A Software Architecture for Porting Interactive Applications to the Web

Brian R. Gaines and Mildred L G Shaw
Knowledge Science Institute, University of Calgary
gaines@cpsc.ucalgary.ca, mildred@cpsc.ucalgary.ca

Abstract

This article addresses porting interactive applications to operate in a client-server mode on the World Wide Web (the *web*). A software architecture is described that factors the application into its user interface module and an interface-independent service module. The user interface module is further factored into generic modules that can be written once as a shell, and implementation-specific modules that differentiate the stand-alone application from the web implementation. Relevant features of HTML as a graphic user interface programming language are analyzed, as are issues of storing the application state at the client or the server.

1 Introduction

Many interactive applications have been written to operate effectively on personal computers without access to networks. With the growth of access to networks there is often user interest in having these applications able to take advantage of a networked environment and:-

- run on a range of platforms
- be simple to maintain and update
- access data on other systems
- make data available to others
- integrate with other applications
- operate as groupware with multiple collaborating users.

Cross-platform user interfaces can be difficult to develop and maintain, and, if written as part of a stand-alone application, they require every user to run a specific application and to upgrade them whenever an update is issued. To avoid these problems, many applications are now being written to operate through the web and use a standard web client to provide their user interfaces.

The problem of porting an application to the web reduces to one of factoring the application into two modules:-

- an application client
  - a standard web browser
  - running locally
  - interfaced to the user through a local graphic user interface (GUI)
  - interfaced to the server through a network
- an application server
  - a special-purpose program
  - running remotely
  - no user interface
  - interfaced to the client through a network

This article is concerned with how to factor an application in this way. It describes a generic software architecture for doing this and exemplifies it through examples.

2 Factoring Interactive Applications

It has long been good software engineering practice to factor an interactive application into two distinct modules as shown Figure 1: a set of *application services* that have no user interface and are accessed through a well-defined application program interface (API); and a *user interface* that supports all the user interaction and communicates with the application services through the API. It will be assumed that the application to be ported is written in this way and the application services will be treated as a black box for porting purposes. It is the user interface module that requires detailed analysis in relation to porting to the web.

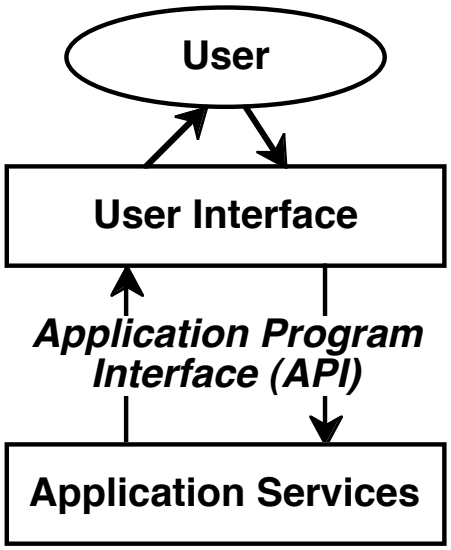

Figure 1 Factoring an interactive application

What is involved in porting the interactive application shown in Figure 1 to the web? The application services will now run on a remote server. If the server uses the same platform as the original application then little is involved in porting the application services. Otherwise, they may need recompilation for a different platform with some modifications to the way in which they access operating system services if these are different on the new platform. Since no user interface is involved and the application services appear as fairly pure code accessed through a well-defined API this generally requires little effort.

It is the user interface that has changed significantly as shown in Figure 2. The user now interacts with the application through a standard web client such as Netscape Navigator or Microsoft Internet Explorer. The client is platform-independent and runs in a standard way on the user's local workstation. It accesses the application through an Internet (or intranet) connection to a web server using the hypertext transfer protocol (HTTP). The web server may be special-purpose, for example HTTP protocol code running as an integrated part of the application, but generally it is one of the many standard web servers that provide access to hypertext markup language (HTML) documents and other files. One of the features of such servers is that they are capable of passing specified requests to auxiliary servers through a common gateway interface (CGI). The way in which such CGI requests are specified is not standardized but generally involves some identifiable feature in the uniform resource locator (URL) being sent to the server to request a document. It is possible for the web server and the auxiliary server to run on different platforms.

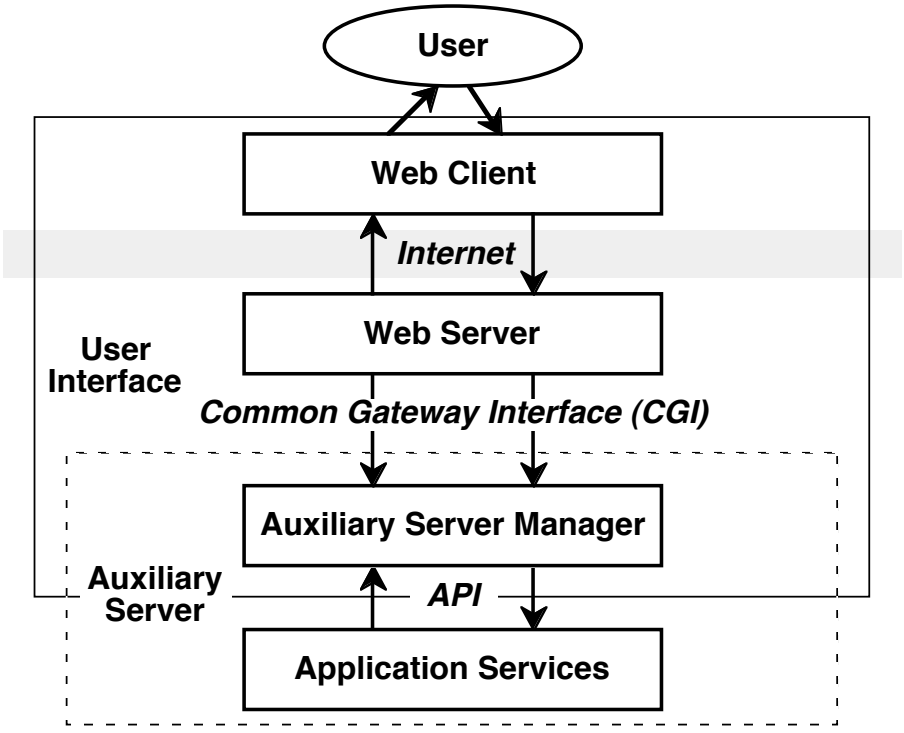

Figure 2 Factoring an interactive application to operate on the web

Note how two different factorings of the application are apparent in Figure 2. From a web technology perspective what is being designed is an auxiliary server consisting of an auxiliary server manager drawing upon application services. However, from an overall application design perspective it is important to recognize that what is being designed is a user interface partitioned into two modules: a web client which supports presentation to, and interaction with, the user; and an auxiliary server manager that generates that presentation and supports that interaction. From both perspectives it is the module termed 'auxiliary server manager' in Figure 2 that requires design, and this will be the focus of the remainder of this article. It will be assumed that the details of the HTTP, HTML, CGI and URL protocols can be gleaned from the wide range of books and web documents concerned with these protocols, and it is the architecture of the user interface in Figure 2 that requires detailed analysis.

3 Some Basic Issues

There are some basic issues in porting applications to the web that need to be addressed. First, the HTML protocol at level 2 and above provides a document-oriented graphic user interface with many capabilities of normal GUI's. However, the set of widgets available is limited and some of those used in the original application may not be available. One approach is to redesign the interface to use those available, and another is to use the advanced capabilities of current browsers to support special-purpose widgets that are downloaded in Java or supplied as plugins. In this article we shall focus on redesign to use those available. Second, the HTTP protocol is stateless and does not itself support the maintenance of the application state information at the server between transactions or the long-term storage of that state. One approach is to maintain the entire state at the client, and another is to do so at the server. In this article we shall focus on the first approach but discuss the issues involved in the second.

