Naming Page Elements in End-User Web Automation

Michael Bolin and Robert C. Miller
MIT CSAIL
32 Vassar St
Cambridge, MA 02139 USA
{rcm,mbolin}@mit.edu

ABSTRACT
The names of commands and objects are vital to the usability of a programming system. We are developing a web automation system in which users need to identify web page elements, such as hyperlinks and form fields, in pages written by other designers. Using a survey of 40 users asking them to provide names for page elements, we found that users' names varied widely. However, when names were restricted to using only visible words from the web page, we were able to develop name resolution techniques that automatically find the desired page element given the user's name for it, striking a balance between usability and the precision required by the programming system.

Categories and Subject Descriptors
D.3.3 [**Programming Languages**]: Language Constructs and Features; D.2.6 [**Programming Environments**]: Interactive environments; H.5.2 [**User Interfaces**]: User-centered design.

General Terms
Algorithms, Experimentation, Human Factors, Languages.

Keywords
End-user web automation, web browsers.

1. INTRODUCTION
The names given to software components – such as variables, functions, classes, and commands – are an important part of the user interface of an end-user programming system. Choice of names, whether made by the system's designers or by end-user programmers themselves, can affect learnability, recall, readability, and maintainability of programs. Professional programmers recognize the importance of names, and naming conventions are the result (e.g., [3],[10]). But a classic study of naming by Furnas *et al.* [2] showed that command names chosen by different people were unlikely to be consistent. The solution proposed by Furnas *et al.* was *unlimited aliasing*, allowing "many, many alternate verbal routes" to the same functionality.

In this paper, we discuss how we have applied unlimited aliasing in the design of an end-user programming system for automating and customizing interaction with the Web. The main question we consider is how a user should refer to elements on a web page (such as hyperlinks and form fields) in customization or scripting, particularly when the web page was authored by another designer.

In the next section (section 2), we discuss a number of design principles that interact in the choice of a name. In section 3, we describe Chickenfoot, the end-user web automation system we are developing. In section 4, we present a pilot study we conducted to learn how users might name web page elements. Finally, in section 5, we outline a name resolution algorithm that implements a form of automatic aliasing that performs well on the kinds of names we discovered in the study.

First Workshop on End-User Software Engineering (WEUSE I). May 21 2005, Saint Louis, Missouri, USA.

2. DESIGN PRINCIPLES
The goal of a name, whether used in programming or in natural language, is to identify a thing, so that both the writer and the reader agree about which thing is under discussion. Unlike natural languages, however, programming languages have two kinds of readers with very different needs: software and humans. In this section, we discuss some of the properties of names that are relevant to programming, and how they matter to these two kinds of readers.

Precision. To software tools, such as compilers or interpreters, the most important property of a name is *precision*. A precise name identifies exactly one thing. Naming systems in software are generally designed to minimize ambiguity, rejecting attempts to introduce names that would be imprecise. For example, file systems generally refuse to allow two files of the same name in the same directory. In Java, two variables in the same scope may not share the same name, and two classes with the same name may not be imported simultaneously. For a software tool, *name collisions* are the worst kind of failure that can occur, since they leave the software unable to resolve references to the name.

Precision is not as important to people, since humans are more tolerant of ambiguity. One way people resolve ambiguity is by appealing to context. For example, in a discussion of Java collection classes, *List* probably means the collection class *java.util.List*, not the user interface widget *java.awt.List*. Another way to resolve ambiguity is to engage in a dialogue ("Which *List* do you mean?"), but this is only feasible when the communication is interactive.

Robustness. Since software engineering is also concerned with the correctness of a program over its entire lifecycle (maintainability) and in other contexts (reuse and extensibility), a well-chosen name in a well-designed naming system should *remain* precise as a program is modified and combined with code written by other programmers. The need for precision over time and space is what drives naming systems to introduce scoping and

package mechanisms, in order to isolate one module's names from another's. When names chosen by different programmers must coexist, naming conventions are developed that reduce the chance of a collision, usually by referring to an external source of unique identifiers. For example, Java programmers are encouraged to prefix their package names with their organization's domain name, such as com.sun.*, since the uniqueness of domain names is guaranteed by domain name registrars [9]. An extreme form of this approach is the use of universally unique identifiers (UUIDs), constructed from a network card's MAC address and the current clock time. UUIDs are used in Microsoft COM to name classes and interfaces, and in RDF to name objects and properties.

