining the programs is a costly task, in the end what the end-user cares about is the performance of your Web-site.

One more point is, though it is possible to write modular programs with Scheme, it doesn't mean that people will really use it in a modular way; I saw a few examples on the Web where they had used state machines and hidden fields in Scheme, just like writing programs in JSP. It is also very possible to misuse continuations, or write the whole parts of the program in the same module!

And, the fact remains that is Scheme itself a more readable language in comparison to Java? Of course it depends highly on your experience with Scheme; when I first started this project the Scheme code was less readable to me than when I finished it. Java code can be more easily read by non-experts compared to Scheme. I wonder if this is a disadvantage or not because after all it is most likely that experts are going to read or maintain the code; but it is a fact that Scheme sometimes really hurts! Obviously working with it and reading it needs more mental effort compared to Java; especially when the logic of the program is complicated. This is probably one of the main reasons why Java is a more popular language than Scheme. Scheme and Lisp might offer a "better" way of thinking, but at the same time, this thinking method needs more effort.

4.4 How the tools can be improved

Creating *servelt2.ss* teachpack was a very cool idea. Example codes that use the functions provided by this teachpack attract non-scheme Web-programmers attention at the first place, such as the addition of two numbers example. Because they simplify the world by providing a Web-interface, these functions make it easier to illustrate the power of continuations in Web-programs.

Unfortunately it is not possible to use this teachpack in practice. The reason is they provide a very simple Web-interface and it is not possible to add graphics to the page while using them. But the existence of such a library shows that if you create a similar customized library, it would be very easy to use it! And in fact creating such a customized library is an easy task – something similar to the interface module in our shopping card application.

But now that they have created the servlet2.ss, it would be perfect if they provided the same functions, but with just the "form" part of it instead of the whole page. That way it was possible to create an XHTML template and place the forms inside it. I think this way more people could use these functions in practice.

As I am a beginner in Scheme, I have problems dealing with it that might not really be general; but one problem I have is when I look at the code and see a function or macro, I can hardly recognize whether that function is a library function, or from another module/unit, or is declared inside the same unit/module. This makes the code less readable to me especially in long scheme programs. I suppose if the scheme programming environment provided a way for distinguishing between these two, the code would become clearer and more readable.

5. Summary

Continuations seem to be essential for writing Web-programs, because the nature of Web-applications is asynchronous: the users do not answer immediately when the request is sent to them. So a mechanism is needed to stop the execution of the program (thread) and resume it later on when the response is received; continuations do this job. As Java and JSP do not support continuations, it is not possible to see the control flow of the program in the code and the business logic is scattered in the application. You have to choose a decomposition which you wouldn't choose if it was not a Web-application. This makes these applications less readable and modular.

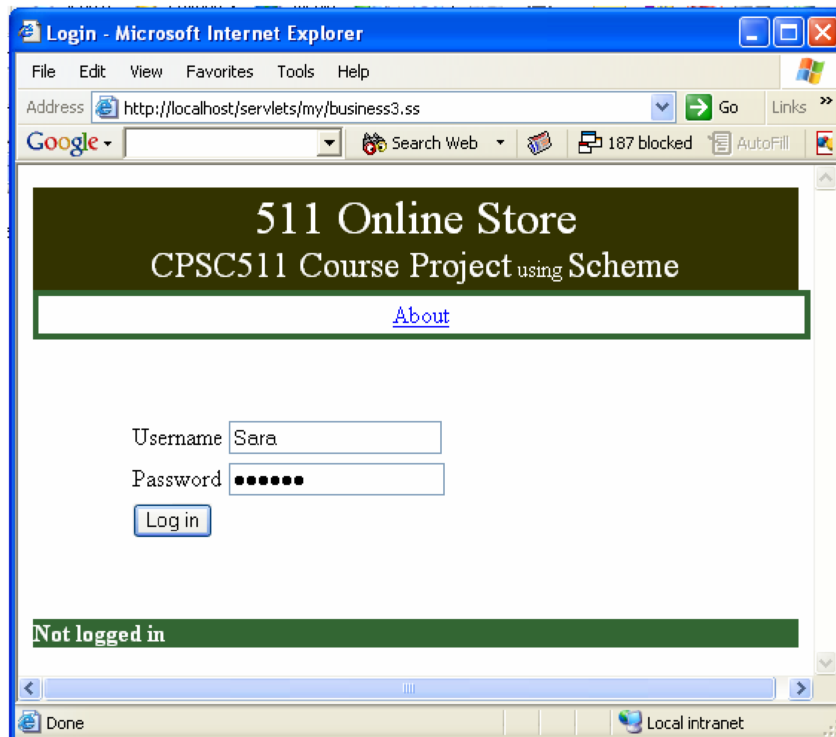
Besides continuations, there are other properties and mechanisms that make Scheme an appropriate (or maybe higher level) language for developing Web-programs. These properties are procedures as first-class objects, macros, mapping operations and lists.