To exemplify the issues we will use the application RepGrid/KSS0 which we have ported to the web as WebGrid. RepGrid/KSS0 is a knowledge elicitation, sharing, comparison, modeling and management tool based on repertory grids that is highly interactive with extensive graphic output, and has been widely used in expert system development [2]. The WebGrid port offers all the functionality of RepGrid/KSS0 across the web. The application will not be described in detail since its operation and application is detailed in other publications [3, 5] and it is freely accessible on the web (http://gigi.cpsc.ucalgary.ca), but examples of the web implementation will be given to illustrate the significant issues discussed in this article.

3.1 HTML as a graphic user interface programming language

The hypertext markup language as originally designed to support typographically-styled documents with embedded hypertext links already provides a primitive programming language for document-oriented graphic users interfaces. When HTML was extended to include clickable graphic images and forms allowing user interaction through typical GUI widgets it became a very powerful GUI programming language. The widgets include:

- Single and multi-line text entry boxes returning the text entered
- Radio and check boxes returning single and multiple selections, respectively
- Popup menus and selection lists returning single and multiple selections, respectively
- Clickable images returning the coordinates clicked
- Hidden data encoding the transaction state, not shown to the user but returned to the server.

These, together with normal HTML text, rules and images, allow a wide range of user interfaces to be encoded that, because they use the same widgets, can be made to resemble forms and dialog boxes in conventional applications.

The active document metaphor is a simple one for users to understand and the automatic layout of text with embedded widgets provides flexible interface management that is simple for the designer to understand. Later features of HTML beyond level 2 have essentially given the designer greater control of the layout through support of tables and frames, and additional features such as JavaScript have added support for local interactivity at the client.

However, a highly interactive application that has been customized for ease of use will likely use widgets that are not provided in HTML. For example, Figure 3 shows a RepGrid/KSS0 screen which allows users to rate entities on a rating scale by dragging the entity name to a rating bar. This interface cannot be emulated directly in HTML. However, WebGrid achieves the same functionality by using a popup menu in HTML as shown in Figure 4. In informal comparisons some users prefer the original interface and others the web one, but both are completely functional.

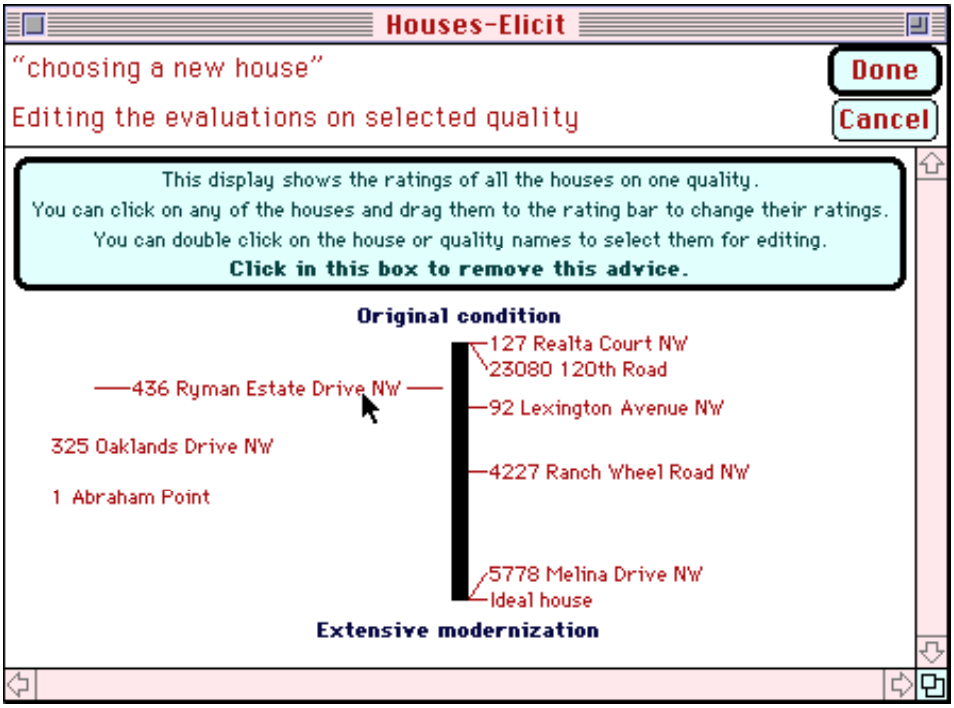

Figure 3 RepGrid/KSS0 click and drag rating scale interface

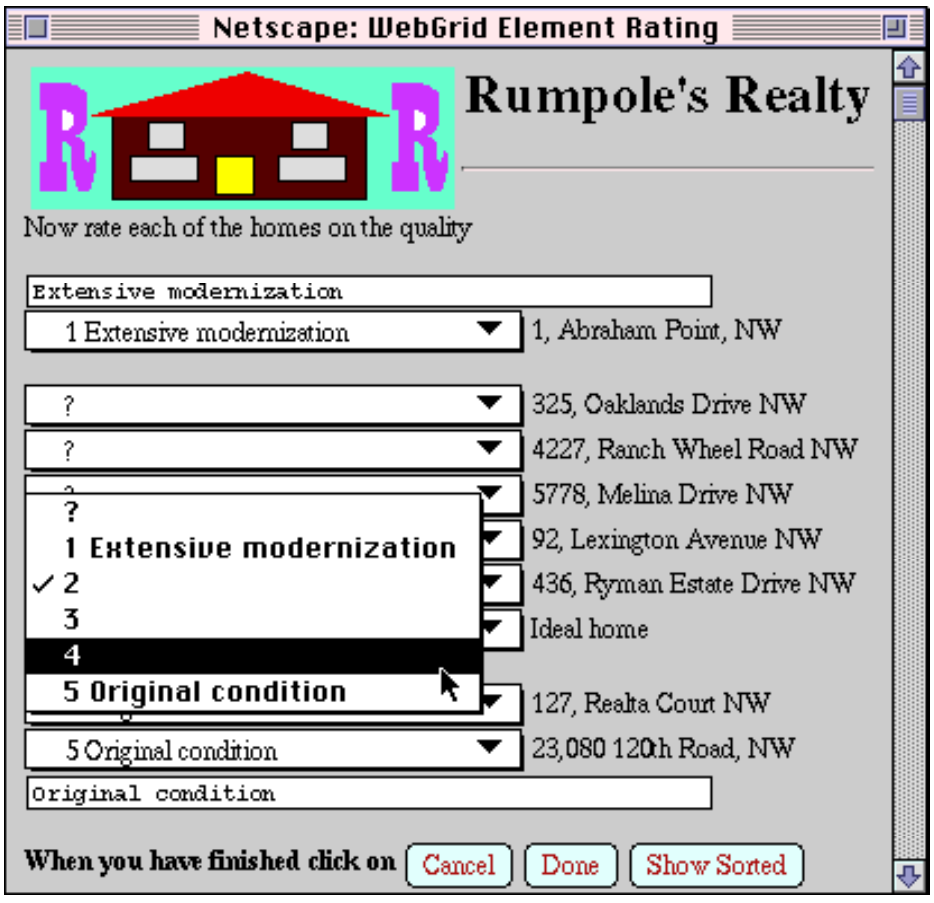

Figure 4 WebGrid click and drag rating scale interface

The type of GUI features that HTML does not provide are those involving the development of interactive graphic objects such as those of our concept mapping tool, KMap, shown in Figure 5. This can be emulated on the web using downloaded Java code [3], but such techniques will not be covered in this article.

