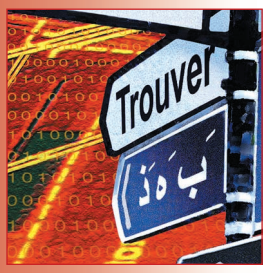

Achieving Universal Usability by Designing for Change

Partitioning function-, language-, and platform-specific elements of an interface into pieces that can be incorporated into templates lets developers adapt to requirement changes and provides universal usability.

Gary Perlman
OCLC Online Computer Library Center

When people discuss usability, they usually mean it narrowly – usable by a target market – but the Web is available to everyone with the economic means for a computer and a modem. While that still does not include most people, it does include users of different browsers on different operating systems, users who speak different languages, who have different physical capabilities, and who have different levels of experience with the Web. A universally usable interface is able to adapt to all these differences, and more, as they arise.

At the OCLC Online Computer Library Center, pent-up demand for new features in our Web-based FirstSearch bibliographic and full-text retrieval system (firstsearch.oclc.org) produced more requirements than we could effectively implement and scale. Moreover, many of the requirements were underspecified, so we anticipated change during review. In response to this, we used a methodology to enable us to easily adapt design decisions to meet new usability requirements. In our methodology, developers represent and implement design decisions in a single place so that any changes are immediately propagated system-wide.

We partition function-, platform-, and language-specific decisions into semi-structured information structures that we insert into templates to build HTML pages dynamically. This partitioning lets many types of contributors make frequent changes independently. As this article shows, the system adapts easily to changing requirements for user interfaces, new languages, and accessibility. Because we used the same methodology to address environmental variables, the universal usability design supports ongoing adaptations to the search sys-

tem at a generally lower cost than adapting to each variable in isolation.

System Requirements

In 1997, OCLC began developing a new version of the FirstSearch system, which is used by more than 15,000 libraries worldwide. The difficulty of adding our new features to the old system, and the relative liberation of a more dynamic Java development environment, led to what Fred Brooks might call a *second-system effect*: We spent about a year developing the detailed requirements and trying to incorporate every feature anyone wanted to add.[1] Design options were unclear at that point because we were charting many new territories, including

- new functionality,
- a new Web server and search engine (OCLC SiteSearch, which was being developed in parallel with FirstSearch requirements),
- a new programming language (Java), and
- a new version of the operating system on new hardware, including a new high-performance file system (later discarded for performance problems).

By early 1998, we had defined several general requirements for the new user interface. Although we had devoted considerable effort to the detailed requirements, we expressed cross-platform, text-only, multilingual, and accessibility and help requirements with little more than a sentence each. This lack of detail concerned us that different people would have different ideas about the right way to proceed in each, and that there would be pressure to be able to make changes, sometimes large.

Platform Independence

As part of the multiplatform requirement, the system needed to work on all current browsers. We proposed supporting the 4.x versions of Netscape Navigator and Microsoft Internet Explorer (MSIE). Initially, we wanted to require JavaScript and cascading style sheets (CSS), but Navigator 4.0 had limited support for features we could provide by other means. Realizing that many sites could not or would not upgrade their browsers, we ultimately chose to support the 3.x versions of Navigator and Explorer, although their limited functionality required some compromises. We did not want to support the *least-common denominator* but rather to take best advantage of each platform's features. We also wanted to be able to work around the bugs on different versions of the browsers.

We committed to supporting systems with JavaScript enabled, disabled, or missing. We also agreed to support different screen sizes across different hardware and operating systems, including Windows and Macintosh, and to test 256-color screens and for grayscale contrast. In addition to the graphical user-interface browsers, we also needed to support a text-only version of the search system that could run over telnet, which provided the only means of access for a small number of high-frequency users. We thus chose to replace FirstSearch's existing telnet version with an interface that worked reasonably well with the text-based Lynx HTML browser that would run on our server.

Multiple Language Support

We planned to support the interface and online help in three languages initially: English, French, and Spanish. We hoped this would be easier than our multilingual effort had been in the old Web-based version, which was not designed with translation in mind (so the same strings, such as Search, appeared in hundreds of places).[2] We had to identify all the places where search was used as an action, as a label, etc., and translate accordingly.

Accessibility and Levels of Users

We had listed complying with the Americans with Disabilities Act (ADA) as a requirement, although we had no knowledge of what that entailed — not least because there were no defined standards at the time. We had thought the text-only Lynx interface would serve the purpose, but later found that while text-screen readers serve some users, many others employ specially adapted graphical browsers. In addition to making the system useful for users with varying abilities and needs, we planned multiple search modes to support users with different levels of expertise, and to allow library and user customization.

