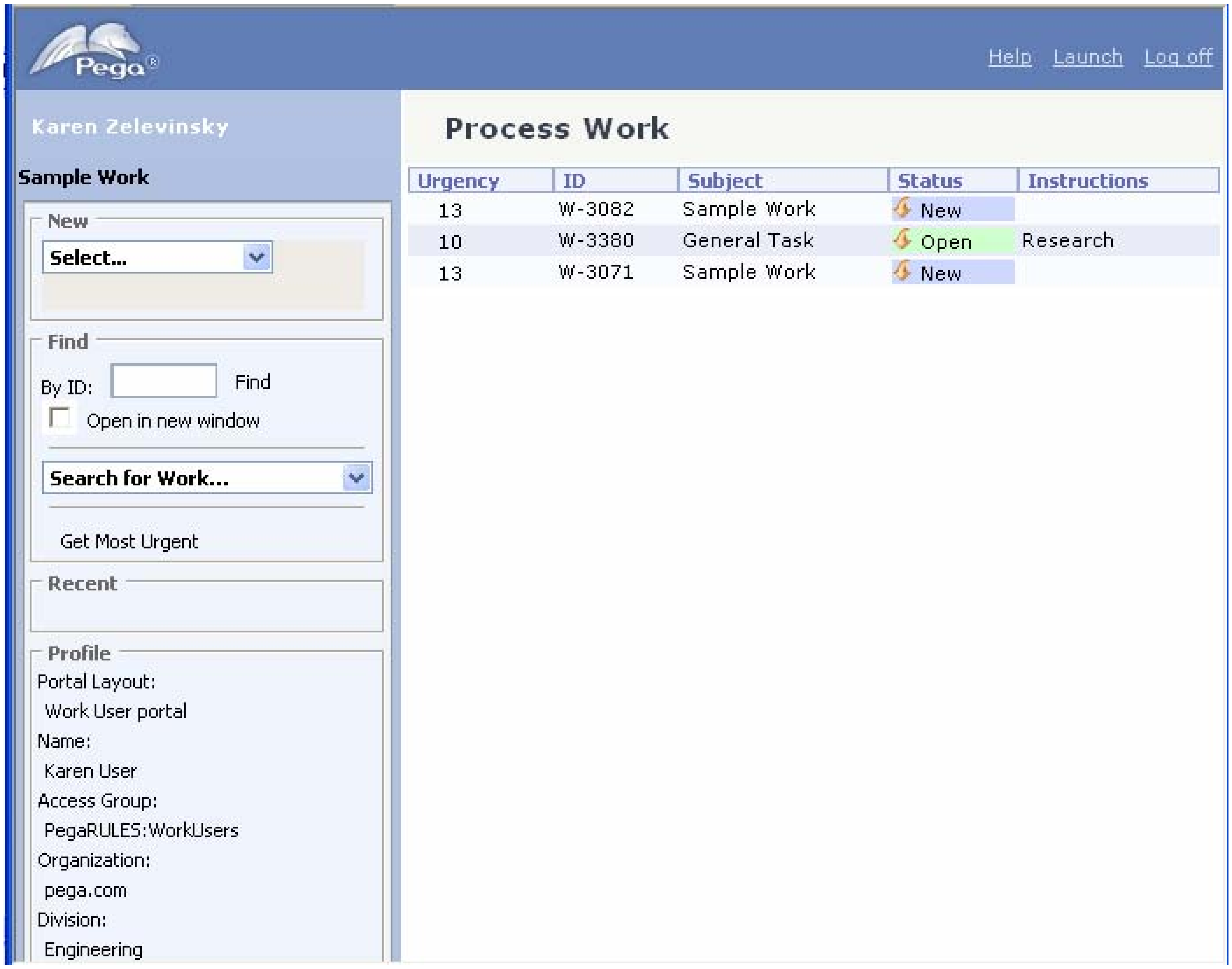
Karen Zelevinsky
CSCI E-180
Due 5/25/2006

Project 3 – Making an Accessible Website

Overview

My company produces a thin-client (IE only) application. We have never given any thought to supporting users with disabilities, so I decided to assess and solve some of our accessibility issues. Please note that this effort was not part of any project at work and everything was accomplished in my own time. The screen that chose to address is the home screen that end-users see, which looks like this:



Portal Layout:
Work User portal
Name:
Karen User
Access Group:
PegaRULES:WorkUsers
Organization:
pega.com
Division:
Engineering

Process Work

Urgency	ID	Subject	Status	Instructions
13	W-3082	Sample Work	New	
10	W-3380	General Task	Open	Research
13	W-3071	Sample Work	New	

Analysis

The first step was to create a stand-alone series of HTML screens based that screen. We use framesets and iframes because we use the same framework to support both end-users and more sophisticated users such as managers and system administrators. I originally anticipated that the frames themselves might be a problem for accessibility but it turns out that frames are supported and in fact can be useful because they organize the content into clear sections. For that reason I kept the original layout of the screen and ended up creating 6 html files:

1) Frameset – basic frameset
2) NavigationPane – left pane with options for find by ID, basic reports, and opening user information
3) WorkPoolClasses – list of new kinds of work that can be entered
4) PortalTitleBar – Logo and links for help, launch, and logoff
5) SpaceTitleBar – Information about which area of the product the user is in
6) ShowWorkView – List of work assigned to the current user

I also saved the supporting style sheets and images into the directory. I updated the HTML to remove most of the javascript and replaced it with alerts that would indicate that an action was attempted. Other than that, I kept the HTML mostly untouched.

Once I had a working HTML, I used WebXACT (http://webxact.watchfire.com/) to analyze how our pages fared in their accessibility analysis. They compare the page against the W3C Web Content Accessibility Guidelines.

Baseline Analysis – Number of Errors per HTML

HTML Name	**Priority 1**	**Priority 2**	**Priority 3**
Frameset	1	3	1
NavigationPane	1	4	3
WorkPoolClasses	0	2	2
PortalTitleBar	1	2	3
SpaceTitleBar	0	3	2
ShowWorkView	1	3	2
Total	**4**	**17**	**13**

Priority 1 issues included:

1) Provide alternate text for images
2) Give each frame a title

Priority 2 issues included:

1) Use public text identifier
2) Explicitly associate form controls and their labels
3) Make sure event handlers don’t require a mouse
4) Include a document TITLE
5) Provide a NOFRAMES section

6) Use relative sizing and positioning rather than absolute

Priority 3 issues included:

1) Identify language of the text
2) Provide a summary for tables
3) Separate adjacent links with more than whitespace
4) Include default, place-holding characters in edit boxes

I also performed the test of trying to access all actions in the page without using a mouse and failed dismally. This test confirmed the need to make sure that you use hands-on testing as well as automated tools such as WebXACT. Although ensuring that event handlers don't require a mouse is listed as a priority 2 issue, since our website is meant to process data rather than display information, the fact that I couldn't process data without a mouse is clearly more severe than the priority 1 issue of not having an alternative text for the logo image.

I encountered two major issues with keyboard accessibility:

1) **Select elements with onchange handlers –** The onchange event fires for mouse-users when they select an option but fires from the keyboard if the user clicks down just to see what the options are. My web research revealed that many sites suggested having a separate button to take action, but since we like saving our users the extra click I will describe a different solution below.
2) **Table of work with onclick handler –** The onclick event is exclusively for mouse-users. Users with a keyboard can't even get focus on the table of work because it is just a table with no form elements to tab to.

I also noticed that tabbing through the entire set of elements was time-consuming and that it would be useful to have a way to jump to commonly used areas of the screen.

Actions

First I went through the errors flagged by WebXACT and solved as many as I could.

Priority 1

1) **Provide alternate text for images**
   Example: added alt="Pegasystems Logo" as an attribute to the logo image in PortalTitleBar.htm
2) **Give each frame a title**
   Example: added title="Navigation Pane" as an attribute to the left frame in frameset.htm

Priority 2

1) **Use public text identifier**
   This was problematic as we don't currently adhere to an HTML standard at my company because we can rely on Internet Explorer 6 being the browser since we sell to companies who use our software in their call centers rather than being generally available on the web. When I added something like <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Frameset//EN" "http://www.w3.org/TR/html4/frameset.dtd"> to the beginning of NavigationPane or PortalTitleBar it caused issues in the way the HTML was displayed. I am sure that these could be resolved by finding

another way to implement the formatting but it fell out of scope for this project. I did add this element to the files that I could (frameset, ShowWorkView, SpaceTitleBar, and WorkPoolClasses).