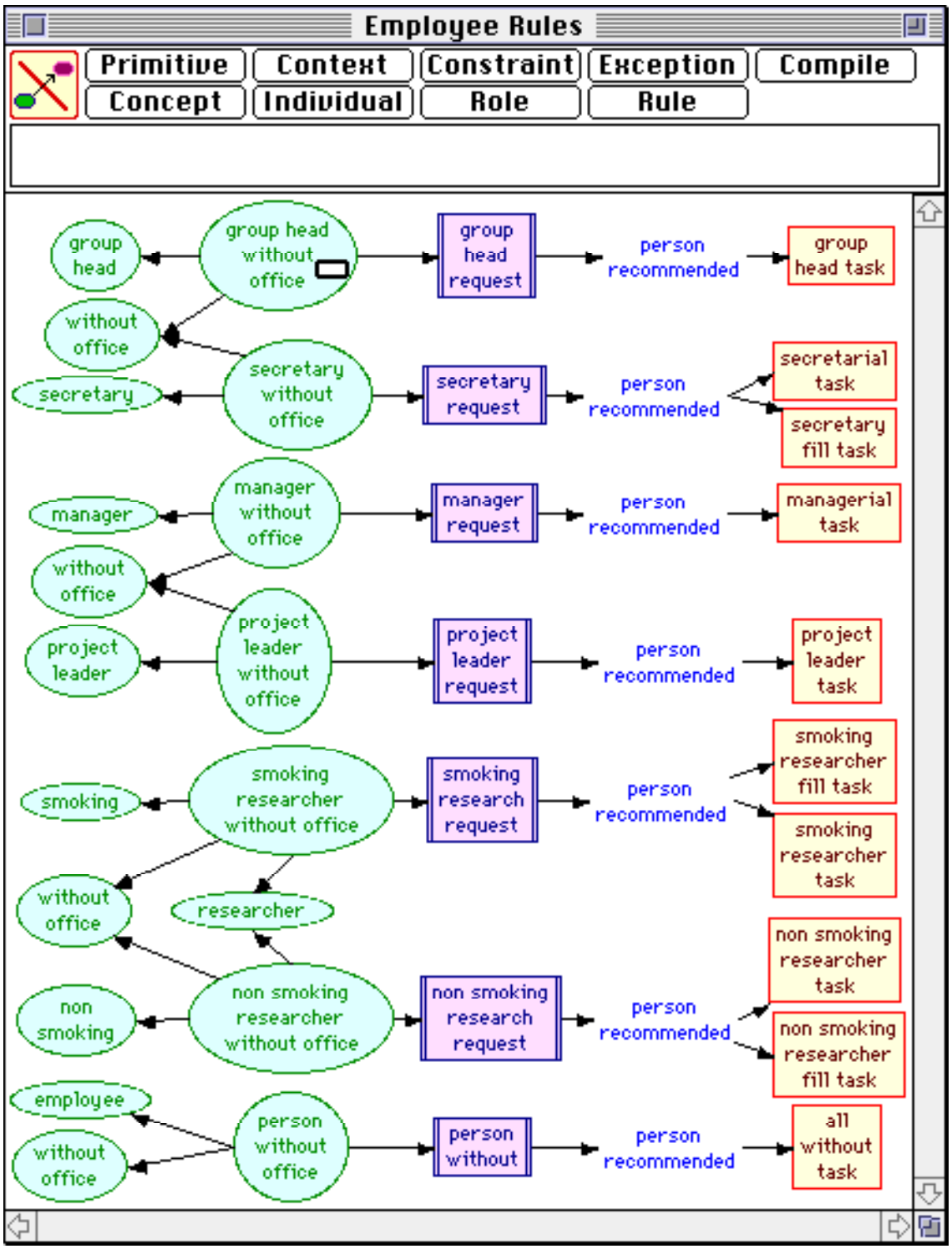

Figure 5 An interactive graphic user interface that cannot be emulated in HTML alone

3.2 Reconceptualizing user interfaces as documents

Even though the user interface shown in Figures 4 appears much like one in a stand-alone application, it is important in designing such interfaces to conceptualize them as *documents* [1]. In particular, they will wrap if the user resizes the screen. It is difficult to design a form interface that looks good under all conditions of wrap. A good strategy is to design for the default window width, not be too overly concerned about narrower windows, and to use line break tag,
, judiciously so that the layout is effective in wider windows.

Figure 6 shows the main status screen from WebGrid to illustrate some of the issues involved in conceptualizing an interactive interface as a document.

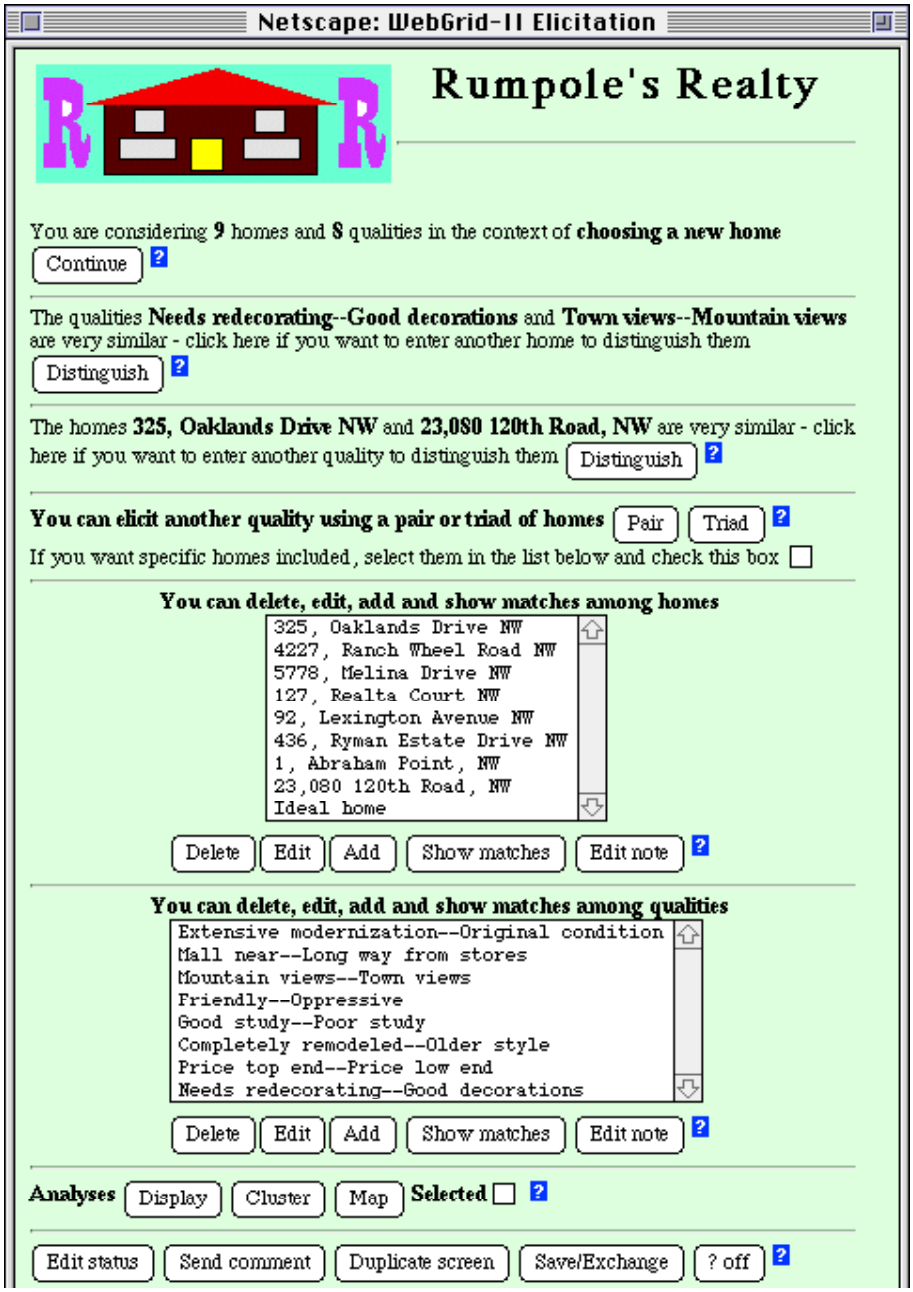

Figure 6 The GUI as a document—main status screen in WebGrid

The main design principles involved are::