Group Coordination Issues

As we began the process of implementing the requirements we had gathered, we assigned different groups primary responsibility for the different dimensions of the user interface:

- *marketing*, for requirements and terminology,
- *development*, for functionality,
- *database*, for loading new databases,
- *graphic design*, for icons, fonts, colors, and layout,
- *usability*, for interaction design and redesign, and

```
[expert]
pagename      = expert
pagetitle     = &Lang.pagetitle.expert;
pagelabel     = &Lang.pagelabel.expert;
tips          = &Lang.tips.expert;
status        = &Lang.status.expert;
controls      =
   &Style.dbinfo.gadget;
   &Style.scanindex.gadget;
   &Style.thesaurus.gadget;
   &Style.news.gadget;
action        = QUERY?searchtype=expert
term          = termexpert
index         = indexexpert
focus         = termexpert
panel         =
   &Style.dialog.begin;
       &Pages.basic.submit;
       &Pages.expert.searchbox;
       &Pages.expert.index;
       &Pages.advanced.limits;
       &Pages.advanced.options;
       &Pages.basic.submit;
   &Style.dialog.end;
searchbox     =
   &Style.dialog.rowbegin;
       &Style.font.labelbegin;
           <label for=termexpert>
               &Lang.label.find;
           </label>
       &Style.font.labelend;
       &Style.dialog.elementbegin;
           <textarea name=termexpert id=termexpert>
               &termexpert;
           </textarea>
       &Style.dialog.elementend;
   &Style.dialog.rowend;
```

Figure 1. Expert search page object specification of the [expert] section in `Pages.ini`*, the configuration file for the functional partition. Entities are often defined in terms of other entities for reuse and consistency. Language-specific terms are defined in the language files, and platform-specific styles are defined in the* `Style.ini` *file.*

- *documentation* (including translation), for onscreen help, and online and printed help.

Of course, this is an oversimplification because many groups contributed to multiple dimensions, but few individuals considered all these concerns when working on specific tasks. One of our coordination goals was therefore to make sure that all elements worked well together. We had to ensure, for example, that:

- nonportable platform-specific HTML or language-specific terms were not used in Java code or database configuration files,
- graphic or user-interface designs worked on all platforms, particularly if they used JavaScript, and
- terminology was used consistently in all parts of the system (including help) and that the terminology physically fit in the space allocated in all languages on all platforms.

All these were required to allow us to be able to make changes easily and in one place so they would change all parts of the system.

Design Approach

Because of the multiple dimensions involved, we needed a generalized approach to ensuring universal access to FirstSearch.[6] Our highest priority goal was to be able to adapt the user interface to the inevitable requests for changes that would arise from future usability, performance, and functionality considerations. With so many unknowns, the interface design had to be incrementally scalable, starting with a simple model that we could expand as we understood more. Previous experience suggested that partitioning the information in the system into orthogonal information sets and generating screens by forming the cross-product of those sets would facilitate incremental elaboration and optimal redesign.[3] We chose to partition the information into three main sets:

- functional (the specific functions for database selection, search, and results);
- platform-dependent (aspects that adapt the display to different platforms); and
- language-dependent (aspects used for translation).

The partitioning method is similar to a word-processing mail merge, but rather than inserting address information into letter and label templates, we insert function attributes into platform-specific templates for Web pages.

Functional Partition

As the first step in developing FirstSeach's user interface, we applied information design to identify key attributes of FirstSearch pages. A page is an object with the information (attributes and methods) used to construct what a user observes and does on a single Web page. The canonical sequence of pages in a FirstSearch session is database-selection, search, and results. Initially, we were not concerned with details about specific pages in the system; instead, we wanted to identify general elements that would affect new pages as well as existing pages that were to be merged with others or deleted. Similarly, we weren't over-

ly concerned with specific page *attributes* because we wanted it to be easy for us to add, delete, or change attributes.

In the end, we identified attributes common to every page in FirstSearch:

- *pagename*: an internal identifier,
- *pagetitle*: a title displayed to users,
- *pagelabel*: a short phrase for links in menus,
- *tips*: on-screen help tips,
- *status*: on-screen status information,
- *controls*: page-specific controls,
- *action*: a form action, and
- *panel*: a main form panel.

