

# Customizing and Automating

# 2

- 3. Rewriting the Web with Chickenfoot . . . . . 39
- 4. A goal-oriented Web browser. . . . . 65
- 5. Collaborative scripting for the Web . . . . . 85
- 6. Highlight: End user re-authoring of existing Web sites. . . . . 105
- 7. Mixing the reactive with the personal: Opportunities for end user programming in personal information management (PIM) . . . . . 127

*In this section we look at five systems that customize and automate the Web experience in different ways. Chickenfoot proposes a scripting interface for controlling and augmenting Web page interfaces, making it possible to modify any bookstore page with local library book availability information. Creo and Miro leverage a vast knowledge base to generate*

*personalized semantic hypertext, like dynamically linking the names of foods to their nutritional information. CoScripter helps knowledge workers collaboratively automate repetitive tasks, such as ordering office supplies. Highlight lets users create a mobile version of any Web site, whereas Atomate makes it possible to quickly set up contextualized reminders for a task.*

*There are many similarities and differences between these systems. Chickenfoot, CoScripter, and Atomate all aim to simplify the underlying Web scripting language (i.e., JavaScript) by introducing keyword-based scripting languages that can be both read by a machine and understood by a human. CoScripter, Highlight, and Creo use a programming by example approach that lets users demonstrate the type of automation they would like to script. Highlight enables users to clip parts of Web pages as part of a demonstration, whereas Creo leverages a large knowledge base of semantic information to generate a script from a single demonstration. CoScripter also explores the social component of end user programming through sharing and reuse with a wiki. The Atomate system proposes that programming by example is not only valuable in automating information gathering and procedural tasks but also in personal information management tasks, such as setting up reminders, organizing notes, and coordinating social events between people.*

---

# Rewriting the Web with Chickenfoot

# 3

Robert C. Miller, Michael Bolin, Lydia B. Chilton, Greg Little, Matthew Webber, Chen-Hsiang Yu

MIT CSAIL

## ABSTRACT

On the desktop, an application can expect to control its user interface down to the last pixel, but on the World Wide Web, a content provider has no control over how the client will view the page, once delivered to the browser. This creates an opportunity for end users who want to automate and customize their Web experiences, but the growing complexity of Web pages and standards prevents most users from realizing this opportunity. This chapter describes Chickenfoot, a programming system embedded in the Firefox Web browser, which enables end users to automate, customize, and integrate Web applications without examining their source code. One way Chickenfoot addresses this goal is a technique for identifying page components by keyword pattern matching. We motivate this technique by studying how users name Web page components, and present a heuristic keyword matching algorithm that identifies the desired component from the user's name. We describe a range of applications that have been created using Chickenfoot and reflect on its advantages and limitations.

---

## INTRODUCTION

The World Wide Web has become a preferred platform for many kinds of application development. Over the past decade, applications that formerly would have been designed for the desktop – calendars, travel reservation systems, purchasing systems, library card catalogs, map viewers, crossword puzzles, and even Tetris – have made the transition to the Web, largely successfully.

The migration of applications to the Web opens up new opportunities for user interface customization. Applications that would have been uncustomizable on the desktop sprout numerous hooks for customization when implemented in a Web browser, without any effort on the application developer's part. Displays are represented primarily by machine-readable HTML or XML, navigation and commands are invoked by generic HTTP requests, and page designs can be reinterpreted by the browser and tailored by style sheets. Unlike desktop applications, Web applications are much more exposed and open to modification. Here are some of the customization possibilities that arise when an application is moved to the Web.

***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 style sheet rules.

**Automating repetitive operations.** Web automation may include navigating pages, filling in forms, and clicking on links. 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 must download each paper, print it, and (later) upload a review for it. Tedious repetition is a good argument for automation. Other examples include submitting multiple queries and comparing the results, and collecting multiple pages of search results into a single page for easy printing or additional sorting and filtering.

**Integrating multiple Web sites.** The simplest kind of integration is just adding links from one site to another, but much richer integration is possible. For example, many retailers' Web sites incorporate maps and directions provided by a mapping service directly into their Web pages, to display their store locations and provide driving directions. But end users have no control over this kind of 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. This is an example of a *mashup*, a combination of data or user interface from more than one site.

These examples involve not only *automating* Web user interfaces (clicking links, filling in forms, and extracting data) but also *customizing* them (changing appearance, rearranging components, and inserting or removing user interface widgets or data). The openness and flexibility of the Web platform enables customizations that would not have been possible on the desktop.

Previous approaches to Web automation used a scripting language that dwells outside the Web browser, such as Perl, Python, or WebL (Kistler & Marais, 1998). 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 an agent running outside the Web browser. Highly interactive Web applications like Google Mail and Google Maps – sometimes called AJAX applications (Garrett, 2005) because they use asynchronous JavaScript and XML – have made this situation worse.

But perhaps the most telling difference, and the most intimidating one to a user, is the simple fact that outside a Web browser, a Web page is just raw HTML. Even the most familiar Web portal can look frighteningly complicated when viewed as HTML source.

Chickenfoot is a programming system we have developed that provides a platform for automating and customizing Web applications through a familiar interface – as Web pages rendered in a Web browser. The challenge for Chickenfoot is simply stated: a user should not have to view the HTML source of a Web page to customize or automate it.

Chickenfoot addresses this challenge in three ways. First, it runs inside the Web browser, so that the rendered view of a Web page is always visible alongside the Chickenfoot development environment. Second, its language primitives are concerned with the Web page's user interface, rather than its internal details. For example, Chickenfoot uses commands like `click`, `enter`, and `pick` to interact with forms. Third, it uses novel pattern matching techniques to allow users to describe components of a Web page (targets for interaction, extraction, insertion, or customization) in terms that make sense for the rendered view. For example, `click` identifies the button to be clicked using keywords from its text



**FIGURE 3.1**

Chickenfoot in action. (a) Chickenfoot sidebar showing a complete script that customizes Google Image Search with a new “Search Icons” button, which (b) automatically fills out the Advanced Search form to (c) display only small GIF images matching the query.

label, rather than the name it was given by the Web page designer. Figure 3.1 shows Chickenfoot in action, with a script for Google Image Search that uses some of these commands.

Chickenfoot is implemented as an extension for the Mozilla Firefox Web browser, written in Java, JavaScript, and XUL. It consists of a development environment, which appears as a sidebar of Firefox, and a library built on top of JavaScript. Chickenfoot customizations are essentially JavaScript programs, so Chickenfoot currently does not support nonprogramming users. We assume that a Chickenfoot user has *some* knowledge of JavaScript and HTML – not an unreasonable assumption, because many power users showed the ability and willingness to learn these during the explosive growth of the Web. The problem Chickenfoot is addressing is not learning JavaScript and HTML syntax, but rather reading and understanding the complex HTML used by today’s Web applications.

Naturally, many users would benefit from a Web automation system that avoids the need to learn programming language syntax. While Chickenfoot is only one step towards this goal, we regard it as a crucial one, since it provides a level of expressiveness and completeness unavailable in special purpose Web automation systems.