Suggestiveness. If every name were a UUID, precision and robustness would be satisfied, and complicated scoping and namespace rules would be unnecessary. (Indeed, many source code analysis tools internally rename all the user's messy names with fresh unique identifiers to simplify managing these rules.) Human programmers, on the other hand, would find this intolerable, since they depend on other properties of a name. The most important of these properties is *suggestiveness*, the extent to which a name describes the content, use, and type of the thing it identifies. A variable named *radius* is more suggestive than one named *r*. Suggestiveness depends strongly on shared experience between the writer of the code and its reader, and also on the context of the code. In code dealing with polar coordinates, *r* may be just as suggestive as *radius*.

Suggestiveness lies behind recommendations to use long identifiers, including whole words and multiple words, and avoiding unnecessary abbreviations. Suggestiveness drives the naming conventions used in many languages and APIs. In Java, for example, case distinctions are conventionally used to suggest whether a name refers to a variable (*string*), a class (*String*) or a constant (*STRING*). Hungarian notation [9], first articulated by Charles Simonyi and widely used in the Microsoft Windows API, uses short prefixes to encode the type of a variable in the name. For example, in *lpszFirstName*, the prefix *lpsz* means *long pointer to string terminated by zero*. Hungarian notation can make finer distinctions in type and usage than the C/C++ type system is capable of expressing. For example, *ichFirstName* and *cchFirstName* are both integer variables, but the former should be used an *index* into a character array, while the latter represents the *count* of the characters in the array.

Names have other properties that are important for human readers. Some of these properties can be derived from well-known usability design heuristics [6]:

Consistency means using similar names for similar things, and dissimilar names for dissimilar things. For example, a human reader can more readily recognize an idiom like *for* (*int i=0; i<n; ++i)* when the loop control variable is consistently named *i*. Conversely, using *s1* to name a string and *s2* for a stream in the same function is ripe for confusion. Naming conventions help improve consistency.

Efficiency means that (all other properties equal) a shorter name is better than a longer name. Shorter names are simply faster to use, whether the user is typing them, reading them, or speaking them. Efficiency often forces a tradeoff with suggestiveness, since shorter names have fewer suggestive cues.

Error prevention is also desirable. A good name should not be prone to misspelling or misreading. For example, *weird* may be easily misspelled as *wierd* or misread as *wired*. We noticed this effect in developing a text pattern language which used *containing* as a pattern operator. So many users mistyped it as *containg* – even we, the system's developers, made the same mistake – that we eventually added *containg* to the grammar as an alias, as well as the less error-prone *contains*.

Pronunciation. Although names in computer systems are primarily used in written form (typed on a keyboard or read on a screen or on paper), pronunciation also matters, since people often talk about the names. In software development, this may happen in design discussions, code reviews, classes, or in pair programming. Unpronounceable names like *m_lprgchName* seriously inhibit this kind of communication. URLs were not designed with pronunciation in mind: *http://www* is so hard to say that most speakers simply omit it, and web browsers wisely tolerate the omission. (Tim Berners-Lee reads *www* as "wuh-wuh-wuh," but that hasn't caught on.)

3. CHICKENFOOT

We have encountered some of these naming issues in the design of Chickenfoot, an end-user programming system integrated into a web browser.

The primary goal of Chickenfoot is to give the user a platform for automating and customizing their interaction with the Web. Although web browsers have a long history of built-in scripting languages, these languages are not designed for the *end user* of a web site. Instead, languages like JavaScript, Java, and Curl [7] are aimed at *designers* of web sites. Granted, many web designers lack a traditional programming background, so they may be considered end-user programmers in that respect. But the needs of a designer, building a web application from whole cloth, differ significantly from the needs of a user looking to tailor or script an existing web site. Current web scripting languages do not serve the needs of web automation.