Individual pages can also have any of about 10 other specialized attributes, for processing form elements in the panel, handling errors, and so on. Attributes are very flexible because they can contain constant text and any number of *entities*.

To make all pages platform- and language-independent, we extracted the platform-dependent parts into a style file and the language-dependent parts into a language file. We then replaced these extracted parts with entities defined in configuration files (called INI files). We placed the resulting platform- and language-independent page definitions in the `pages.ini` configuration file, which includes a section for each page. The resulting definition for the expert search page, for example, looks like the specification in Figure 1, which appears to users like Figure 2. For reuse and consistency, entities are often defined in terms of other entities. The expert panel uses the same submit buttons as the basic search screen (`&Pages.basic.submit;`), for example, and the same limits and options as the advanced search screen. Figure 3 shows the specification for a page that displays detailed records.

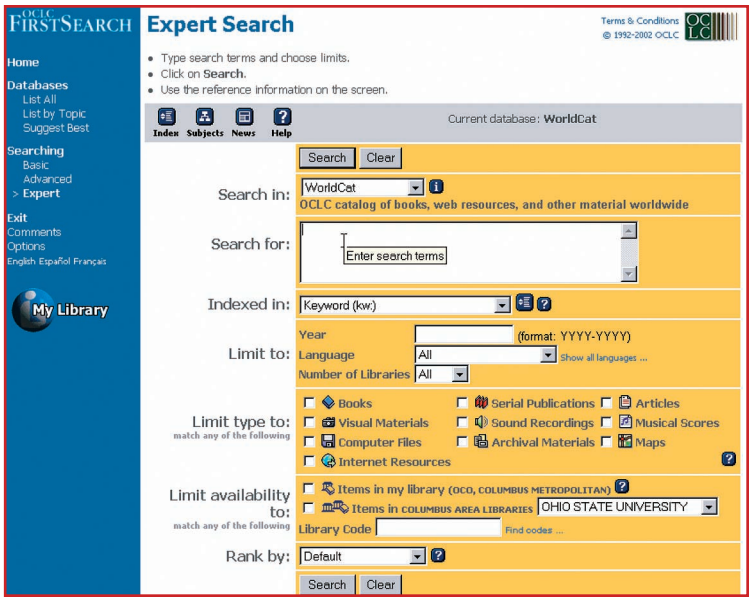

Figure 2. Expert search screen. The screen is constructed from the expert search page object specification based on the current database, user preferences, and account settings. Accessibility features show tips on input elements.

```
[record]
pagename    = record
pagetitle   = &Lang.pagetitle.record;
pagelabel   = &Lang.pagelabel.record;
tips        = &Lang.tips.record;
status      = &Lang.status.record;
controls    =
   &Style.thesaurus.gadget;
   &Style.ill.gadget;
   &Style.holdings.gadget;
   &Style.email.gadget;
   &Style.print.gadget;
action      = FETCH?fetchtype=record
panel       =
   &Style.dialog.begin;
      &Style.dbsuggest.gadget;
      &Style.navigate.gadget;
      &Style.record.gadget;
      &Style.navigate.gadget;
   &Style.dialog.end;
```

Figure 3. Detailed record page object. In this specification from Pages.ini, entities with names ending with "gadget" have values that invoke classes to generate HTML.

Entity Substitution and Dynamic HTML

In OCLC SiteSearch, variables are called entities. Entities can be scalar variables, values from configuration files, and calls to Java methods with access to the user's session information. Many of the attributes in INI files are references to language entities (such as `&Lang.tips.expert;`) defined in language files (one for each language). Some attributes are style entities (such as `&Style.dialog.begin/end;`) that mark the beginning and end of structurally meaningful parts. Other entities include references to parts of pages, which allow us to reuse modular definitions (for example, all search screens use the basic submit buttons defined in `&Pages.basic.submit;`). Special-purpose attributes indicate the names of terms and indexes used on search screens, where to focus the cursor if JavaScript is available, and so forth.

When FirstSearch displays a page, it inserts the associated attributes into a template for the page type. Current values of entities are substituted recur-

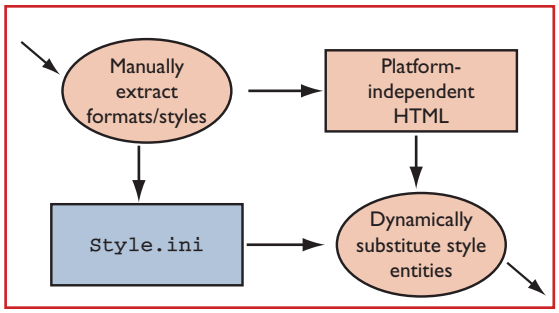

Figure 4. Partitioning platform-dependent information. Any platform-specific HTML is moved out of Java code, HTML, or language strings into style entities that are dynamically substituted when pages are generated.