One system similar to Chickenfoot in implementation is Greasemonkey (<http://www.greasespot.net>), a Firefox extension that can run user-written JavaScript on Web pages just after they are loaded in the browser. Though Greasemonkey enables mutation and automation of Web pages in the browser, it still requires users to understand the HTML underlying the page. Platypus

(<http://platypus.mozdev.org>) is another Firefox extension, designed to work with Greasemonkey, that allows some customization of rendered Web pages, but not automation or integration of multiple Web sites.

The rest of this chapter is organized as follows. First we give an overview of the Chickenfoot language and development environment, and describe a range of applications we have built using Chickenfoot. Then we delve deeper into a novel aspect of Chickenfoot: the pattern matching used to identify Web page elements for automation or customization. We describe a survey of Web users that motivated the design of the pattern matching, and present the algorithm we developed as a result. Finally we discuss some experience and lessons learned from the Chickenfoot project, review related work, and conclude.

---

## CHICKENFOOT

Chickenfoot is an extension to the Mozilla Firefox Web browser, consisting of a library that extends the browser's built-in JavaScript language with new commands for Web automation, and a development environment that allows Chickenfoot programs to be entered and tested inside the Web browser. This section describes the essential Chickenfoot commands, including pattern matching, form manipulation, page navigation, and page modification. The section concludes by describing the development environment (Figure 3.1a).

### Language

Chickenfoot programs are written in JavaScript, using the JavaScript interpreter built into Mozilla Firefox. As a result, Chickenfoot users have access to the full expressiveness of a high-level scripting language, with a prototype-instance object system, lexically scoped procedures, dynamic typing, and a rich class library.

Because Chickenfoot uses JavaScript, Web developers can transfer their knowledge of JavaScript from Web page development over to Chickenfoot. Chickenfoot predefines the same variables available to JavaScript in Web pages – for example, `window`, `document`, `location`, `frames`, `history` – so that JavaScript code written for inclusion in a Web page can generally be used in a Chickenfoot script that operates on that Web page. JavaScript has its own ways of visiting new pages (`location`), manipulating form controls (`document.forms`), and modifying page content (using the Document Object Model, or DOM). These mechanisms can be used by Chickenfoot scripts in addition to the Chickenfoot commands described in the next few sections. We have found that the native JavaScript mechanisms generally require reading and understanding a Web page's HTML source. But by providing access to them, Chickenfoot provides a smooth escape mechanism for script developers that need to do something more low-level.

### Pattern matching

Pattern matching is a fundamental operation in Chickenfoot. To operate on a Web page component, most commands take a pattern describing that page component.

Chickenfoot supports three kinds of patterns: keywords, XPath, and regular expressions. A keyword pattern consists of a string of keywords that are searched in the page to locate a page component, followed by the type of the component to be found. For example, "Search form" matches a form containing the keyword "Search," and "Go button" matches a button with the word "Go" in its label. The component type is one of a small set of primitive names, including `link`, `button`, `textbox`, `checkbox`, `radiobutton`, `listbox`, `image`, `table`, `row`, `cell`, and `text` (for text nodes). When a keyword pattern is used by a form manipulation command, the type of page component is often implicit and can be omitted. For example, `click("Go")` searches for a hyperlink or button with the keyword "Go" in its label. Case is not significant, so `click("go")` has the same effect. Keyword patterns are one example of *sloppy programming* (see Chapter 15).

An XPath pattern uses the XPath pattern matching syntax to identify nodes in the HTML document. For example, the XPath pattern `"/b"` finds all elements in the page that are tagged with `<b>` (bold); `"/b//a"` finds hyperlinks inside bold elements; and `"/div[@class='box']"` finds `<div>` elements that have the class attribute "box". The XPath syntax is far more powerful and precise than keyword patterns, but harder to learn and use correctly, and often requires close examination of the page's HTML to create, understand, and debug a pattern. XPath patterns are generally used to identify parts of a page for modification or extraction; form manipulation is often easier to do with keyword commands instead.

The third kind of pattern, regular expressions, search the visible text content of the page, using JavaScript's regular expression syntax and semantics. For example, `/[A-Z]\w+/` finds capitalized words in the page. For the purpose of matching, the page content is divided into *paragraph blocks* by hard line breaks (`<br>` elements) and HTML block-level element boundaries. The pattern is applied separately to each paragraph. Thus `/^\w+/` matches words at the start of a block, and `/ISBN:.*` will match only part of a block, not the entire page after "ISBN" first appears.

Chickenfoot's `find` command takes one of these patterns and searches for it in the current page. A keyword pattern is represented simply as a string, whereas XPaths and regular expressions are represented by XPath and RegExp objects:

```
find("Search form") // finds a form with "Search" somewhere in it
find(new XPath("/b//a")) // finds hyperlinks in bold
find(/\w+/) // finds words in the page
```

`find` returns a `Match` object, which represents the first match to the pattern and provides access to the rest of the matches as a JavaScript iterator. Here are some common idioms using `find`:

```
// test whether a pattern matches
if (find(pattern).hasMatch) { ... }
// count number of matches
find(pattern).count
// iterate through all matches
for each (m in find(pattern)) { ... }
```

A `Match` object represents a contiguous region of a Web page, so it also provides properties for extracting that region. If `m` is a `Match` object, then `m.text` returns the plain text it represents, that is, the text that would be obtained by copying that region of the rendered Web page and pasting it to a text editor

that ignored formatting. Similarly, `m.html` returns the source HTML of the region, which is useful for extracting the region with formatting intact. Finally, `m.element` returns the `DOM Element` object represented by the region, if the region contains a single outermost element. This `element` can be used to get and set element attributes, for example: `find("link").element.href`.

The `find` command is not only a global procedure, but also a method of `Match` (returned by a previous `find`) and `Tab` (which identifies a Firefox tab or window). Invoking `find` on one of these objects constrains it to return only matches within the page or part of a page represented by the object. Here are some common idioms:

```
// nested finds
for each (t in find("table")) {
  for each (r in t.find("row")) {
    ...
  }
}
// find in a different tab
otherTab.find(pattern)
```

## Clicking

The `click` command takes a pattern describing a hyperlink or button on the current page and causes the same effect as if the user had clicked on it. For example, these commands click on various parts of the Google home page:

```
click("Advanced Search") // a hyperlink
click("I'm Feeling Lucky") // a button
```

Keyword patterns do not need to exactly match the label of the button or hyperlink, but they do need to be unambiguous. Thus, `click("Lucky")` would suffice to match the “I’m Feeling Lucky” button, but in this case, `click("Search")` would be ambiguous between the “Google Search” button and the “Advanced Search” link, and hence would throw an exception. Exact matches take precedence over partial matches, however, so if there was a single button labeled `Search`, then the `click` command would succeed.

Another way to eliminate ambiguity is by prefixing the pattern with an index, like “first”, “second”, “third”, and so on, so `click("second Search")` will click on the second search button.

When a button or link has no text label, but an image instead, then the search can use keywords mentioned in the image’s `alt` and `title` attributes, if any. The keyword matching algorithm is described in more detail later in this chapter.