A second goal of Chickenfoot is to allow the end user to automate and customize web sites using a familiar interface, namely the web site's user interface. Existing approaches to web automation use a scripting language that dwells outside the web browser, such as Perl, Python, screen-scraper [1], and WebL [4]. For an end-user, the distinction is significant. Cookies, authentication, session identifiers, plugins, user agents, client-side scripting, and proxies can all conspire to make the Web look significantly different to a script running outside the web browser. But perhaps the most telling difference, and the most intimidating one for an end user, is the simple fact that outside a web browser, a web page is just raw HTML. Even the most familiar web portal looks frighteningly complicated when viewed as HTML source. So the challenge for Chickenfoot can be simply stated: a user should never have to view the HTML source of a web site in order to customize or automate it.

Chickenfoot is targeted mainly at three kinds of automation:

Automating repetitive operations. For example, many conferences now use a web site to receive papers, distribute them to reviewers, and collect the reviews. A reviewer assigned 10 papers to read and review faces a lot of repetitive web browsing to download each paper, print it, and later upload a review. Tedious

repetition is a strong argument for automation. Other examples include submitting multiple search queries and comparing the results, and collecting multiple pages of search results into a single page for sorting, filtering, or printing.

Integrating multiple web sites. Some web sites already provide some level of integration with other sites. For example, many retailers use MapQuest to display their store locations and provide driving directions. But end-users have no control over this integration. For example, before buying a book from an online bookstore, a user may want to know whether it is available in the local library—a question that can be answered by submitting a query to the library's online catalog interface. Yet the online bookstore is unlikely to provide this kind of integration, not only because it may lose sales, but because the choice of library is inherently local and personalized to the user.

Transforming a web site's appearance. Examples of this kind of customization include changing defaults for form fields, filtering or rearranging web page content, and changing fonts, colors, or element sizes. Web sites that use Cascading Style Sheets (CSS) have the potential to give the end user substantial control over how the site is displayed, since the user can override the presentation with personal stylesheet rules. With the exception of font preferences, however, current web browsers do not expose this capability in any usable way.

3.1 Design

Chickenfoot is being developed as an extension to the Mozilla Firefox web browser. Chickenfoot's design has two parts: (1) a development environment that allows users to enter and test Chickenfoot programs, and (2) a library that extends the browser's built-in Javascript language with new commands for web automation.

Figure 1 shows a screenshot of the development environment presented by the current Chickenfoot prototype, which appears as a sidebar in Firefox. At the top of the sidebar is a text editor which accepts a Javascript program, which may be merely a single expression or command to execute, or a larger program with function and class definitions. The bottom of the sidebar is a console output window, which displays error messages, printed output, and the result of evaluating the Javascript code (i.e., the value of the last expression). This interface, though minimal, goes a long way toward making the Javascript interpreter embedded in every web browser actually *accessible* to the end-user. Previously, there were only two ways to run Javascript in a web browser: by embedding it in a web page (generally impossible if the page is fetched from a remote web site, since the user can't edit it), or by using a *javascript:* URL, which requires the entire program to be written on a single line.

Figure 1. Chickenfoot development environment running inside the Firefox web browser.

A Javascript program running in the Chickenfoot sidebar operates on the web page shown in the main part of the window. Unlike most Javascript, Chickenfoot scripts run with no security restrictions, since they are developed and run by the end-user, not downloaded from a potentially malicious remote site. A Chickenfoot script is therefore free to interact with web pages from arbitrary sites and examine any aspect of the web browser's history or user interface.

Chickenfoot extends the standard client-side Javascript with a number of commands to simplify web automation. Some of these commands simulate actions that a web user can perform on the hyperlinks and forms of a web page:

```
click (link-or-button)
enter (textbox, value)
pick (menu-or-list, option)
check (checkbox-or-radiobutton)
uncheck (checkbox)
```

These commands raise the question at the heart of this paper: what name should we use for the page object (link, button, or other widget) that a command should act on?