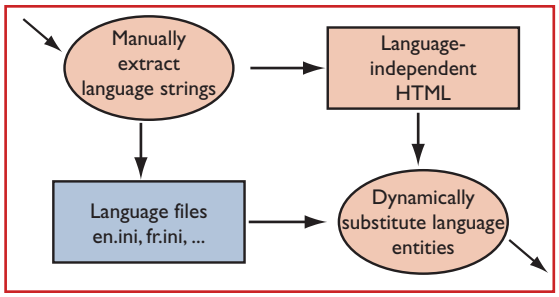

Figure 5. Partitioning language strings. Natural language-specific strings are moved out of Java code and HTML into language entities that are substituted when pages are generated.

sively. While we could have chosen many methods to generate dynamic displays and adapt to different platforms, we picked entity substitution as a matter of convenience. We considered using XML, CSS, and Java applets, but the SiteSearch Web server already provided many of the features we needed, whereas XML was untried, CSS was not supported by all browsers, and Java was unacceptable to many users because of security or performance concerns.

Although the use of proprietary SiteSearch code might seem to limit the methodology's generality, many (if not most) development environments can read configuration files and substitute variables. Recently, we added the capabilities to Tomcat servlet containers (jakarta.apache.org/tomcat) to move to a less proprietary environment.

Practical Partitioning

With the page object defined, the most difficult aspect of partitioning the language and style information was locating the information and then performing the process consistently. We found it nearly impossible to explain to developers why and how to keep this information separate, perhaps because it required an understanding of translation, cross-platform development, accessibility issues, and general usability considerations. Added to that, the terminology and formats were being developed in parallel, so developers had to develop more abstract methods.

Instead, we had developers create screens using "untamed" English and HTML and then partitioned the information for them. We extracted platform-specific parts and replaced them with style entities for runtime substitution based on the user's platform and preferences (Figure 4). To internationalize the design, we moved language-specific strings into the language files, replacing them in the templates with entities to create language-independent HTML. Later, we inserted the language-specific values into the HTML by substituting language entity values in the user's language (Figure 5). During the process, we could review the choice of language and interface design and, later, centralize design decisions so that these could be changed.

Platform-Dependent Partition

The page object contains references to platform-dependent parts, which will display differently on different platforms. Many factors affect how the interface design is presented to users, including browser type and version, operating system, screen size, and whether JavaScript is enabled. For example, if JavaScript is available, a pop-up Help window can open at a size that works with the screen dimensions and shows shortcut keys based on the operating system.

Many methods are available for adapting to different displays, ranging from using lowest-common denominator features to developing unique subsites for different displays or dynamically customized displays. Given the number of platforms planned for FirstSearch, and the many differences among them, dynamic generation of HTML was an obvious choice.

The method we chose was designed to abstractly represent display structure separately from the final rendering. For example, an untamed error message might be initially marked up as

```
<font color=red size=5><b>
   Something bad happened
</b></font>
```

Styles could be replaced by entities:

```
&ErrorBegin;
   Something bad happened
&ErrorEnd;
```

and defined in style entities:

```
[styles]
ErrorBegin = <font color=red size=5><b>
ErrorEnd   = </b></font>
```

A line in a search form might be marked up as:

```
&SearchFormBegin;
   ...
&SearchLineBegin;

      &LabelBegin;
         Find:
      &LabelEnd;
      &FormElementBegin;
         <input type=text name=terms>
      &FormElementEnd;
   &SearchLineEnd;
   ...
&SearchFormEnd;
```

We have simplified these examples to better explain the methods used in FirstSearch; the real versions have many gory details. Because entities are substituted into the outgoing HTML, changing the definition of structural entities changes the HTML that will be generated. On graphical browsers, the error message above might be made larger or smaller to match the screen size, or it might be preceded by an error icon and appear in a large red font. On a nongraphical browser such as Lynx, the error message could be bold and surrounded by lines.

The fine granularity of control and the likelihood of editorial changes made it undesirable to code these changes in Java. Instead, we adopted a declarative method of specifying custom values. When a session starts, FirstSearch stores all the potential customizing variables, such as browser attributes, in about 30 entities. The system then reads default entities (about 50) from an initializing configuration file and sets customizing entities based on values read from conditional sections in the style INI files.