- A major part of the functionality has been made available in a non-modal way so that the user does not have to fetch additional HTML documents from the server just to select functionality
- The main features of the current state of the application are displayed so that the user is able to make decisions about the next transaction without having to fetch additional documents
- The various functions available are split into small modules separated by horizontal rules so that the user can keep a complete sub-context in view even if they have to scroll through the document
- The various functions are being generated from the current state of the application in such a way as to be as helpful as possible in making clear the options available
- Some of the functions are optional and will not appear unless appropriate—there is no fixed layout
- Context-sensitive on-line help in a separate window is provided through the "?" icons.

RepGrid/KSS0 performs various data analyses which it can return in graphic form. WebGrid emulates this functionality by converting the graphic to GIF format and embedding the resultant image in the document returned. Figure 7 shows the document returned when the "Map" button is clicked near the bottom of Figure 6 to request a principal components analysis of the repertory grid data. The Compuserve GIF format is information lossless and will reproduce pixel for pixel the exact form of the original graphic, and is the preferred format for most graphic output. It is restricted to a 256-color palette, but this is the most common color capability at browsers. The GIF format uses Lempel-Ziv run-length encoding of horizontal scan lines to compress the image structure. This scheme is very effective at compressing pictures comprised of long horizontal segments of identically colored pixels, and poor at encoding images involving pixel-by-pixel changes along such a scan line. This means that it is not the absolute size of a picture that determines the data that must be transmitted, but rather the horizontal complexity of the image. Thus one may include large colored images provided the horizontal color changes are few. The image in Figure 7 uses 6.4 Kbytes.

Algorithms for converting a color bit map to the GIF format are simple and well-documented [4]. For use in an interactive service it is very important to optimize the algorithm for speed. Simply written algorithms can take tens of seconds to do the conversion, whereas current RISC processors can convert large images, say 500 by 300, in less than 1 second. GIF conversion is a pixel by pixel operation and its speed is dominated by one small inner loop that takes 99% of the computation time. Careful optimization of this loop can result in substantial speedups, and if image output is a substantial part of the application then converting the loop to native instructions is worth considering.

An image embedded in an HTML form can be treated as an interactive widget in that it can be used to submit data to the server when the user clicks in it, returning the value of the coordinates where the click occurred. This allows generated images to be used to support complex user interactions. For example, clicking in on an element or construct name in Figure 7 causes WebGrid to return documents allowing the values in the grid to be edited for the element or construct selected.

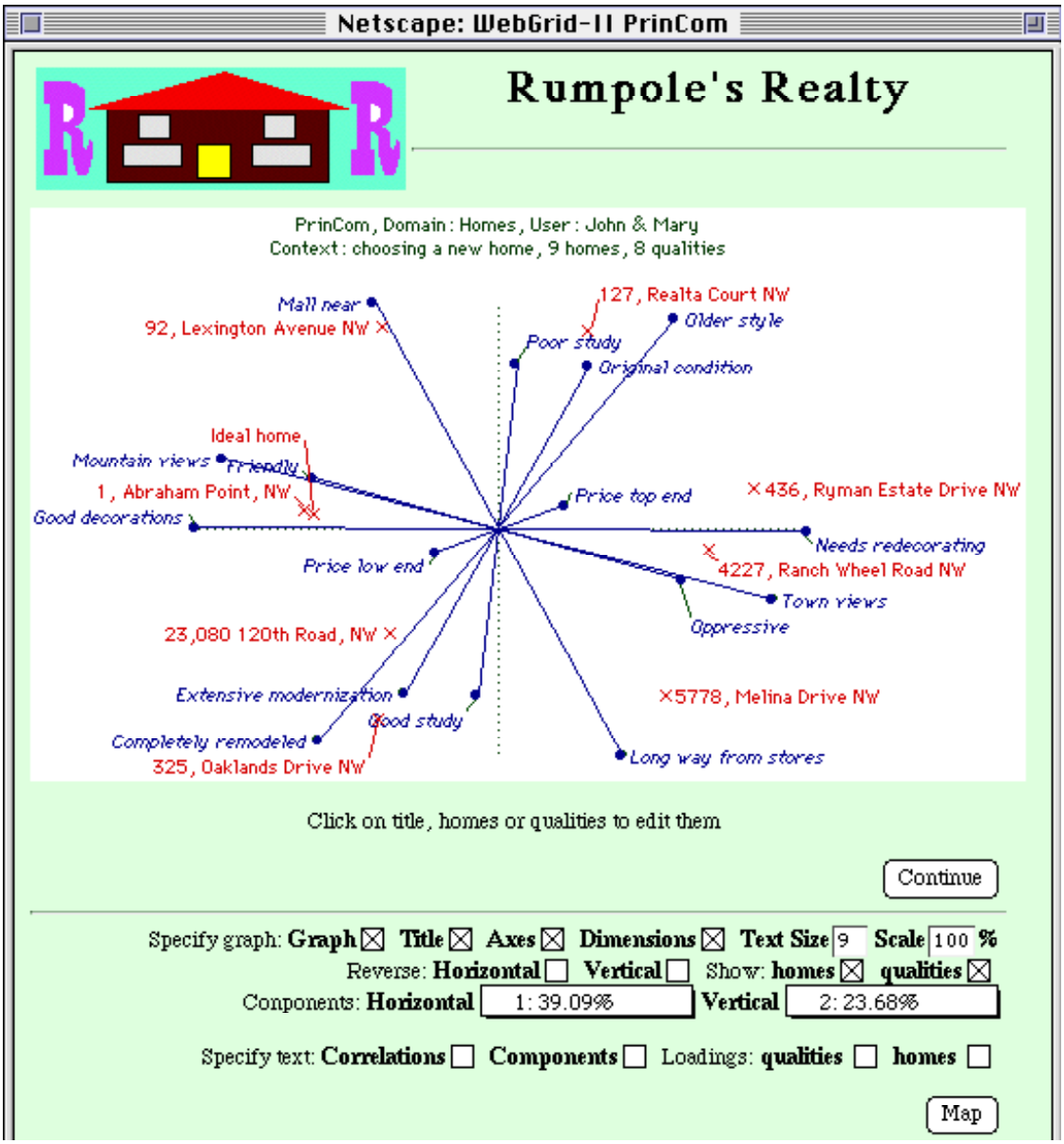

Figure 7 Graphical data analysis returned as an interactive clickable image

3.4 HTTP as a stateless protocol not supporting application state maintenance and storage

The hypertext transfer protocol was designed to be stateless and not to require any maintenance or storage of the application state across transactions. This is appropriate to a protocol for fetching documents from a server but does not support in itself interactive applications where much of the role of the interaction is to change the application state and where the state must be stored between transactions. HTML does provide for state information to be stored in a document in such a way that it is returned to the server, either in user-visible fields in a form or in 'hidden' fields that have no visible effect on the presentation. The HTTP protocol has also been extended to support 'cookies', small data structures that the server can send to, and retrieve from, the client browser. This article will focus on the use of hidden fields to maintain the application state.