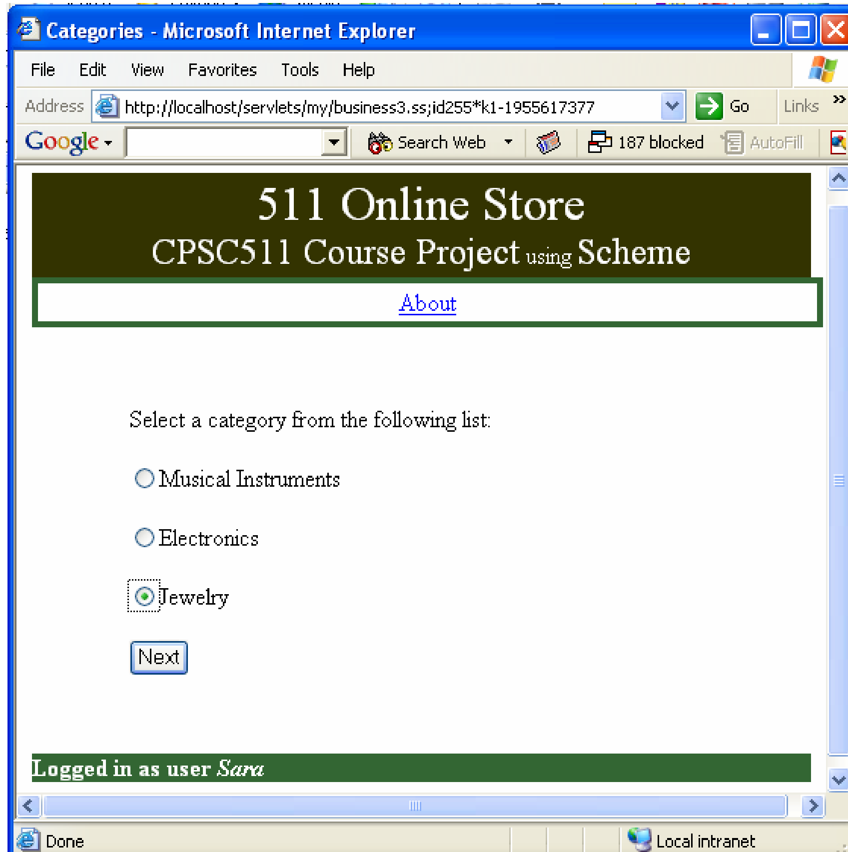
But of course there are trade-offs: JSP and Scheme servlets should be compared according to their performance and possible problems in memory usage. And this fact cannot be ignored that Scheme procedures themselves are less readable and more difficult to deal with compared to Java. But for people who are good at both Java and Scheme, Scheme – according to readability and cleaner design – seems to be a better choice for developing Web applications.