The `click` command can take a `Match` object instead of a pattern, if the button or hyperlink to be clicked has already been found. For example, to identify the “I’m Feeling Lucky” button using an XPath pattern, the user might write something like this:

```
click(new XPath("//input[@name='btnI']"))
```

This example illustrates why XPath is generally less readable than keywords.



## Form manipulation

The `enter` command enters a value into a textbox. Like `click`, it takes a pattern to identify the textbox, but in this case, the keywords are taken from the textbox's caption or other visible labels near the textbox. For example, to interact with the Amazon login page, a script might say:

```
enter("e-mail address", "rcm@mit.edu")
enter("password", password)
```

When the page contains only one textbox, which is often true for search forms, the keyword pattern can be omitted. For example, this sequence does a search on Google:

```
enter("EUP book")
click("Google Search")
```

Checkboxes and radio buttons are controlled by the `check` and `uncheck` commands, which take a keyword pattern that describes the checkbox:

```
check("Yes, I have a password")
uncheck("Remember Me")
```

The `pick` command makes a selection from a listbox or drop-down box (which are both instantiations of the HTML `<select>` element). The simplest form of `pick` merely identifies the choice by a keyword pattern:

```
pick("California")
```

If only one choice in any listbox or drop-down on the page matches the keywords (the common case), then that choice is made. If the choice is not unique, `pick` can take two keyword patterns, the first identifying a listbox or drop-down by keywords from its caption, and the second identifying the choice within the listbox:

```
pick("State", "California")
```

Like `find`, all the clicking and form manipulation commands are also methods of `Match` and `Document`, so that the keyword search can be constrained to a particular part of a page:

```
f = find("Directions form")

f.enter("address", "32 Vassar St")
f.enter("zip", "02139")
f.click("Get Directions")
```

The form manipulation commands described so far permit setting the value of a form widget, which is the most common case for Web automation. To read the current value of a widget, a script can use `find` to locate it, and then access the value of its `Element` object in conventional JavaScript fashion, for example:

```
find("address textbox").element.value
```

Chickenfoot also provides a `reset` command, which resets a form to its default values, though `reset` is rarely needed.

## Other input

Chickenfoot can also simulate more low-level mouse and keyboard input. The `click` command can simulate a mouse click on any page component. By default it searches only through buttons and hyperlinks, but this default can be overridden by specifying a component type. For example, instead of using `check` or `uncheck` to set a checkbox, one can toggle it with `click`:

```
click("Remember Me checkbox")
```

Similarly, the `keypress` command sends low-level keystrokes to a particular page component. Unlike `enter`, which takes a string of text to fill a textbox, `keypress` takes a sequence of keywords describing individual keypresses. Valid keywords include characters (a, A, %, etc.), key names (`enter`, `f1`, `home`, `tab`, `backspace`, etc.), and modifier keys (`shift`, `ctrl`, `control`, etc.). The key sequence is interpreted from left to right, and modifier keys affect the first character or key name that follows them; for example, `"ctrl x c"` produces Control-X followed by “c”. Examples include:

```
keypress("password", "enter") // presses Enter on the password textbox
keypress("ctrl s") // sends Ctrl-S to the only textbox on the page
keypress("E U P space b o o k") // types "EUP book" in the only textbox
```

The last `keypress` command above is almost equivalent to `enter("EUP book")`, except that `enter` discards the old contents of the textbox, if any, whereas `keypress` inserts the keystrokes wherever the textbox’s selection happens to be.

## Navigation and page loading

In addition to loading pages by clicking on links and submitting forms, Chickenfoot scripts can jump directly to a URL using the `go` command:

```
go("http://www.google.com")
```

If the string is not a valid URL, `go` automatically adds the prefix “http://”.

Conventional browser navigation operations are also available as Chickenfoot commands: `back`, `forward`, and `reload`.

The `openTab` command opens a page in a new tab, returning a `Tab` object representing the tab:

```
google = openTab("www.google.com")
```

To retrieve a page without displaying it, the `fetch` command can be used. It returns a `Tab` object representing the invisible page:

```
google = fetch("www.google.com")
```

The JavaScript `with` statement is convenient for performing a sequence of operations on a different tab, by implicitly setting the context for Chickenfoot pattern matching and form manipulation:

```
with (fetch("www.google.com")) {
  enter("syzygy")
  click("Google Search")
}
```

A tab created with `openTab` and `fetch` can be brought to the front by calling its `show` method and closed with its `close` method. The `Tab` object representing the current tab is stored in a read-only variable, `tab`.

Pages retrieved by `go`, `openTab`, `fetch`, and `click` are loaded asynchronously by the browser, while the Chickenfoot script continues to run. Thus, a script can fire off several `fetch` requests in parallel, without forcing each request to complete before the next one starts. When a subsequent Chickenfoot command needs to access the content of a page, such as `find`, the command automatically blocks until the page is fully loaded. The `wait` and `ready` commands make this blocking available to programmatic control. Both commands take a `Tab` object or an array of `Tabs` as an argument. With no arguments, the default is the current tab. `wait` blocks until at least one of the specified tabs is fully loaded, and returns that tab. `ready` returns a loaded tab only if it has already completed, otherwise it immediately returns `null`.

## Page modification

Chickenfoot offers three primitive commands for changing the content of Web pages: `insert`, `remove`, and `replace`.

The `insert` command takes two arguments: a location on a page and a fragment of Web page content that should be inserted at that location. In its simplest form, the location is a keyword pattern, and the Web page content is simply a string of HTML. Since insertion must use a single point in the page, not a range of content, the `before` or `after` commands must be used to reduce the pattern match to a point:

```
insert(before("textbox"), "<b>Search: </b>")
```

The location can also be a `Match` object:

```
t = find("textbox")
insert(after(t), "<b>Search: </b>")
```

The page content to be inserted can also be a `Match` or `Node`, allowing content to be extracted from another page and inserted in this one:

```
map = mapquest.find("image") // where mapquest is a tab showing a map
insert(after("Directions"), map)
```

The `remove` command removes page content identified by its argument, which can be a pattern or `Match` object. For example:

```
remove("Google image")
```

The `replace` command replaces one chunk of page content with another. It is often used to wrap page content around an existing element. For example, the following code puts every number in the page in boldface:

```
for each (num in find(/\d+/)) {
  replace(num, "<b>"+num.text+"</b>")
}
```

## Widgets

When a Chickenfoot script needs to present a user interface, it can create links and buttons and insert them directly into a Web page. Hyperlinks are created by the `Link` constructor, which takes a chunk of HTML to display inside the hyperlink and an event handler to run when the link is clicked:

```
new Link("<b>Show All</b>", showAll)
```

The event handler should be a JavaScript function object. Buttons are created similarly by the `Button` constructor.