A hidden field in HTML is an embedded INPUT tag in a document that is specified as having type HIDDEN and contains data (of up to 1024 characters) under a specified name. The name and data are returned to the server when the user takes action that causes data to be sent to the server. There is no limit to the number of hidden fields so that, in theory, an indefinite amount of state information can be embedded in a document. In practice, the time taken to transmit the document limits the size of document one can create without causing interactivity to deteriorate and this depends on the bandwidth of the user's connection through the Internet. For users accessing the Internet though modems a few thousand characters of state data will begin to impair interactivity.

The attribute/value datasets used in repertory grids typically are small enough that it was reasonable in WebGrid to design the implementation such that the application state is completely maintained in the document and no information is kept at the server. This has the advantage that anonymous use of the service can be supported very simply since users do not need to manage data at the server. Figure 8 shows the HTML generating the document in Figure 6 so that the hidden data can be seen. A tab characters is used as a separator to allow multiple data items to be stored in a single field, and this has been replaced by a "|" to make this visible in Figure 8. The browser uses the tab key to move the cursor between fields and the tab character can not be entered by the user. Note that the server has returned a full pathname to itself in the initial BASE tag. This is important in enabling the user to store the data by storing the HTML source in a file at the client. When the form is loaded locally and user action is recommenced it will reconnect to the server.

The advantages of storing all the data in the documents are:

- No data is stored at the server and no management of persistent data is necessary
- Application state filing can be supported at the client by saving the source of the HTML form
- The user can use the "Back" command in the browser freely as a multiple-level "Undo" without any special support at the server.

The disadvantage has already been noted, that the amount of data being transmitted over the network may impair interactivity. In large-scale applications, or in groupware ones involving multiple users accessing the data, the technique described may not be applicable. It is feasible to maintain state information at the server by saving it there between transactions and using a hidden field in the document to reference the stored data. The number 2971012332 in the first hidden field near the top of Figure 8 is a unique identifier for the transaction to support such linkage that is automatically generated by the shell auxiliary server manager in which WebGrid is written. It would be possible to store the state information for WebGrid transactions under this ID without changing its apparent mode of operation to the user.

```
<HTML><HEAD><BASE HREF="http://gigi.cpsc.ucalgary.ca/">
<TITLE>WebGrid-II Elicitation</TITLE></HEAD>
<BODY BGCOLOR="DDFFDD"><IMG SRC="/RealEstate/RR.gif" ALIGN="left"><H1 ALIGN="center">Rumpole's
    Realty</H1><HR><BR CLEAR="left"><FORM ACTION="WebGrid/Main.k" METHOD=POST>
<INPUT TYPE="hidden" NAME="WebGrid" VALUE="10|2971012332|2/22/98|4:15:43 PM">
<INPUT TYPE="hidden" NAME="Head" VALUE="6|0|1|1|0|0|9|8|9|R|1|5|">
<INPUT TYPE="hidden" NAME="Labels" VALUE="John & Mary|Homes|choosing a new home|John &
    Mary|:Datasets:RealEstate|quality|qualities|home|homes">
<INPUT TYPE="hidden" NAME="C0" VALUE="R|0|0|10|10|1|5||Extensive modernization|Original condition">
<INPUT TYPE="hidden" NAME="C1" VALUE="R|0|0|10|10|1|5||Mall near|Long way from stores">
    .....
<INPUT TYPE="hidden" NAME="E0" VALUE="0|0|4|1|0|0|0|0|4|325, Oaklands Drive NW|<P ALIGN=^CENTER^><IMG
    SRC=^/RealEstate/H107.gif^><IMG SRC=^/RealEstate/R324.gif^> <BR>Drive needs resurfacing. Mary--kitchen has
    a lovely view.Good master bedroom? <BR>">
    .....
<INPUT TYPE="hidden" NAME="Header" VALUE="<IMG SRC=^/RealEstate/RR.gif^ ALIGN=^left^><H1
    ALIGN=^center^>Rumpole's Realty</H1><HR><BR CLEAR=^left^>">
<INPUT TYPE="hidden" NAME="Body" VALUE="<BODY BGCOLOR=^DDFFDD^>">
<INPUT TYPE="hidden" NAME="Line" VALUE="<HR>">
<BR CLEAR="LEFT">
You are considering <B>9</B> homes and <B>8</B> qualities in the context of <B>choosing a new home</B>
<INPUT TYPE="Submit" NAME="Continue" VALUE="Continue">
<INPUT TYPE="IMAGE" SRC="/q.gif" BORDER=0 NAME="!Main.Continue">
<INPUT TYPE="hidden" NAME="CPair" VALUE="7|-32766|">
<HR>The qualities <B>Needs redecorating--Good decorations</B> and <B>Town views--Mountain views</B> are very
    similar -
click here if you want to enter another home to distinguish them
<INPUT TYPE="Submit" NAME="CMatch" VALUE="Distinguish">
<INPUT TYPE="IMAGE" SRC="/q.gif" BORDER=0 NAME="!Main.CMatch">
<INPUT TYPE="hidden" NAME="EPair" VALUE="0|7|">
<HR>The homes <B>325, Oaklands Drive NW</B> and <B>23,080 120th Road, NW</B> are very similar -
click here if you want to enter another quality to distinguish them
<INPUT TYPE="Submit" NAME="EMatch" VALUE="Distinguish">
<INPUT TYPE="IMAGE" SRC="/q.gif" BORDER=0 NAME="!Main.EMatch">
<HR><B>You can elicit another quality using a pair or triad of homes</B>
<INPUT TYPE="Submit" NAME="Pair" VALUE="Pair">
<INPUT TYPE="Submit" NAME="Triad" VALUE="Triad">
<INPUT TYPE="IMAGE" SRC="/q.gif" BORDER=0 NAME="!Main.Triad">
<BR>If you want specific homes included, select them in the list below and check this box
<INPUT TYPE="CHECKBOX" NAME="TSelect" VALUE="TSelect">
<DIV ALIGN="CENTER">
<HR><B>You can delete, edit, add and show matches among homes</B>
<BR><SELECT NAME="Elements" MULTIPLE SIZE="9">
<OPTION VALUE="0">325, Oaklands Drive NW
<OPTION VALUE="1">4227, Ranch Wheel Road NW
    .....
</SELECT><BR>
<INPUT TYPE="Submit" NAME="EDelete" VALUE="Delete"><INPUT TYPE="Submit" NAME="EEdit"
    VALUE="Edit"><INPUT TYPE="Submit" NAME="EAdd" VALUE="Add">
<INPUT TYPE="Submit" NAME="EShow" VALUE="Show matches">
<INPUT TYPE="Submit" NAME="ENotes" VALUE="Edit note">
<INPUT TYPE="IMAGE" SRC="/q.gif" BORDER=0 NAME="!Main.Elements">
</DIV>
    .....
</FORM></BODY></HTML>
```

Figure 8 HTML document generating user interface in Figure 6

The issues that have to be resolved when persistent storage at the server is used are:

- Long-term state filing at the server must be controlled, for example, through an authorization system, or provision must be made for users to download the state for local storage and to upload it to continue the transaction
- The effect of the "Back" command in the browser must be carefully managed since the user may step back in a series of transactions and attempt to take actions based on past state information, for example, by keeping state data for a period of time and generating an error message when it has been purged
- If multiple users can access the data stored at the server then appropriate transaction interlocks must be provided.