2) **Explicitly associate form controls and their labels**
   Example: added label element for text near the "Open in new window" checkbox
   ```
   <input type="checkbox" id="NWin" style="border:0px" value="ON" />
    <LABEL FOR="NWin">Open in new window</LABEL>
   ```
   This had the added benefit of making it so a user can click the text to change the value of the checkbox. This could be useful since it is a larger target than the checkbox itself.
3) **Make sure event handlers don't require a mouse**
   See additional information on this below. Rather than remove the mouse events, I added additional functionality to ensure that the keyboard was also supported.
4) **Include a document TITLE**
   Example: added <title>Project 3 for Karen Zelevinsky</title> to frameset.htm
5) **Provide a NOFRAMES section**
   Added <NOFRAMES>Please upgrade to a browser that supports frames in order to view Project 3</NOFRAMES> to frameset.htm. Ideally the NOFRAMES element would contain an alternate form of the data instead of just an error message, but that doesn't make sense either for my company's application or for this project.
6) **Use relative sizing and positioning rather than absolute**
   Since we are using style sheets that set size perhaps the best solution would be to allow users to supply their own style sheets so they could select the font, size, and colors that work best for them. This fell out of scope for this project.

Priority 3

1) **Identify language of the text**
   Added <html lang="en"> to all the HTML files
2) **Provide a summary for tables**
   Added summary="List of your Work" to the table in ShowWorkView. All the other tables are used for formatting purposes so should not have a summary. I did verify that the tables could be linearized (read in order as they appear as if the table didn't exist) so screenreaders shouldn't have a problem with them, but it would be better if we used other ways to align text in the future.
3) **Separate adjacent links with more than whitespace**
   Added | in between the links for Help, Launch, and Log Off in PortalTitleBar
4) **Include default, place-holding characters in edit boxes**
   Added value="Put ID here" for the Find by ID in NavigationPane

Next I tackled the issues with keyboard accessibility.

1) **Select elements with onchange handlers**

I created a js file called accessibleonchange.js which looks like this:

```
// If the select has really changed, do some action
function selectChanged(theSelect)
{
    if (theSelect.changed == "false")
    {
```

```
                return false;
        }
        alert(theSelect.value);
        theSelect.options[0].selected = true;
        return true;
}

// If user clicks with the mouse, set changed to true
function selectClicked(theSelect)
{
        theSelect.changed = "true";
}

// When select gets focus, save the initial value
function selectFocused(theSelect)
{
        theSelect.initValue = theSelect.value;
        return true;
}

// Determine whether onchange should be called when a key is pressed
function selectKeyed(theSelect)
{
        var theEvent = event;

        var keyCodeTab = "9";
        var keyCodeEnter = "13";
        var keyCodeEsc = "27";

        // If the user clicked enter or tabbed off after changing the value, fire onchange
        if (theEvent.keyCode == keyCodeEnter || (theEvent.keyCode == keyCodeTab && theSelect.value !=
theSelect.initValue) )
        {
                theSelect.changed = "true";
                selectChanged(theSelect);
        }
        else if (theEvent.keyCode == keyCodeEsc) // Reset if user hits escape
        {
                theSelect.value = theSelect.initValue;
        }
        else // Do not fire onchange yet
        {
                theSelect.changed = "false";
        }

        return true;
}
```

Then I updated the select elements so instead of just calling a function for onchange, they handle other events as well:

```
<select id="StartProcessSelect" onfocus="selectFocused(this)" onchange="selectChanged(this)"
onkeydown="selectKeyed(this)" onclick="selectClicked(this)">
```

The result is that a user with a mouse can click an option to cause an event and a keyboard user can view and browse the options without triggering any event and then can either hit enter on the option they want or simply tab off the dropdown (this only happens if the value was changed from the original value).