Other widgets can be created by inserting HTML, e.g.:

```
insert(..., "<input type=checkbox>")
```

If an `onclick` attribute is included in this HTML element, however, the code it contains will execute like conventional, untrusted JavaScript code downloaded with the page. Commands defined by Chickenfoot would be unavailable to it. Instead, the right way to add a Chickenfoot handler to a button is to use the `onClick` command:

```
onClick(button, showAll)
```

Keyboard handlers can also be attached using the `onkeypress` command, which takes an optional page component, a key sequence (specified using words as in the `keypress` command) and a handler function:

```
onkeypress("ctrl s", saveWikiPage)
onkeypress("username textbox", "tab", fillInPassword)
```

## Development environment

Figure 3.2 shows a screenshot of the development environment provided by Chickenfoot, which appears as a sidebar in Firefox. At the top of the sidebar is a text editor used to compose Chickenfoot code. The editor supports multiple tabs editing different script files. This simple interface goes a long way toward making the Web browser's JavaScript interpreter more accessible to the 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 cannot edit it), or by encoding it as a *bookmarklet*, in which the entire program is packed into a one-line URL. The Chickenfoot sidebar makes it much easier for an end user to write and run scripts. Recent Web debugging tools, such as Firebug (<http://getfirebug.com>), now include multiline script editors for a similar reason.

The bottom of the sidebar has three tabbed panels. The Output panel is an output window that displays output printed by the Chickenfoot `output` command, error messages and stack traces, and the result of evaluating a Chickenfoot script (i.e., the value of the last expression). If the "Record Actions" checkbox is enabled, then the Output panel also displays Chickenfoot commands corresponding to the manual actions that the user does in the browser, like `click("Search button")` or `enter("Username", "rcm")`. This feature enables some degree of self-disclosure (DiGiano & Eisenberg, 1995). A manual browsing sequence can be used as the basis for a script by recording it into the Output panel and copying it into a script.

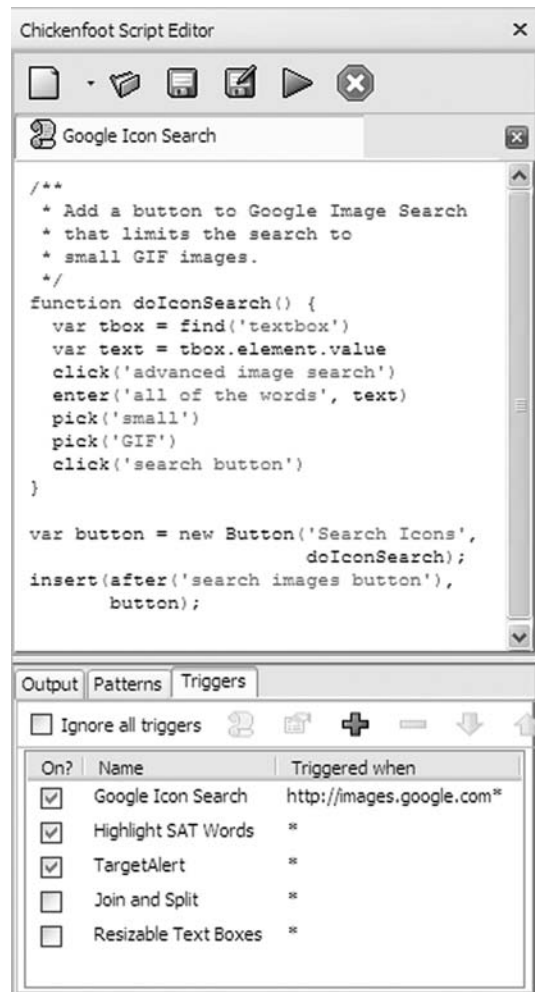
The Patterns panel displays an interface for developing patterns, which allows the user to enter a pattern and highlight what it matches in the current page. This panel also displays the types of page components that can be used for keyword commands (link, button, textbox, etc.).

The Triggers panel allows a Chickenfoot script to be installed into the browser for regular use. For automatic invocation, a script can be associated with a *trigger*, which is a URL pattern, such as `http://www.amazon.com/*`. Whenever a page is loaded, if its URL matches a trigger, then the associated script executes automatically. If a page matches multiple triggers, the associated scripts execute in the fixed order given by the Triggers panel. Trigger scripts can also be invoked manually by right-clicking on a page and selecting “Run Trigger on This Page”. The Triggers panel provides an interface for adding and removing trigger scripts and enabling and disabling triggers.

Finally, the Triggers panel provides a “Package Script as Firefox Extension” command, which binds one or more trigger scripts and supporting files, together with an embedded version of the Chickenfoot library, and exports the result as a Firefox extension (XPI) file. Any Firefox user can install this extension in their browser to use the triggers without having to install Chickenfoot first. Thus, customizations developed in Chickenfoot can be easily exported as full-fledged Firefox extensions.

## Security

All current Web browsers, including Firefox, implement a security model for JavaScript to protect Web users from malicious downloaded scripts. A major part of this security model is the *same-origin policy*, which prevents JavaScript code downloaded from one Web server from manipulating a Web page downloaded from a different server. This restriction is clearly too severe for Chickenfoot, because its primary purpose is integrating and customizing multiple Web sites. As a result, Chickenfoot scripts run at a privileged level, where they have access to the entire Web browser, all pages it



**FIGURE 3.2**  
Chickenfoot sidebar.

visits, and the user's filesystem and network. Users must trust Chickenfoot code as much as they trust any other desktop application. As a result, Chickenfoot scripts cannot be embedded in downloadable Web pages like other JavaScript. But Chickenfoot code *can* inject new behavior into downloaded pages, for example, by inserting new widgets and attaching event handlers to them.

---

## KEYWORD MATCHING ALGORITHM

One of the novel aspects of Chickenfoot is the use of keyword patterns to identify page elements, such as "Search button" and "address textbox". A heuristic algorithm resolves a keyword pattern to a Web page component. Given a name and a Web page, the output of the algorithm is one of the following: (1) a component on the page that best matches that name, (2) *ambiguous match* if two or more components are considered equally good matches, or (3) *no match* if no suitable match can be found.

Some components, like buttons and hyperlinks, have a label directly associated with the component, which makes the keyword search straightforward. Other components, like textboxes, checkboxes, and radio buttons, typically have a label appearing nearby but not explicitly associated with the component, so Chickenfoot must search for it. (HTML supports a <label> element that explicitly associates labels with components like these, largely to help screen reader software, but this element is unfortunately rarely used by Web sites.) This section outlines the algorithm for locating a textbox given a keyword pattern. Other components are handled similarly, but with different heuristics for finding and ranking labels.

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>, <br>, <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 extra white space. Then each label is searched for keywords occurring in the name. 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 that it might identify. 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. The first heuristic 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, because 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, because these are unusual places for a caption. (We do not completely rule them out, however, 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*) pair is scored by the length of the shortest path through the DOM tree from the *label* node to the *textbox* node. 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*.

---

## USER STUDY OF KEYWORD MATCHING

To explore the usability of the keyword pattern matching technique, we conducted a small study to learn what kinds of keyword patterns users would generate for one kind of page component (textboxes), and whether users could comprehend a keyword pattern by locating the textbox it was meant to identify.

### Method

The study was administered over the Web. It consisted of three parts, always in the same sequence. Part 1 explored free-form 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 study 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 3.3a), wordy captions (Figure 3.3b), captions displayed as default values for the textbox (Figure 3.3c), and missing captions (Figure 3.3d). 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 participated (20 females, 20 males), including both programmers and nonprogrammers (24 reported their programming experience as “some” or “lots,” while 15 reported it as “little” or “none,” meaning at most one programming class). All but one subject were experienced Web users, reporting that they used the Web at least several times a week.