For a form widget, like a textbox or a checkbox, one possibility is the name assigned to the widget by the web page designer. This is the name used by Javascript embedded in the web page, and in the HTTP request sent back to the web server when the form is submitted. One key drawback of this name is that it isn't readily available to a web user without examining the HTML source, which contradicts one of the goals of Chickenfoot. The Chickenfoot development environment could solve this problem (e.g., by making form field names visible in the page on command). But these names have a second problem: since they are not chosen by the web user nor intended to be seen by the web user, they are not likely to be *suggestive*. For example, Google forms use names like *as_q*, *as_qdr*, and *as_occt*; MapQuest fields look like *2c* and *2s*. These names are virtually opaque to a user.

Another possibility is to use *pointing* to identify a page object, rather than a textual name. Indeed, this approach makes a lot of sense when the user is developing a new script, and our future plans include creating a programming-by-demonstration system on top of Chickenfoot, so that the user's clicks and keystrokes are translated automatically into Chickenfoot statements. But even if the user points at page objects to generate Chickenfoot code, there remains the question of what names to display in the generated code. Although visual representations of the code are possible (e.g. [5],[6]), a compact textual name would be more efficient of screen real estate and more pronounceable.

We chose to explore a third option: using visible labels in the page to identify page objects. For example, hyperlinks and buttons typically contain a visible text label that can be used with the *click* command:

```
click("Google Search")
```

Other form widgets, such as textboxes and lists, have captions adjacent to the widget that can be used with other commands:

```
enter("User name", "john@hotmail.com")
enter("Password", "bri56ght")
click("Sign In")
```

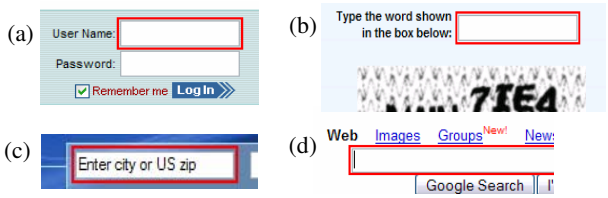

Figure 2. Sample textboxes used in the web survey.

Visible labels are very likely to be suggestive names, because they are chosen by a web site designer to be read and understood by a user, and also because the user is likely to be familiar with them from manually interacting with the web site. One challenge for this technique is the use of inline images for labeling hyperlinks and buttons. Fortunately, well-designed web sites offer ALT text for these images, intended to help visually-impaired users with screen readers, but which can help Chickenfoot as well. For images with no ALT text, we must fall back to other naming methods, such as internal page names.

4. NAMING SURVEY

To explore the usability of visible labels as names, we conducted a small pilot survey to find out what kinds of names users would generate for form fields, and whether they could comprehend names based on visible labels. Our survey focused on textboxes, which are probably the most common form field on the Web.

4.1 Method

The survey was presented entirely over the Web. It consisted of three parts, always in the same sequence. Part 1 explored freeform generation of names: given no constraints, what names would users generate? Each task in Part 1 showed a screenshot of a web page with one textbox highlighted in red, and asked the user to supply a name that "uniquely identified" the highlighted textbox. Users were explicitly told that spaces in names were acceptable. Part 2 tested comprehension of names that we generated from visible labels. Each task in Part 2 presented a name and a screenshot of a web page, and asked the user to click on the textbox identified by the given name. Part 3 repeated Part 1 (using fresh web pages), but also required the name to be composed only of "words you see in the picture" or "numbers" (so that ambiguous names could be made unique by counting, e.g. "2nd Month").

The whole survey used 20 web pages: 6 pages in Part 1, 8 in Part 2, and 6 in Part 3. The web pages were taken from popular sites, such as the Wall Street Journal, the Weather Channel, Google, AOL, MapQuest, and Amazon. Pages were selected to reflect the diversity of textbox labeling seen across the Web, including simple captions (Figure 2a), wordy captions (Figure 2b), captions displayed as default values for the textbox (Figure 2c), and missing captions (Figure 2d). Several of the pages also posed ambiguity problems, such as multiple textboxes with similar or identical captions.

Subjects were unpaid volunteers recruited from the university campus by mailing lists. Forty subjects took the pilot survey (20 females, 20 males), including both programmers and nonprogrammers (24 reported their programming experience as "some" or "lots", 15 as "little" or "none", meaning at most one programming class). All but one subject were experienced web users, reporting web usage at least several times a week.