These issues are present in conventional client-server systems and can be solved within a transaction processing framework. However, if one is just porting an interactive application to the web it is worth avoiding them as much as possible. One should not assume that state information has to be stored at the server. As much of the information as possible should be stored in hidden data in forms and server storage used only when it is essential to the application.

In many applications it is appropriate to combine the two techniques. For example, WebGrid supports collaborative interaction in which grids need to be uploaded to the server to make them available for comparison with others. This is done through a scheme whereby an anonymous user may cache a grid under a system-generated unique identifier which is returned to the user in a URL which may be supplied to others. Cached data is cleared after 3 months but may be refreshed at any time so that the management of the cache is under control of the anonymous users without any central authorization system.

3.5 Encoding dynamic HTML data from the user for display, editing and hidden fields

To take full advantage of an application running on the web it is useful to be able to support the user entering HTML data which will be inserted in the output in appropriate places. For example, the "Rumpole's Realty" header in Figures 4, 6 and 7 is generated through user customization of the WebGrid header. The text box at the bottom of Figure 9 shows the data to do this being made available for entry and editing.

User entered dynamic HTML data has to be translated between three different forms:

- Raw HTML which is received as POST data from the text box and output in other forms as part of the form
- Encoded HTML which is output to the text box for editing purposes and is to be interpreted as character data and not as tags, so that, for example, the "<" character needs to be encoded as the string "<"
- Encoded HTML which is stored in hidden fields and must neither be interpreted as character data nor as tags, so that some alternative character must be substituted for the quotation character—WebGrid uses "^" and does not allow this character in data entry.

It is simple to write routines that translate between these formats and offer the user the opportunity to enter HTML data, such as personal notes and links to images, that is used as part of the interface documents.

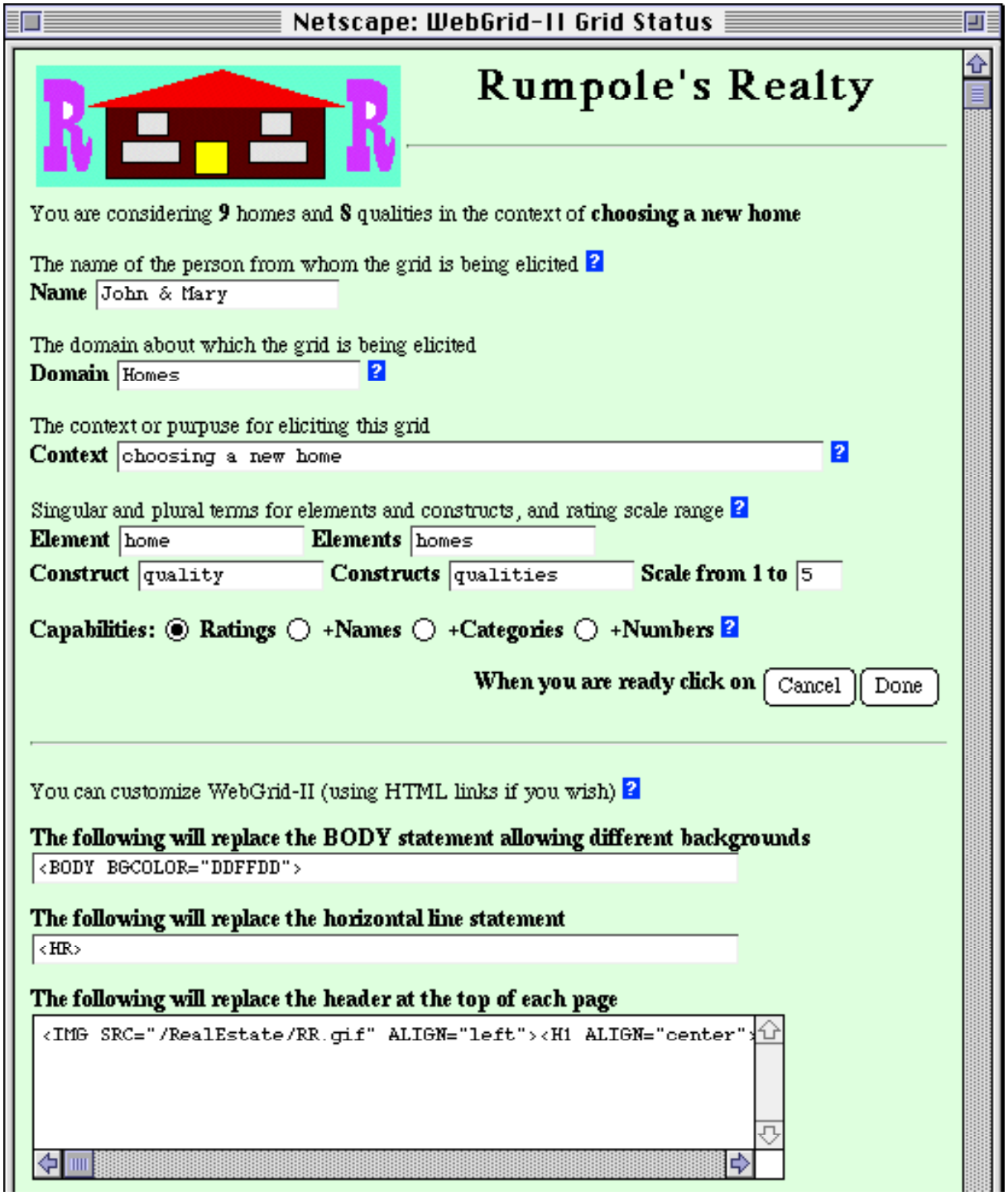

Figure 9 HTML data entered and edited by users to customize output

3.6 Designing for multiple languages and for customization

It has become standard practice in interactive computer applications to embed all user dialog in some form of "resource file" that can be edited quite separately from the application code. This enables applications to be customized for different purposes and localized for use in different languages by editing the resource file without having to recompile the code.

It is worthwhile taking a similar approach to client-server systems on the web, and putting all user dialog into separate script files that can be edited in a text editor. This allows the server to operate in multiple languages based on a simple parameter embedded in the form with values, “English”, “French”, “German” and so on. The script files can usually take the form of HTML with embedded macros that are expanded from parameters set up by the code.