### 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 3.3a); (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

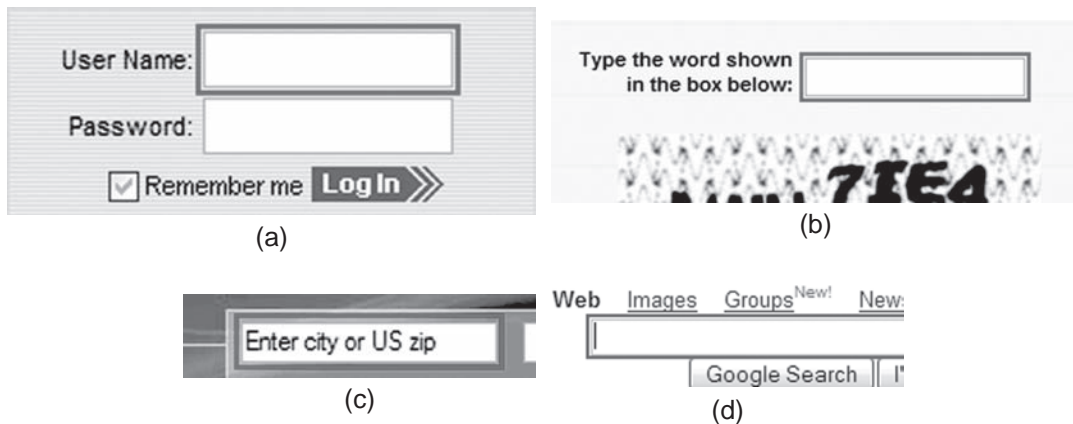


FIGURE 3.3

Some of the textboxes used in the Web survey.

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 because the study 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 3.3a 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 3.3b 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 3.3d were visible).

Part 2 found that users could flawlessly find the textbox associated with a visible name, as long as 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 that 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”).

Observations from the user study motivated several features of Chickenfoot’s pattern matching. Chickenfoot does not insist on exact matches between patterns and captions, tolerating missing words, rearrangements, and capitalization changes. Chickenfoot also supports counting for disambiguation, such as using “search 2” or “second search box”.

---

## APPLICATIONS

This section describes a few of the example scripts we have created using Chickenfoot. These examples cover the three types of Web page customization discussed in the introduction (*transforming*, *automating*, and *integrating*), and some examples have aspects of two or more types.

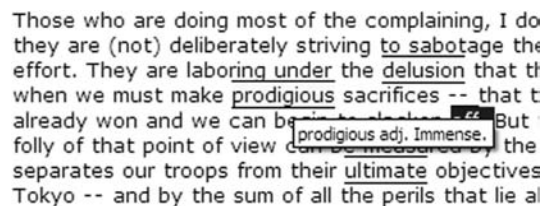
### Highlighting vocabulary words

We start with a simple example that *transforms* the presentation of Web pages. Students studying for college placement exams, such as the SAT, often work hard to expand their vocabulary. One way to make this learning deeper is to highlight vocabulary words while the student is reading, so that the context of use reinforces the word’s meaning. One of our Chickenfoot scripts takes a list of vocabulary words and definitions (posted on the Web) and automatically highlights matching words in any page that the user browses (see Figure 3.4). The script uses a `title` attribute to pop up the word’s definition as a tooltip if the mouse hovers over it:

```
for each (w=find(/\w+/)) {
  if (w.text in vocab) {
    html = "<span style='background-color: yellow' title='"
          + vocab[w.text] + ">"
          + w + "</span>"
    replace(w, html)
  }
}
```

### Sorting tables

A feature that some Web sites have, but many lack, is the ability to sort a table of data by clicking one of its column headers. A Chickenfoot script can add this functionality automatically to most tables, by replacing every table header cell it finds with a link that sorts the table. This is another example of a *transforming* customization.



Those who are doing most of the complaining, I do they are (not) deliberately striving to sabotage the effort. They are laboring under the delusion that it when we must make prodigious sacrifices -- that t already won and we can be prodigious. But folly of that point of view separates our troops from their ultimate objectives Tokyo -- and by the sum of all the perils that lie al

**FIGURE 3.4**

Vocabulary word highlighting.

Most of the script is concerned with managing the sort, but here is the part that replaces headers with links:

```
column=0
for each (h in table.find(new XPath("//th//@text"))) {
    f = makeRowSorter(table, column++)
    replace(h, new Link(h, f))
}
```

The `makeRowSorter` function returns a function that sorts the specified table by the specified column number.

### Concatenating a sequence of pages

Search results and long articles are often split into multiple Web pages, mainly for faster downloading. This can inhibit fluid browsing, however, because the entire content isn't accessible to scrolling or to the browser's internal Find command. Some articles offer a link to the complete content, intended for printing, but this page may lack other useful navigation.

We have written a Chickenfoot script that detects a multipage sequence by searching for its table of contents (generally a set of numbered page links, with "Next" and "Previous"). When a table of contents is found, the script automatically adds a "Show All" link to it (Figure 3.5). Clicking this link causes the script to start retrieving additional pages from the sequence, appending them to the current page. In order to avoid repeating common elements from subsequent pages (such as banners, sidebars, and other decoration), the script uses a conservative heuristic to localize the content, by searching for the smallest HTML element that contains the list of page links (since the content is nearly always adjacent to this list) and spans at least half the height of the rendered page (since the content nearly always occupies the majority of the page). The content element from each subsequent page is inserted after the content element of the current page.

In terms of types of Web customization, this example not only *transforms* the page (by inserting additional content into it), but also *automates* it (by clicking on "Next" links on behalf of the user).

### Integrating a bookstore and a library

An example of *integrating* two Web sites is a short script that augments book pages found in Amazon with a link that points to the book's location in the MIT library:

```
isbn = find(/ISBN.*:*(\d+)/).range[1]
with (fetch("libraries.mit.edu")) {
```



FIGURE 3.5

"Show All" link injected for page concatenation.

```

// run on MIT library site
pick("Keyword")
enter(isbn)
click("Search")
link = find("Stacks link")
}
// now back to Amazon
if (link.hasMatch) {
    insert(before("Available for in-store pickup"), link.html)
}

```

Figure 3.6 shows the result of running this script.

### Making a bug tracker more dynamic

For issue tracking in the Chickenfoot project, we use the Flyspray bug tracker (<http://flyspray.org>), which has the advantage of being free and simple. One downside of Flyspray is that some of its interfaces are inefficient, requiring several clicks and form submissions to use. For example, when we meet to discuss open bugs, we often use Flyspray's list view (Figure 3.7a), which displays bugs in a compact tabular form. But changing a bug's properties from this view – particularly, its priority, severity, or assigned developer – requires five clicks and waiting for three different pages to load. What we want instead is direct manipulation of the list view, so that bug properties can be edited immediately without leaving the view.