4.2 Results

We analyzed Part 1 by classifying each name generated by a user into one of four categories: (1) *visible* if the name used only words that were visible somewhere on the web page (e.g., "User name" for Figure 2a); (2) *semantic* if at least one word in the name was not found on the page, but was semantically relevant to the domain (e.g., "login name"); (3) *layout* if the name referred to the textbox's position on the page rather than its semantics (e.g., "top box right hand side"); and (4) *example* if the user used an example of a possible value for the textbox (e.g. "johnsmith056"). About a third of the names included words describing the type of the page object, such as "field", "box", "entry", and "selection"; we ignored these when classifying a name.

Two users consistently used *example* names throughout Part 1; no other users did. (It is possible these users misunderstood the directions, but since the survey was conducted anonymously over the Web, it was hard to ask them.) Similarly, one user used *layout* names consistently in Part 1, and no others did. The remaining 37 users generated either *visible* or *semantic* names. When the textbox had an explicit, concise caption, *visible* names dominated strongly (e.g., 31 out of 40 names for Figure 2a were visible). When the textbox had a wordy caption, users tended to seek a more concise name (so only 6 out of 40 names for Figure 2b were visible). Even when a caption was missing, however, the words on the page exerted some effect on users' naming (so 12 out of 40 names for Figure 2d were visible).

Part 2 found that users could flawlessly find the textbox associated with a visible name when the name was unambiguous. When a name was potentially ambiguous, users tended to resolve the ambiguity by choosing the first likely match found in a visual scan of the page. When the ambiguity was caused by both visible matching and semantic matching, however, users tended to prefer the visible match: given "City" as the target name for Go.com, 36 out of 40 users chose one of the two textboxes explicitly labeled "City"; the remaining 4 users chose the "Zip code" textbox, a semantic match that appears higher on the page. The user's visual scan also did not always proceed from top to bottom; given "First Search" as the target name for eBay.com, most users picked the search box in the middle of the page, rather than the search box tucked away in the upper right corner.

Part 3's names were almost all visible (235 names out of 240), since the directions requested only words from the page. Even in visible naming, however, users rarely reproduced a caption exactly; they would change capitalization, transpose words (writing "web search" when the caption read "Search the Web"), and mistype words. Some Part 3 answers also included the type of the page object ("box", "entry", "field"). When asked to name a textbox which had an ambiguous caption (e.g. "Search" on a page with more than one search form), most users noticed the ambiguity and tried to resolve it with one of two approaches: either counting occurrences ("search 2") or referring to other nearby captions, such as section headings ("search products").

5. AUTOMATIC NAME RESOLUTION

We have used the names from Part 3 of the survey to develop a heuristic algorithm for resolving names to textboxes in

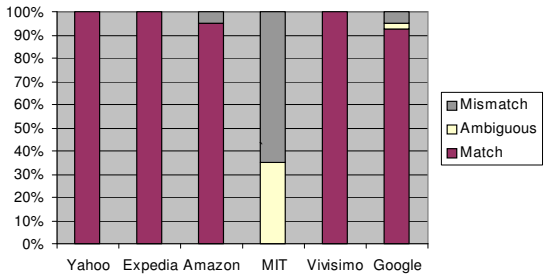

Figure 3. Precision of automatic name resolution.

Chickenfoot. Given a name and a web page, the output of the algorithm is one of the following: (1) a textbox on the page that best matches that name; (2) *ambiguous match* if two or more textboxes are considered equally good matches; or (3) *no match* if no suitable match can be found.

The first step is to identify the text labels in the page that approximately match the provided name, where a *label* is a visible string of content delimited by block-level tags (e.g. <P>,
, <TD>). Button labels and ALT attributes on images are also treated as visible labels. Before comparison, both the name and the visible labels are normalized by eliminating capitalization, punctuation, and white space. Then each label is searched for an approximate occurrence of the name, using a conventional edit distance algorithm to tolerate typos and omitted words. Matching labels are ranked by edit distance, so that closer matches are ranked higher.

For each matching label, we search the web page for textboxes for which it might be a label. Any textbox that is roughly aligned with the label (so that extending the textbox area horizontally or vertically would intersect the label's bounding box) is paired with the label to produce a candidate *(label,textbox)* pair.