Figure 10 shows the WebGrid script for the HTML which generates the status screen of Figure 9. The backslash character “\” is used to demarcate the script sections and to introduce macro codes. A line containing only a backslash is a section terminator. A line containing a double backslash followed by a letter starts a section named by that letter. In the code the start of a new section causes a case statement jump to a routine associated with that letter.

```
<TITLE>WebGrid-II Grid Status</TITLE>\
<IMG SRC="WebGrid/WebGrid.gif" ALIGN="left" ALT="WebGrid Icon">
<H1 ALIGN="center">WebGrid-II <I>Setup Grid</I></H1>\I
\
\\N
You are considering <B>\0</B> \1 and <B>\2</B> \3 in the context of <B>\D</B>
\
\\U
<P>\#0The name of the person from whom the grid is being elicited\
\#Enter your name\

\?Status.Name\
<BR><B>Name </B><INPUT NAME="User" VALUE="\u"><BR><BR>
\
\\F
The domain about which the grid is being elicited<BR>
<B>Domain </B><INPUT NAME="Domain" VALUE="\d">
\?Status.Domain\
<BR><BR>The context or purpuse for eliciting this grid<BR>
<B>Context </B><INPUT NAME="Context" VALUE="\D" SIZE=60>
\?Status.Context\
<BR><BR>Singular and plural terms for elements and constructs, and rating scale range
\?Status.Terms\
<BR><B>Element <INPUT NAME="ETerm" VALUE="\e" SIZE=15>
Elements </B><INPUT NAME="ETerms" VALUE="\E" SIZE=15><BR>
<B>Construct <INPUT NAME="CTerm" VALUE="\c" SIZE=15>
Constructs </B><INPUT NAME="CTerms" VALUE="\C" SIZE=15>
<B>Scale from 1 to <INPUT NAME="High" VALUE="\R" SIZE=3><BR>
<P>Capabilities: <INPUT TYPE="Radio" NAME="Mode" VALUE="R" \4> Ratings
<INPUT TYPE="Radio" NAME="Mode" VALUE="N" \4> +Names
<INPUT TYPE="Radio" NAME="Mode" VALUE="C" \4> +Categories
<INPUT TYPE="Radio" NAME="Mode" VALUE="V" \4> +Numbers</B>
\?Status.Capabilities\
\
\\B
<DIV ALIGN="RIGHT"><P><B>When you are ready click on</B>
<INPUT TYPE="Submit" NAME="Cancel" VALUE="Cancel">\
<INPUT TYPE="Submit" NAME="OK" VALUE="Done"></DIV>
\
\\C
<P>\I<P>You can customize WebGrid-II (using HTML links if you wish)
```

```
\?Status.Customize\
<BR><P><B>The following will replace the BODY statement allowing different backgrounds<BR>
<INPUT NAME="Bodying" VALUE="\B" SIZE=60><BR>
<P>The following will replace the horizontal line statement<BR>
<INPUT NAME="Lining" VALUE="\L" SIZE=60><BR>
<P>The following will replace the header at the top of each page<BR>
<TEXTAREA NAME="Heading" ROWS=5 COLS=60>\h</TEXTAREA>
<P>The following will appear at the bottom of each page</B><BR>
<TEXTAREA NAME="Trailing" ROWS=5 COLS=60>\H</TEXTAREA>\
\
\\
```

Figure 10 WebGrid script for the status screen of Figure 9

In the script text a backslash followed by a number indicates a parameter to be substituted by some data already set up in the code. A backslash followed by a "#" and a number introduces a list of possible strings to be selected on the basis of a numeric parameter set up by the code. It is simple to develop a macro substitution language that can be used to parameterize the output from a range of applications. The effort is amply repaid by the capability to change the user interface without recompilation, and to customize the application.

Figure 11 shows the initial part of the script for the document of Figure 6 in Spanish and Figure 12 shows the resulting screen. The English user input can be seen embedded in the Spanish script.

```
<TITLE>WebGrid Elicitation</TITLE>\
<img src="\S/WebGrid/WebGrid.gif" align="left" alt="WebGrid Icon">
<h1 align="center">WebGrid <i>Elicitation</i></h1><hr>
\
\\N
Usted est&#225; considerando <b>\0</b> \1 y <b>\2</b> \3 en el contexto de <b>\D</b>
\
\\m
<hr>Los \C <b>\0</b> y <b>\1</b> son muy similares -
Oprima aqui si Usted desea a&#241;adir algun otro \e para diferenciarlos
<input type="Submit" name="CMatch" value="Diferenciar">
\
\\M
<hr>Los \E <b>\0</b> y <b>\1</b> son muy similares -
Oprima aqui si Usted desea a&#241;adir algun otro \c para diferenciarlos
<input type="Submit" name="EMatch" value="Diferenciar">
\
\\T
<hr><b>Usted puede generar otro \c utilizando un tr&#237;o de \E</b>
<input type="Submit" name="Triad" value="Tr&#237;o"><br>
Si desea inclu&#237;r un \E espec&#237;fico, Usted podr&#225; seleccionarlo de la siguiente lista
\
.....
```

Figure 11 First part of WebScript in Spanish for status screen of Figure 6

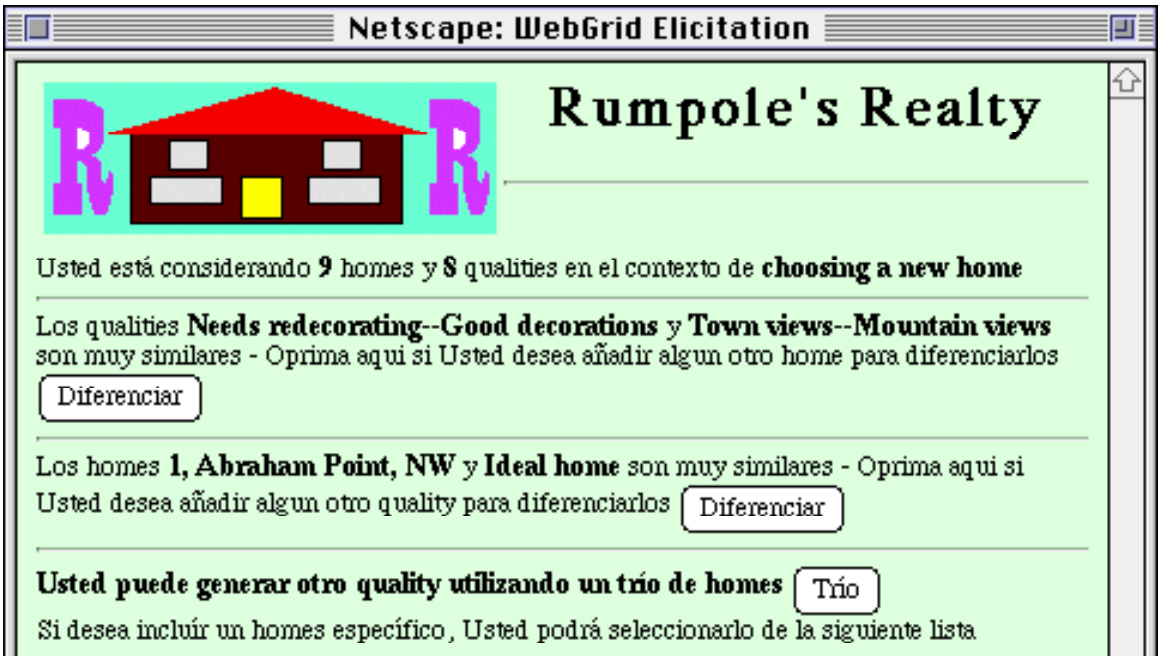

Figure 12 Resultant status screen document in Spanish

4 A Software Architecture for Porting Interactive Applications to the Web

Figure 13 refines Figure 2 to show the detailed software architecture of the auxiliary server shell through which WebGrid was ported. The application-dependent modules in the auxiliary server are shown as shaded boxes, and the main data structures are shown in rounded corner boxes. At the top the web client is shown partitioned into its modules for: user interface presentation controlled by HTML as a GUI programming language; user interface interaction controlled by HTML level 2 forms; caching controlled by the browser; and filing controlled by the user. A link is shown between the presentation and interaction modules to indicate that the presentation changes locally during interaction, and this link can be enhanced by appropriate use of JavaScript.

Below this the web server is shown as a device for managing the HTTP protocol and supporting the auxiliary server through a common gateway interface. At the bottom the application services module is shown as a black box not subject to redesign as part of the port. The rest of the auxiliary server modules have been the subject of this article and this section provides additional details of their operation.

The top two modules of the auxiliary server are the input and output device protocol managers. These are specific to the web server being used and standardize the interface to the server through its CGI providing whatever data conversions that are necessary and accessing the server through whatever interface it provides.