ID	Category	Severity	Priority	Summary
495	Bug	High	Immediate	can't click on javascript: links
178	Usability	Low	High	Open File should default to Chickenfoot profile directory.
219	Documentation	Medium	High	Examples on the web site are broken
478	Documentation	Low	High	SAT word example is broken
493	New feature	Low	High	dynamically-scoped withTab
500	Bug	Critical	Immediate	FF2 native libraries break 64-bit Ubuntu
476	Platform bug/Plugin conflict	High	Urgent	Load ChickenSleep manually
411	Usability	Medium	Normal	Recorder should be on all the time
412	Bug	Medium	Normal	Recorder records lots of spurious commands

(a)

FIGURE 3.7a

Flyspray bug summary display.



FIGURE 3.6

A link to a local library injected into a bookstore Web page.

495	Bug	High	Immediate	can't click on javascript: links
178	Usability	Low	High	Open File should default to Chickenfoot profile directory.
219	Documentation	Medium	Flash	Examples on the web site are broken
478	Documentation	Low	Immediate	SAT word example is broken
493	New feature	Low	Urgent	dynamically-scoped withTab
500	Bug	Critical	High	FF2 native libraries break 64-bit Ubuntu
476	Platform bug/Plugin conflict	High	Normal	Load ChickenSleep manually
411	Usability	Medium	Low	Recorder should be on all the time
			Urgent	
			Normal	

(b)

**FIGURE 3.7b**

Flyspray augmented by a Chickenfoot script so that bug properties can be edited in place.

We added this behavior to Flyspray using a Chickenfoot script, which replaces values in the table with a drop-down menu (Figure 3.7b). The script automatically discovers the available choices for each drop-down menu by opening the edit page for some bug (invisibly, using `fetch`) and extracting the HTML for each of its drop-down menus:

```

firstBug = find(new XPath("//td[@class='project_summary']/a"))

with (fetch(firstBug.element.href)) {
  click("Edit this task")
  priorityMenu = find("Priority listbox").html
  severityMenu = find("Severity listbox").html
  assignedToMenu = find("Assigned To listbox").html
  ... // for other fields
  close() // closes the fetched edit page
}

```

The script then attaches an `onClick` handler to each cell of the table, so that when a cell is clicked, the value is replaced with the appropriate drop-down menu for that column (Figure 3.7b). If the user changes the value of the property using the drop-down menu, then the script drills into the edit page for that bug (again, invisibly) and changes it:

```

with(fetch(href)) {
  click("Edit this task")
  pick("Priority", newValue)
  click("Save Details")
  wait() // make sure changes are saved before closing tab
  close()
}

```

The overall effect of this script is to make Flyspray, a traditional Web application driven entirely by form submission, feel more like a modern AJAX-driven Web application, which responds immediately and updates the server in the background. This example not only *transforms* Flyspray's interface, by adding drop-down menus, but also *automates* it when a drop-down menu is changed.

It's also worth considering why we used Chickenfoot in this case. Unlike the previous examples, which modified third-party Web sites, we control our own installation of Flyspray, which is an open source system written in PHP. Technically, we could modify Flyspray ourselves to add this feature. The main reason we chose not to was the complexity barrier of understanding and using Flyspray's internal interfaces, which we had never seen since we did not originally develop it. With Chickenfoot, however, the interface we needed to understand and use was the Flyspray *user* interface, which was simple and already very familiar to us as users of Flyspray. Writing the Chickenfoot script took less than an hour; delving into Flyspray and making it AJAX-capable would have been considerably longer.

### Adding faces to Webmail

Usable security is an active area of research that aims to make computer security easier to use, and use correctly, for end users (Cranor & Garfinkel, 2005). One interesting and underexplored problem in this area is *misdirected email*, such as pressing "Reply All" when you meant to press "Reply", or mistyping an email address. Sending email to the wrong recipients is an access control error, since it inadvertently grants access to the email's contents to a different group of people than intended. One example of this error made headlines when a lawyer for the drug company Eli Lilly, intending to send a confidential email about settlement discussions to another lawyer, accidentally sent the message to a *New York Times* reporter who happened to have the same last name as the intended recipient (NPR, 2008), resulting in a front-page story (Berenson, 2008). Note that even secure email suffers from this error, because a digitally signed and encrypted message can be sent, securely, to the wrong people.

Using Chickenfoot, we have devised a novel solution to this problem that displays the faces of the recipients directly in the mail composition window of a Webmail system. The faces appear automatically as the user is composing the message, acting as a peripheral interface that informs the user without demanding direct attention. Figure 3.8 shows our interface, called Kangaroo, extending the IMP Webmail system. Our studies have found that the face display makes a substantial difference in users' ability to detect whether an email is misdirected (Lieberman & Miller, 2007).

Kangaroo is implemented as a collection of Chickenfoot scripts. One script triggers whenever a mail composition page appears, to automatically add the Face section to the page. This script triggers on URLs used by Gmail and IMP (the Webmail system MIT uses), but it could be readily extended to other Webmail systems. The script also attaches event handlers to the To, Cc, and Bcc boxes on the page, so that Kangaroo is notified when the user edits the recipient list. The same simple keyword patterns (e.g., "To textbox") are sufficient to identify these controls across the three Webmail systems Kangaroo currently supports, so the porting costs to new Webmail systems are likely to be minimal.

A second set of scripts in Kangaroo are concerned with finding faces to display. The database that maps between email addresses and faces is cached locally in the user's Web browser, rather than at a server. This database is populated automatically by Chickenfoot scripts that automate searches of various Web services, including Google Images and Facebook. Another set of scripts expands mailing list addresses into their subscribers wherever possible, using Web interfaces for Mailman and Athena Blanche (a mailing list system used internally at MIT). New scripts can be