These pairs are further scored by several heuristics that measure the degree of association between the label and the textbox. First is pixel distance: if the label is too far from the textbox, the pair is eliminated from consideration. Currently, we use a vertical threshold of 1.5 times the height of the textbox, but no horizontal threshold, since tabular form layouts often create large horizontal gaps between captions and their textboxes. The second heuristic is relative position: if the label appears below or to the right of the textbox, the rank of the pair is decreased, since these are unusual places for a caption. We don't completely rule them out, though, because users sometimes use the label of a nearby button, such as "Search", to describe a textbox, and the button may be below or to the right of the textbox. The final heuristic is distance in the document tree: each (*label,textbox*) is scored by the length of the minimum path from the *label* node to the *textbox* node in the document's element tree. Thus labels and textboxes that are siblings in the tree have the highest degree of association.

The result is a ranked list of (*label, textbox*) pairs. The algorithm returns the textbox of the highest-ranked pair, unless the top two pairs have the same score, in which case it returns *ambiguous match*. If the list of pairs is empty, it returns *no match*.

The performance of this algorithm is shown in Figure 3, tested on the 240 names (40 for each of the 6 pages) from Part 3 of the survey. For each name, the algorithm had three possible results: finding the right textbox (*Match*), reporting an ambiguous match (*Ambiguous*), or finding the wrong textbox (*Mismatch*). Precision is high for 5 of the 6 pages. Performance is poor on the MIT page because it involved an ambiguous caption, and our heuristic algorithm does not yet recognize the disambiguation strategies used for this caption (counting and section headings). This evaluation is only preliminary, but it suggests that names derived from visible labels can be automatically resolved with high precision.

6. CONCLUSION

We have shown preliminary results that visible naming (unlimited aliasing that uses words that are visible in the page) is a promising strategy for identifying elements in web pages. Web pages are just one kind of user interface that can be customized and automated. We anticipate that these results will generalize to other user interfaces that include textual labeling.

Future work includes improving the Chickenfoot development environment so that ambiguous names can be disambiguated during code entry, which allows for an ambiguity resolution dialog between the user and the system that wouldn't be sensible at runtime. We are also looking at the robustness of syntactic names against change in web sites. Dealing with web sites that change without warning is a challenge for web automation, but as yet no one has adequately characterized the kinds of changes that occur.

7. ACKNOWLEDGMENTS

We thank all the pilot users who took our web survey, as well as Maya Dobuzhskaya, Vineet Sinha, Philip Rha, and other members of the LAPIS group who provided valuable feedback on the ideas in this paper. This work was supported in part by the National Science Foundation under award number IIS-0447800. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

8. REFERENCES

[1] Ekiwi, LLC. screen-scraper: solutions for web data extraction. http://www.screen-scraper.com/

[2] Furnas, G.W., Landauer, T.K., Gomez, L.M., and Dumais, S.T. "The vocabulary problem in human-system communication." *Commun. ACM, 30*, 11 (Nov 1987).

[3] Green, R. "How To Write Unmaintainable Code." http://mindprod.com/unmain.html

[4] Kistler, T. and Marais, H. "WebL – a programming language for the Web." *Proc. WWW7*, 1998.

[5] Kurlander, D. "Chimera: Example-based graphical editing." In Cypher, A., ed., *Watch What I Do: Programming By Demonstration*, pp 271–292. MIT Press, 1993.

[6] Modugno, F., Corbett, A.T., and Myers, B.A. "Graphical Representation of Programs in a Demonstrational Visual Shell - An Empirical Evaluation." *ACM TOCHI*, *4*, 3, pp 276-308.

[7] Müffke, F. "The Curl programming environment." *Dr. Dobb's Journal*, Sept. 2001.

[8] Nielsen, J. *Usability Engineering*. Academic Press, 1993.

[9] Simonyi, C. and Heller, M. "The Hungarian Revolution," BYTE, *16,* 8 (Aug. 1991).

[10] Sun Microsystems. "Code Conventions for the Java Programming Language." http://java.sun.com/docs/codeconv