Ordinary INI files contain named sections (such as `[styles]`) that list entity definitions. Conditional sections are named by entity-value pairs — the browser entity might be Mozilla, MSIE, or Lynx, for example — and conditional styles could be defined for each browser value, or for those that require special settings:

```
[styles]
section*   = browser
[browser=Lynx]
ErrorBegin = <h1><b>
ErrorEnd   = </b></h1>
[browser]
ErrorBegin = <font color=red size=5><b>
ErrorEnd   = </b></font>
```

The reference to `section*` causes the system to read the conditional section named `browser`. If the browser is Lynx, the system uses the section called `[browser=Lynx]`; otherwise, it uses the default `[browser]` section. We could also add other sets of conditional sections. If we wanted the error message font size to depend on the screen size, for example, we could insert an entity into the error message style and set the value of the entity in conditional sections:

```
[styles]
section*   = browser
section*   = screensize
[browser=Lynx]
ErrorBegin = <h1><b>
ErrorEnd   = </b></h1>
[browser]
ErrorBegin = <font color=red
size=&ErrorSize;><b>
ErrorEnd   = </b></font>
[screensize=large]
ErrorSize  = 5
[screensize=medium]
ErrorSize  = 4
[screensize]
ErrorSize  = 3
```

Once conditional sections are set up, we can easily add conditions to set more than 100 entities. Setting these in INI files lets us view the changes while the system is running. Users can thus reread the INI files and reset entities without changing any code or retracing steps to the current screen. Another advantage is that all the peculiarities of particular platforms are specified together. For example, the color scheme for MSIE 3 is different than for the rest of the system because that browser version does not support changing text color in a hot link.

Language Partition

We internationalized FirstSearch by moving all language-specific terms (about 5,000) into INI files and replacing those terms with entities that referred to the appropriate section and entity name. In FirstSearch language files, sections serve to distinguish how and where some text will be used. For example, all diagnostic messages reside

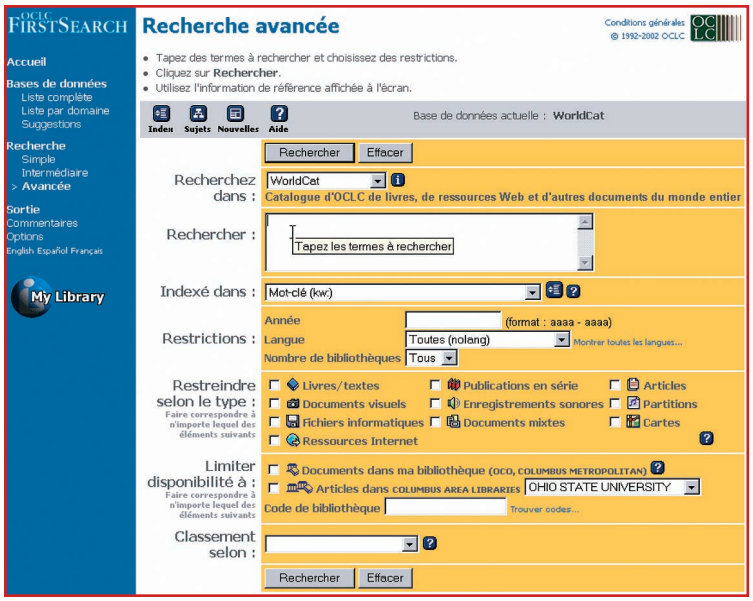

Figure 6. French version of the expert screen. Users can change language at the bottom of the left navigation menu. The only change in the system is that the language entities come from cached sections of the selected language file.

in a section called `[msg]`:

```
[msg]
bad    =  Something bad happened
nohits =  Your search matched no records
nojs   =  Your browser doesn't support
JavaScript
```

Users access entities in language files by naming the entity `(Lang)` followed by the section `(msg)` and the variable name `(bad)`. This error message's platform- and language-independent version therefore becomes

```
&ErrorBegin;
   &Lang.msg.bad;
&ErrorEnd;
```

When users choose new languages, FirstSearch associates entities in the appropriate language INI file with their session. Figure 6 shows the French version of the expert search interface.