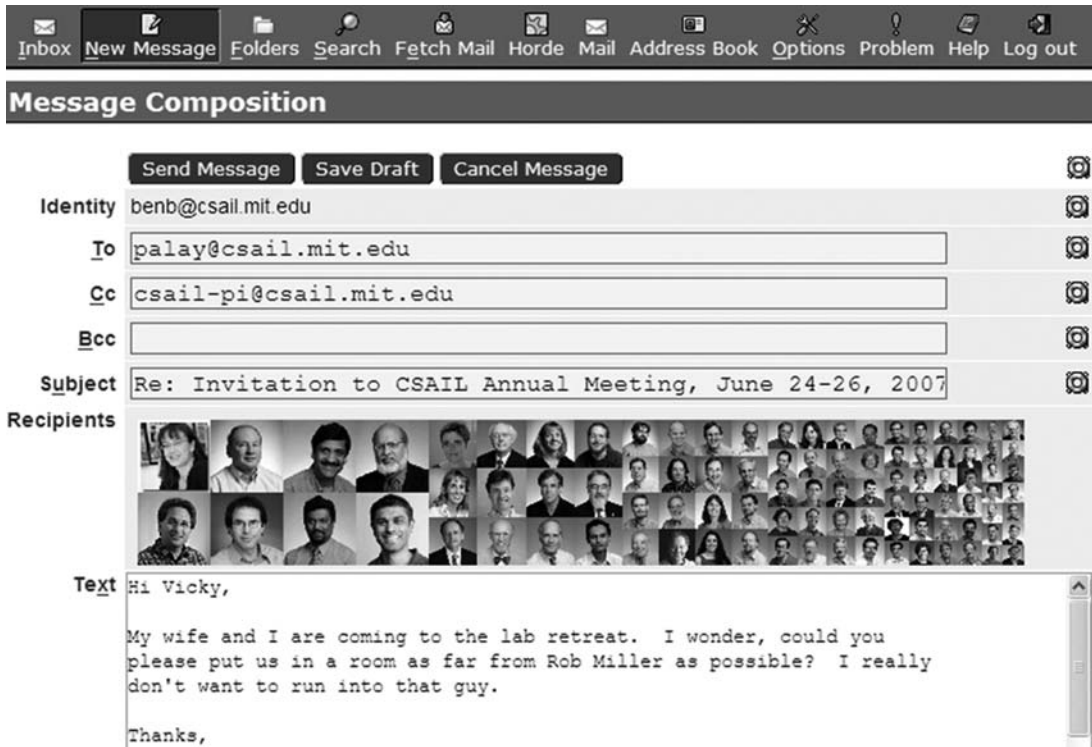


FIGURE 3.8

Kangaroo recipient-visualization interface, augmenting the IMP Webmail system. In this example, the user pressed “Reply All” by mistake, so the crowd of faces is a highly salient reminder about who will read the message.

added to support new sources of faces and new kinds of mailing lists. For example, to use Kangaroo in a corporate email environment, the corporate ID database and mailing list server could readily be connected.

Kangaroo is an example of a substantial and interesting application built on top of Chickenfoot that embodies several of Chickenfoot’s key ideas: *transforming* an existing Web application (in this case, Gmail), *automating* and *integrating* multiple Web applications (like Mailman and Facebook), and programming at the level of *rendered user interfaces* (“To `textbox`”) rather than application programming interfaces (APIs). We also use Chickenfoot’s trigger-packaging functionality to make Kangaroo downloadable as a standalone Firefox extension (<http://groups.csail.mit.edu/uid/kangaroo/>).

In addition to Kangaroo, our research group has used Chickenfoot as a prototyping tool for several other research systems, including minimal-syntax programming (see Chapter 15), bookmarks for modern AJAX sites (Hupp & Miller, 2007), a command line for Web applications (Miller et al., 2008), and a tool that improves page readability for nonnative English readers. Chickenfoot has proven to be a flexible and useful tool for exploring new ideas in Web user interface.

---

## DISCUSSION

This section reflects on some of the lessons learned and surprises encountered over several years of developing and using Chickenfoot, and mentions some of the open areas of future work.

A guiding design principle, mentioned in the introduction of this chapter, was that Chickenfoot users should not have to look at the HTML source of a Web page to automate and customize it. This is a hard goal to achieve, and Chickenfoot only gets some of the way there. Some success can be seen in the Kangaroo system mentioned earlier, which uses keyword patterns effectively to customize Webmail systems in a portable way. But in other cases, getting a good, unambiguous keyword pattern for some Web page controls can be very hard, so users fall back on less robust counting patterns (“third textbox”) or XPath. It’s not uncommon for users to use a Web page inspector, like Firebug, to help analyze and understand the Web page, and then take what they learned to write a Chickenfoot script.

An aspect of Chickenfoot that turned out to be much more important, however, was the *synchronous, page-independent* nature of Chickenfoot scripts, relative to ordinary JavaScript or Greasemonkey. A Chickenfoot script can be written as a simple, straight-line sequence of commands, even if those commands end up affecting different pages. The Chickenfoot interpreter automatically handles waiting for pages to load, which substantially reduces the complexity burden on the programmer. In ordinary JavaScript, this simple script might need to be torn apart into many little pieces in event handlers.

Although originally designed for end user customization of Web applications, Chickenfoot has been put to a few uses that we never anticipated, but that turn out to be very fruitful. One use already mentioned is as a testbed for our own research prototypes. Some of these prototypes customize the *browser itself* with new toolbar buttons or menu items, not just changing the Web sites it displays. This is made possible by two incidental facts: first, Firefox’s user interface is mostly written in XUL (an XML-based user interface language) and JavaScript, which is very similar to the HTML/JavaScript model used in Web sites; and second, Chickenfoot scripts run with the same privileges as browser-level JavaScript, so they have permission to modify XUL and attach event listeners to it. Thus Chickenfoot can be used to prototype full-fledged Firefox extensions. One area of ongoing work is to generalize Chickenfoot’s library of commands and patterns so that it works as well on XUL as it does on HTML.

Another unexpected but popular use of Chickenfoot on the Web is for functional Web testing, that is, scripts that exercise various functions of a Web application to check that they work correctly. Unlike the scenario Chickenfoot was envisioned for, in which end users customize Web sites they did not themselves create, functional Web testing is done by the Web site’s own developers, who have deep knowledge of the site and its internal interfaces. Chickenfoot turns out to be a useful tool for these users nevertheless, for several reasons. First, it is based on JavaScript, so it was familiar to them, and it could directly access and call JavaScript interfaces inside the Web page. Second, a testing script written at the level of the rendered interface (e.g., click “search button”) is decoupled from the internal implementation of that interface, allowing the internal interfaces to change without having to change the testing code. And finally, Chickenfoot’s synchronous programming model, which automatically blocks the script while pages load, makes it easy to write test scripts.

## Future work

Chickenfoot's current commands and pattern language seem well designed for *automation*, but are still weak for *data extraction*, also called Web scraping. The keyword pattern approach should be applicable here, for identifying captioned data (like "List Price: \$27.95") and tabular data with labeled rows and columns. Currently, most data extraction in Chickenfoot is done with XPaths or regular expressions.

Complex or large Web pages are hard to automate efficiently and effectively with Chickenfoot. For example, nested `<iframe>` elements (which embed other HTML documents into a Web page) cause problems for XPath pattern matching, because Chickenfoot uses the XPath matcher built into Firefox, which expects only one tree with one root, and which cannot dig into an `<iframe>`. For keyword patterns and regular expressions, Chickenfoot searches all embedded documents, but then the problem becomes efficiency. Chickenfoot is written in JavaScript, and traversing large DOM trees is very slow using Firefox's current JavaScript engine. But browser developers are devoting significant resources to improving the performance of JavaScript and DOM manipulation (Paul, 2008), since they are so critical to modern Web applications, so we can anticipate that this situation will improve in the future.

A final question concerns security. Can a malicious Web site do harm to a user who is using Chickenfoot scripts? Can a malicious Chickenfoot script be installed, possibly as a Trojan horse that pretends to do something useful? The answer to both questions is probably yes, although no cases of either have been observed in the wild so far. Chickenfoot's design includes some decisions aimed at mitigating security risks. For example, Chickenfoot code cannot be embedded in Web pages like ordinary JavaScript, and Chickenfoot puts none of its own objects or interfaces into the namespace seen by ordinary Web page JavaScript. For comparison, Greasemonkey's architecture had to be rewritten to fix security holes related to the fact that Greasemonkey scripts were injected into the same namespace as the remote, untrusted JavaScript (Pilgrim, 2005).