2) **Table of work with onclick handler –** In ShowWorkView, I picked the most unique and critical column of the work table (the work ID) and made each entry into an anchor. This way the user can use tab to highlight each row of the table and screenreaders will have more meaningful text for the anchor. I updated the style sheet so that it wouldn't look like an anchor. Then I added code so that the row would look highlighted either if the mouse went over or if the anchor had focus. Hitting enter while focused on the anchor is the same as clicking the table. A simpler version of the code in question looks like this:

```
<html>
<head>
<style>
A.noAnchor {
	font-family: Tachoma, Verdana, Arial;
	color: #003366;
	text-decoration: none;

}

A.noAnchor:hover {
	font-family: Tachoma, Verdana, Arial;
	color: #003366;
	text-decoration: none;
}

.listTableRowStyleSelected {
  background: #FFCC99;
  color: black;
  background-color: #FFCC99;
  text-align: left;
  font-family: 'Tahoma', Tahoma, Arial, Verdana;
  padding-left: 5px;
  padding-right: 2px;
  cursor: hand;
  font-size: 8pt;
  padding: 2px;
}

.listTableRowStyle {
  background: white;
  color: black;
  background-color: white;
  text-align: left;
  font-family: 'Tahoma', Tahoma, Arial, Verdana;
  padding-left: 5px;
  padding-right: 2px;
  cursor: hand;
  font-size: 8pt;
  padding: 2px;
}
</style>

<script>
function openWork()
{

	alert("open work");
}
</script>
</head>
<body>
<table onclick="openWork()">
	<tr class="listTableRowStyle" onmouseover="this.className='listTableRowStyleSelected'"
onmouseout="this.className='listTableRowStyle'">
		<td>Data1</td>
		<td><a
href="#none"onfocus="this.parentElement.parentElement.className='listTableRowStyleSelected'"
onfocusout="this.parentElement.parentElement.className='listTableRowStyle'" class=”noAnchor”>Data2</a></td>
		<td>Data3</td>

	</tr>
</table>
</body>
</html>
```

The last thing I did was to add accesskeys so enable users to easily jump from one part of the screen to another. This would probably be a usability enhancement to all users. This is done by adding an accesskey attribute to a form element (example, accesskey=”F”).

Alt H – Starts help (note that it takes precedence over the browser’s Alt H). Since this is an anchor and so would normally just get focus rather than triggering any event and I wanted it to fire the help immediately, I added a hidden button with the accesskey that also calls the doHelp function in PortalTitleBar.

Alt F – Goes to Find

Alt W – Goes to first item on the work list

Conclusions

The before and after HTMLs are on the web. Note that these only work correctly in Internet Explorer so please do not use other browsers.

Before: http://karen.zelevinsky.googlepages.com/framesetb.htm

After: http://karen.zelevinsky.googlepages.com/frameset.htm

After all my changes, I reran WebXACT and here are the results (the numbers in parentheses show the change from the baseline analysis):

After Changes – Number of Errors per HTML

	Priority 1	Priority 2	Priority 3
Frameset	0 (-1)	1 (-2)	0 (-1)
NavigationPane	0 (-1)	3 (-1)	1 (-2)
WorkPoolClasses	0	1 (-1)	1 (-1)
PortalTitleBar	0 (-1)	2	1 (-2)
SpaceTitleBar	0	0 (-3)	1 (-1)
ShowWorkView	0 (-1)	1 (-2)	0 (-2)
Total	**0 (-4)**	**8 (-9)**	**4 (-9)**

The only valid errors remaining are for using a public text identifier and for using relative sizing and positioning. The other errors are for making sure event handlers don't require a mouse (fixed manually by adding extra code to support the keyboard), defaulting place-holding characters (showing as an error for WorkPoolClasses but shouldn't be as far as I can tell), and providing a summary for tables (since the tables in question are used for layout it would actually be incorrect to supply a summary).

Given how much time I spent just to make a single simple screen accessible, this exercise has made me realize how time-consuming and challenging it would be to make our entire application accessible. However I do think that we will need to do it because we have customers who need to consider Section 508 compatibility. This will be especially difficult for us to support for the "developer" user. We had previously thought that we would just have to support accessibility for end users and managers since our customers have thousands of those types of users and only a handful of developers. However my research on Section 508 revealed that since it applies to *all* federal software, the excuse that there are not many developers will not be sufficient. I applaud this inclusive attitude for society and my company will just have to make the commitment to fix this problem now and going forward. I think it will be good for us in the long-term since making an application accessible often improves the experience for a broad range of users and not just those with disabilities. We have another goal to support browsers other than IE and following HTML and accessibility standards will help move us in the right direction there too.