Structure. Page title, tips, and status are all sections in the language file that contain variables defining the page title, on-screen help tips, and status for each page. Storing the variables in the same section makes it easier to keep the text consistent for different screens. Developers must, however, place page attributes in different sections of different INI files. To make it easier for the user-interface and database groups to work together, we separated the user-interface language INI file from a language file for database-specific terminology (which accounted for about two-thirds of the language used in the system). This approach reduced contention while both files went through hundreds of revisions.

English as the second language. Although we did the development in English, moving strings into the English language INI files for later translation, there was an initial translation step that took almost as long as the translation into Spanish and French. The initial language was a dialect of English used by librarians and developers of systems for searching library materials; call it Jargonese. Some of the terminology was inappropriate for library-naive users, all the more common because of the advent of the Web. Thus, a screen might be called "history" internally, but users would know it as "previous searches."

Finding entities and previewing translation. To help translators and documentation writers determine where an entity was defined, we created an entity language in which the value of an entity was the section and variable name where it was defined (for example, `&Lang.pagetitle.history;` would be displayed as pagetitle~history). Then they would know that the string displayed in English as "previous searches" was the `history` variable in the `pagetitle` section. This method also allows page name changes to be propagated automatically throughout the system, including documentation.

We included a facility for dynamically reloading entity values on the current screen, which allowed translators to see their translations without starting a new session. Preview was important for ensuring that the edits fit and did not break any embedded HTML or entities. Marketing staff even used the mechanism as they replaced Jargonese with English.

Adding non-Western languages. Because all the natural language in the system is localized, our current efforts to translate FirstSearch into Chinese (two character sets) and Japanese using Unicode (UTF-8 encoding) are reasonably straightforward. The only change we have had to make to the system is to include `charset=UTF-8` in the HTTP header when displaying a non-Western interface.

Template-Based Page Generation

We based the first prototype systems on ad hoc flat-file databases accessed with Perl scripts and used the semistructured toolkit to generate HTML

files.[4] Eventually, the information about pages migrated into semistructured INI files, and we moved from static to dynamically generated HTML files. Keeping the pages and attributes in an easily editable format was an important feature as they evolved and gradually increased in complexity over time. The ease with which we were able to change the software highlights the independence of the methods we used.

FirstSearch generates HTML pages by inserting page-specific entities into templates like the one shown in Figure 7. We have created a graphical-browser version, a text-only Lynx version, a printer-friendly version, and others. Templates can (and perhaps should) start as simple renderings of some attributes, but they can be augmented easily for scaling and major changes. To move the controls for all pages, for example, we only need to move one line.

Many changes are unanticipated, so the flexibility of being able to make global changes is highly desirable. Initial versions of the FirstSearch interface were framed, for instance, but we decided to evaluate an unframed version because of server transaction costs. Creating an unframed version of the interface took about an hour, and the performance results motivated us to change the entire system — which took one person less than a day to do.

The specification in Figure 7 is a simplified template for the Lynx interface, which is simpler than the graphical version. Note that most page attributes have been assigned to entities. Figure 8 shows the user's view of the screen. This Lynx template is one of many possible renderings of the parts of pages. One advantage the interface's framed version offered was that the main frame contained all and only the information that users would want to print. When the framing was removed, it took about an hour to create a template for a printer-friendly format that did not show menus and controls.

Because templates can be defined hierarchically, they can share reusable parts. This can minimize the cost of creating new versions of templates, say, for a version that takes full advantage of CSS. At various times, we have designed, created, and viewed completely different interface designs that were fully functional systems. Small changes, such as placing an entity value on every page in the system, require a one-line change. When quality assurance staff wanted to add specially formatted comments to delimit logical sections of the screens (to help highlight differences in regression test scripts), the change took less than an hour. Figure 9 (next page) shows a combined view of dynamic HTML page generation by inserting entities into templates.

```
<html pagename="&pagename;">
<head>
   <title>&pagetitle;</title>
</head>
<body>
   &pagetips;
   &pagestatus;
   <form method="POST" action="&pageaction;">
      &pagepanel;
      &pagecontrols;
      &Style.Menu.gadget;
   </form>
</body>
</html>
```

Figure 7. Template for Lynx displays. FirstSearch displays HTML pages by substituting entity values, values of values, and so forth — sometimes to 12 levels deep — into templates like this one. Even the template, itself, is an entity (&Style.template.lynx;).

```
Expert Search
Current database: WorldCat

Type search terms and choose limits.
Click on Search.

[Search] [Clear]
_____________________________________
_____________________________________
_____________________________________
_____________________________________
_____________________________________
_____________________________________
Indexed in: [Keyword (kw:)__________]
Limit to:
 Year          1990-_______
 Language      [English___________]
 Libraries     [All___]
 Document Type [Books_________]
 Library Code  ____________
 [_] Items in my library (OCO)
Rank by: [Default___________]
[Search] [Clear]

[index] [subjects] [news] [help]
```

Figure 8. Expert screen for Lynx text-browser users. The navigation menu (not shown) is appended to the display. The Lynx layout of controls is adapted to better match the linear access to form elements.