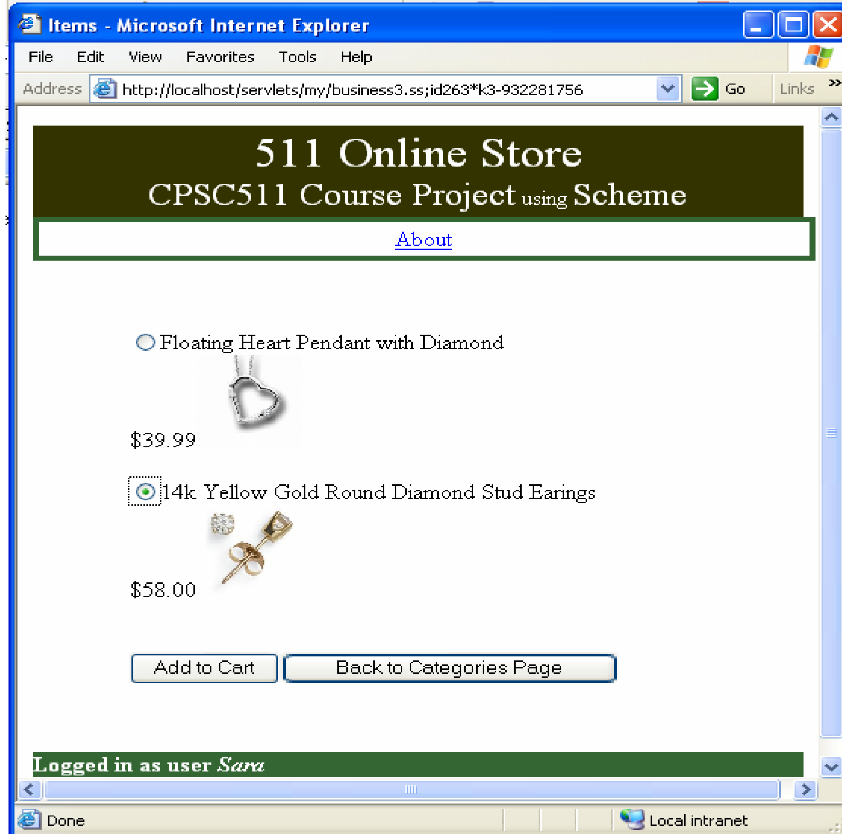
References

1. PLT Web Server manual (http://download.plt-scheme.org/scheme/docs/html/Web-server/)
2. Shriram Krishnamurthi, The CONTINUE Server, *Symposium on the Practical Aspects of Declarative Languages,* 2003
3. Paul Graunke, Shriram Krishnamurthi, Van der Hoeven and MatthiasFelleisen. "Programming the Web with High-Level Programming Languages". *Proceedings of ESOP* 2001.
4. Shrim Krishnamurthi et al., How to Design Programs, The MIT Press, September 2003 Version
5. Jason Hunter, "The Problems with JSP", http://www.servlets.com/soapbox/problems-jsp.html
6. Vincent DiBartolo, Free Maker: An open alternative to JSP, http://www.javaworld.com/javaworld/jw-01-2001/jw-0119-freemarker.html
7. Govind Seshadry, Understanding Java Server Pages Model 2 Architecture, http://www.javaworld.com/javaworld/jw-12-1999/jw-12-ssj-jspmvc.html
8. Abhijit Blapurkar, Use continuations to develop complex Web applications, http://www-106.ibm.com/developerworks/library/j-contin.html
9. Stephanie Bodoff, JavaBeans Components in JSP Pages, http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/JSPBeans.html
10. Ken Williams**,** Implementing a Simple Picture Gallery using spgsql, Scheme Boston: January 2003 meeting

Appendix – Snapshots of 511 Online Store Web application in Scheme






Items - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://localhost/servlets/my/business3.ss;id263*k3-932281756
Go
Links
511 Online Store
CPSC511 Course Project using Scheme
About
Floating Heart Pendant with Diamond
$39.99
14k Yellow Gold Round Diamond Stud Earings
$58.00
Add to Cart
Back to Categories Page
Logged in as user Sara
Done
Local intranet

Shopping Cart - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://localhost/servlets/my/business3.ss;id264*k8-1608513666
Go
Links
511 Online Store
CPSC511 Course Project using Scheme
About
Item
Price
Floating Heart Pendant with Diamond
$39.99
14k Yellow Gold Round Diamond Stud Earings
$58.00
Total Price:
97.99
Checkout
Continue Shopping
Logged in as user Sara
Local intranet