This enables a standard presentation generator driving a standard script interpreter to be used to generate the HTML documents that control the presentation from one of a family of scripts. This is interfaced to an application-dependent set of script processes that expand the script macros through access to the application state. It is also interfaced to a database of scripts that customize the application.

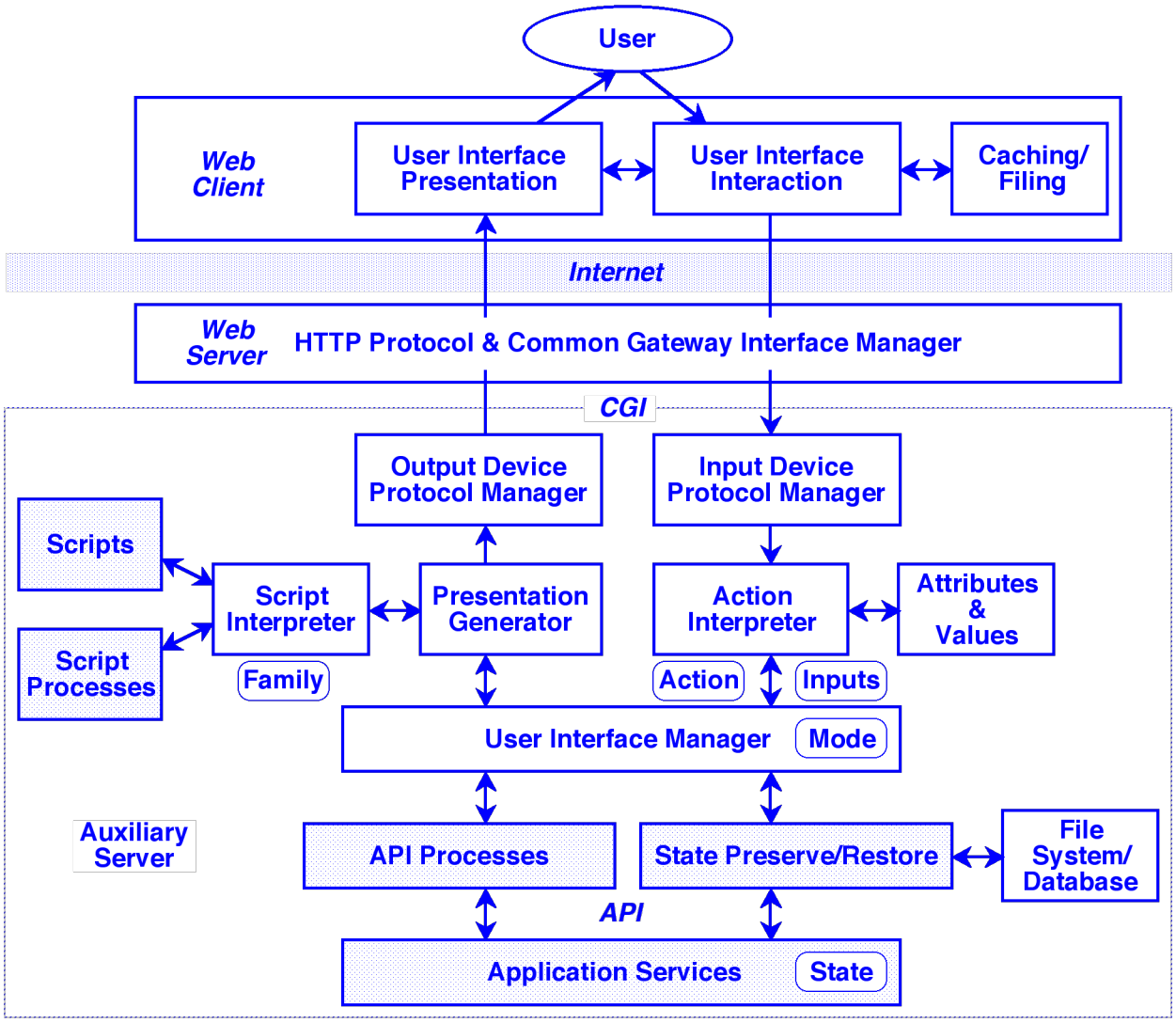

Figure 13 Software architecture for porting interactive applications to the web

It also enables a standard action interpreter to be used to access the attributes and values delivered from the user through the CGI and make them available as an action code and a set of input data.

The action interpreter drives a standard user interface manager module that in turn drives the presentation generator (the arrows in the diagram are shown as two-way because most data access is done by call-backs). The user interface manager has a state which determines the mode of the interaction which the user sees as the type of document produced. It accesses the application services through an application-dependent set of API processes that are small routines accessing and changing the application state according to the mode, action and inputs. It also accesses the services through state preserve and restore routines which store and retrieve the state either through access to a local database or by packing it into hidden fields in the document.

The logic of the auxiliary server automaton is that:-

Family x Mode → Script	the script called is determined by the family and mode
Mode x Action → API Process	the API process called is determined by the mode and action
API Process x Inputs x State → Next State	the next state is determined by the API process, the inputs and the current state
API Process x State → Next Mode	the next mode called is determined by the API process and, possibly, the current state

The first two steps in this logic is implemented through a set of tables set up for the specific application.

In the current WebGrid implementation, the script family and the interface manager mode are encoded into the document URL since this is recorded in the server log and thus enables the sequence of transactions to be seen. The application state is encoded in hidden fields in the documents, and the action plus other inputs originate from the other input fields in the form.

A significant feature of this architecture is that it is equally applicable to a stand-alone implementation of the application where the Internet and Web Server layers are not present. The output and input device protocol managers, and the scripts, are all that need to be changed between the stand-alone and web implementations. The overall architecture represents good software engineering practice for implementing any interactive application.

5 Conclusions

Porting interactive applications from personal computers to operate in a client-server mode on the web is a matter of factoring them effectively and of recognizing some restrictions of the HTML GUI and HTTP protocol. A software architecture has been described that factors the application into its user interface module and an interface-independent service module. The user interface module has been further factored into generic modules that can be written once as a shell, and implementation-specific modules that differentiate the stand-alone application from the web implementation. Once an application has been factored into the architecture defined it is feasible to maintain the stand-alone and web versions as a common set of application-specific code modules that are linked with differing device protocol managers to generate the different versions.

The application used as an example uses only HTML level 2 capabilities and illustrates what can be achieved within these limitations. The architecture also supports more advanced web features such as JavaScript and download of Java widgets which enable complex customized interfaces to be implemented.

References

[1] B.R. Gaines and M.L.G. Shaw, “Documents as expert systems,” in *Research and Development in Expert Systems IX. Proceedings of British Computer Society Expert Systems Conference,* M.A. Bramer and R.W. Milne, Editor. Cambridge University Press: Cambridge, UK. p. 331-349, 1992.

[2] B.R. Gaines and M.L.G. Shaw, "Eliciting knowledge and transferring it effectively to a knowledge-based systems," *IEEE Transactions on Knowledge and Data Engineering*, vol. 5, no. 1 pp. 4-14, 1993.

[3] B.R. Gaines and M.L.G. Shaw, "Knowledge acquisition, modeling and inference through the World Wide Web," *International Journal Human-Computer Studies*, vol. 46, no. 6 pp. 729-759, 1997.

[4] M. Luse, *Bitmapped Graphics Programming in C++*, Reading, Massachusetts: Addison-Wesley. 1993.

[5] M.L.G. Shaw and B.R. Gaines, "Comparing constructions through the web," in *Proceedings of CSCL95: Computer Support for Collaborative Learning,* J.L. Schnase and E.L. Cunnius, Editor. Lawrence Erlbaum: Mahwah, New Jersey. p. 300-307, 1995.