Accessibility Issues

Initially, we thought that the text-only Lynx version would be the best platform for a sight-impaired user's screen reader. After interviewing one of our staff, who is blind and uses Web-aware HTML-reading software, we broadened our approach to include all browsers. For the Lynx version, we used a simpler vertical page template, took full advantage of the fact that Lynx inter-

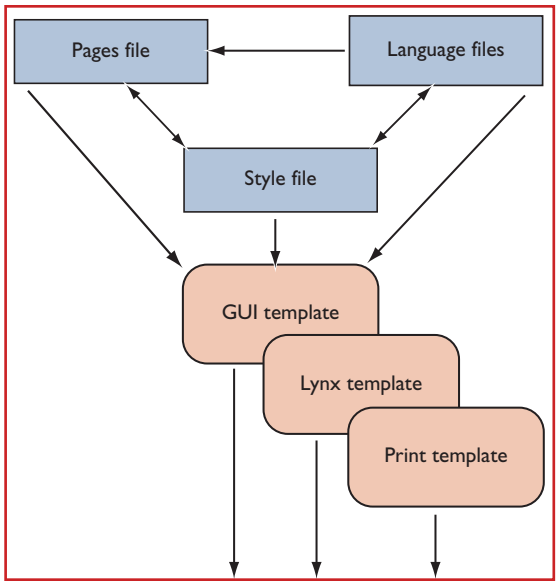

Figure 9. HTML dynamic page generation integrates functional page information, language values, and styles into platform-specific templates (GUI, Lynx, and Print, for example). Any template can include function-specific entities from `Pages.ini`, `Style.ini`, *or the current language. These can, in turn, include other entities. This reuse of entities allows reuse and helps enforce consistency where desired.*

preted the table row tag <TR> as starting a new line, and set Lynx-only format entities (such as space, bar, break, line, paragraph, and comment) in a conditional INI file section.

Because the HTML for formatting the display is localized in style files, we can make most changes centrally to adapt to the Web Accessibility Initiative (WAI) guidelines (and later, U.S. Section 508 rules). Microsoft's Internet Explorer 4+ provides substantial support for accessibility-oriented tags, including some features that are useful for all users:

- *Title*. The title attribute provides extra information about the element it is attached to. FirstSearch uses title tags for input fields to provide more detailed prompts, and on links to explain where they will lead the user. (See the text areas in Figures 2, 6, and 8.)
- *Label*. The label tag allows a label to be more formally associated with a form element with which it is logically associated. Web screen readers know that a label is associated with a checkbox, for example, and MSIE 4+ lets users control form elements by clicking on their labels. See Figure 1 (page 48).
- *Accesskey*. The accesskey attribute allows us to associate `Alt-x` shortcuts with form elements. FirstSearch associates Alt-s with submit buttons, for instance, and Alt-c with the clear button.

Although these attributes and tags are not supported in the 3.x versions of the major browsers and Netscape 4.x, they do no harm. In early versions of Netscape 6, label tags produced display problems, so they were removed using conditional entities.

Levels of Users

We designed FirstSearch for different levels of users with three search levels: basic, advanced, and expert. The levels differed in the number and size of search boxes, the number of indexes (3 for basic, 10-15 for advanced, and 20-30 for expert), and the detail and location of context-sensitive help. These adaptations were designed and implemented using the same methods used for other dimensions of user variability.

Customization

The FirstSearch administrative module lets libraries customize FirstSearch by choosing default search modes, topic areas, library logo, links into library catalogs, and most options for controlling the access to for-fee items. We are also exploring patron customization of the interface by saving settings across sessions. We implemented patron settings in a few hours because the user interface allows setting groups of entities. The same framework is flexible enough for us to explore gender and age-based customization.

Coordination Issues

The partitioning of the user interface, and the plain text format of the interface initialization files, let nondevelopers make changes to the system without involving programmers. For the first time, nonprogrammers had interactive control over the parts of the system for which they had responsibility, and it took many user-interface decisions out of the hands of programmers (which was generally received positively by all).

Most contributors could not follow detailed instructions about how to develop platform- and language-independent screens. A few guidelines proved to be more effective (such as, "no HTML in Java code"). For practical purposes, most programmers found it easier to write untamed code and partition it when it was ready. As we identified problems, we enhanced checking scripts to find problems automatically.

Conclusion

We designed the partitioned user interface primarily so that it could be adapted to changing requirements. In achieving that goal, we allowed the rapid exploration and implementation of a variety of universal usability dimensions: cross-platform, multilingual, accessible, and, in general, environment-sensitive versions. With conditional sections of initialization files, we could develop parameters to adapt to the presence of JavaScript, the quirky performance of certain browsers (old and new), and custom parameters for different screen sizes, among other considerations. The performance costs for late binding and dynamic generation of HTML have been small, and in some ways have improved performance because generated pages do not require any file access.

As we have gathered feedback and done more usability testing, we have made changes to the system. Returning to the highest priority goal of adaptability, it was not critical to get the design right, but it was critical to be able to change what was wrong. Partitioning the design into independent single-source representations made it easy to know where and how to make global and local changes. Global changes to reorganize all screens took minutes or hours of editing a single template instead of days or longer. Small changes have had invariably low costs, and larger changes have had generally proportional costs (although they sometimes have multiplicative benefits when applied to templates because templates apply to many pages). New pages could be added independent of other pages, but follow the same global conventions and appear immediately in all languages. System-wide styles could be changed independent of the system functionality and terminology and tuned to fine-grained details of particular browsers, platforms, and so forth. Terminology, including new language translations, could be added and changed independent of the rest of the system.

One drawback of the methodology is that it requires that developers work more formally when making changes, understanding where and how changes should be made (perhaps this is a benefit and not a drawback). But change is not without cost. Even when it was easy to make changes, we had to factor in the costs of regression testing (which checks for differences in the HTML) and of training (which in most cases is done by individual users for themselves).

The methodology repeatedly let us adapt to change, leading us to conclude that it was more important to be able to change than to "get things right." Usability testing, user feedback, new browser releases, new features, and new application layers would continually require change. What is perhaps the most striking result of designing a user interface capable of adapting to change is how it helped with areas for which we did not anticipate change. We knew the screen layout would change, and we knew the terminology would change, but we did not know we would be using label tags and title attributes for accessibility, nor that the same methods to adapt to platforms could be used for user-customizable versions of the system. The demands of a few universal access issues required a design that helped address many.

Acknowledgments

I would like to thank Mike Prasse, the reviewers, and the editors for their comments. I am eternally grateful to James Fitch for providing the ability to re-read updated INI files in context without restarting, and I would like to thank Allan Schreiber for his inspiration on generalization.

References

1. F.P. Brooks, *The Mythical Man-month: Essays on Software Engineering*, Addison-Wesley, Reading, Mass., 1975.
2. D. Hysell and G. Perlman, "Lessons Learned from Internationalizing a Global Resource," *Proc. 1st Int'l Workshop Internationalization of Products and Systems (IWIPS99)*, G. Prabhu and E. delGaldo, eds., Backhouse Press, Rochester, N.Y., 1999, pp. 183–192.
3. G. Perlman, "Coordinating Consistency of User Interfaces, Code, Online Help, and Documentation" *Coordinating User Interfaces for Consistency*, J. Nielsen, ed., Academic Press, San Diego, Calif., 1989, pp. 35–55.
4. G. Perlman, "Information Retrieval Techniques for Hypertext in the Semi-Structured Toolkit," *Proc. ACM Hypertext 93*, ACM Press, New York, 1993, pp. 260–267.
5. G. Perlman, "The FirstSearch User Interface Architecture: Universal Access for Any User, in Many Languages, on Any Platform," *Proc. 1st ACM Conf. Universal Usability,* ACM Press, New York, 2000, pp. 1–8.
6. G. Perlman, "CHI 99 SIG: Universal Web Access: Delivering Services to Everyone," *SIGCHI Bull.*, vol. 31, no. 4, May 1999, pp. 53–54.

Gary Perlman is a consulting research scientist at the OCLC Online Computer Library Center where he works on user interfaces for bibliographic and full-text retrieval. His research interests are in making information technology more useful and usable for people. He is a member of the IEEE Computer Society, ACM, and Human Factors and Ergonomics Society.

Readers can contact the author at perlman@oclc.org.