Chickenfoot also benefits from features of Firefox designed to make general Firefox extension development safer. For example, when a Chickenfoot script obtains a reference to an object from the Web page (like the Document object, or Window, or a Node), the reference is hidden by a security wrapper called XPCNativeWrapper, which protects against certain kinds of attacks. But Chickenfoot code runs with full privileges, like other Firefox extensions or indeed like any executable run on the desktop, so it should be subjected to a high level of scrutiny. A thorough and detailed security audit of Chickenfoot remains to be done.

---

## RELATED WORK

Several systems have addressed specific tasks in Web automation and customization, including building custom portals (Sugiura & Koseki, 1998), crawling Web sites (Miller & Bharat, 1998), and making multiple alternative queries (Fujima et al., 2004a; Bigham et al., 2008). Chickenfoot is a more general toolkit for Web automation and customization, which can address these tasks and others as well.

One form of general Web automation can be found in scripting language libraries, such as Perl's WWW::Mechanize or Python's ClientForm. These kinds of scripts run outside the Web browser,



where they cannot easily access pages that require session identifiers, secure logins, cookie state, or client-side JavaScript to run.

In an attempt to access these “hard-to-reach pages” (Anupam et al., 2000), some systems give the user the ability to record macros in the Web browser, where the user records the actions taken to require access to a particular page, such as filling out forms and clicking on links. Later, the user can play the macro back to automate access to the same page. LiveAgent (Krulwich, 1997) takes this approach, recording macros with a proxy that sits between the user’s browser and the Web. The proxy augments pages with hidden frames and event handlers to capture the user’s input, and uses this information to play back the recording later. Unfortunately, the proxy approach is also limited – for example, pages viewed over a secure connection cannot be seen, or automated, by the proxy. WebVCR (Anupam et al., 2000) is another macro recorder for Web navigation, which skirts the proxy problem by using a signed Java applet to detect page loads and instrumenting the page with event-capturing JavaScript after the page loads. Because part of WebVCR runs as an applet inside the browser, it can record all types of navigation. But neither LiveAgent nor WebVCR enable the user to modify the pages being viewed.

CoScripter (see Chapter 5) also uses the macro recording model, but unlike these earlier systems, CoScripter runs directly in the browser as a Firefox extension. In addition, recorded CoScripter scripts are displayed to the user as a list of actions that can be read and edited by humans, such as “click on the Google Search button.” CoScripter executes these instructions in a manner similar to Chickenfoot keyword pattern matching.

Toolkits such as WBI (Barrett, Maglio, & Kellem, 1997) and Greasemonkey focus on giving the user the ability to modify pages before, or just after, they are loaded in the user’s Web browser. WBI uses a proxy to intercept page requests, letting user-authored Java code mutate either the request or the resulting page. Giving users the ability to automate pages with Java and all its libraries is a powerful tool; however, WBI is still hampered by the limitations of a proxy.

Though both WBI and Greasemonkey enable the user to mutate pages, neither of them eliminates the need to inspect the HTML of the page to mutate it. For example, the sample scripts on the Greasemonkey site are full of XPath patterns that identify locations in Web pages. These scripts are difficult to create because they require the author to plumb through potentially messy HTML to find the XPath, and they are difficult to maintain because they are not resilient to changes in the Web site. Chickenfoot avoids this problem by giving users a high-level pattern matching language based on keyword matching that enables the user to identify pages without knowledge of the page’s HTML structure, facilitating development and increasing readability and robustness. In Chickenfoot, users can fall back on XPath expressions to select elements and JavaScript to manipulate a page’s DOM, but they are not restricted to these low-level tools.

WebL (Kistler & Marais, 1998), a programming language for the Web, also focused on giving users a higher-level language to describe Web page elements. In WebL, the user provides names of HTML elements to create *piece-sets*, where a piece-set is a set of *piece* objects, and a piece is a contiguous text region in a document. WebL provides various methods to combine piece-sets called *operators*, including set operators such as `union` and `intersection`, positional operators such as `before` and `after`, and hierarchical operators such as `in` and `contain`. Although these operators help produce more readable scripts, the language does not eliminate the need to inspect a Web page for the names of its HTML elements, as the user must provide those to construct the basic pieces on which the operators work.

Another drawback of WebL, and of most of the aforementioned tools (with the exception of the macro recorders), is that they do not allow scripts to be developed inside the Web browser. We consider the ability to experiment with a Web site from the script development environment one of the greatest advantages of Chickenfoot – the user does not have to wait to see how it will affect the appearance of the Web page, because Chickenfoot gives immediate feedback on the rendered page. LAPIS (Miller & Myers, 2000), a predecessor of Chickenfoot, took a similar approach, giving the user an interactive environment in which to experiment with pattern matching and Web automation. Unfortunately, the LAPIS Web browser does not support Web standards like JavaScript, cookies, and secure connections, so it fails to provide the user with a complete Web experience.

In the time since Chickenfoot was originally released, many new systems have appeared for creating mashups between Web services. Examples include Marmite (Wong & Hong, 2006), Web Summaries (see Chapter 12), Intel MashMaker (see Chapter 9), Yahoo Pipes (<http://pipes.yahoo.com/pipes/>), and Vegemite (Lin et al., 2009). Most of these systems are themselves Web services, rather than browser extensions, so they have less access to the user's own data and less ability to transform the user's experience in small or personalized ways, like adding a bit of data, a keyboard shortcut, or a new link into an existing Web page. These mashup systems tend to have much richer support than Chickenfoot for data extraction and manipulation, however.

## SUMMARY

Chickenfoot is a programming system for Web automation, integrated into the Firefox Web browser. Chickenfoot enables the user to customize and automate Web pages without inspecting their source, using keyword pattern matching to name page components. We showed that keyword patterns correspond closely to the names users actually generate for page components, and we presented a heuristic algorithm that implements keyword matching.

As of 2009, Chickenfoot is still being actively developed and used. The latest version is available at <http://groups.csail.mit.edu/uid/chickenfoot/>.

---

## Acknowledgments

We thank all the students who have worked on Chickenfoot over the years, including Michael Bernstein, Prannay Budhraj, Vikki Chou, Mike Fitzgerald, Roger Hanna, Darris Hupp, Eric Lieberman, Mel Medlock, Brandon Pung, Philip Rha, Jon Stritar, Kevin Su, and Tom Wilson, as well as other members of the UID group who provided valuable feedback on the ideas in this chapter, including ChongMeng Chow, Maya Dobuzhskaya, David Huynh, Marcos Ojeda, and Vineet Sinha. This work was supported in part by the National Science Foundation under award number IIS-0447800, and by Quanta Computer as part of the T-Party project. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the sponsors.

This chapter is based on a paper that originally appeared as “Automation and Customization of Rendered Web Pages,” in Proceedings of the 18th Annual ACM Symposium on User Interface Software and Technology (UIST 2005), © ACM, 2005. See <http://doi.acm.org/10.1145/1095034.1095062>.

## CHICKENFOOT

Intended users:	Script programmers
Domain:	All Web sites
Description:	Chickenfoot is a system for automating and customizing Web sites by writing scripts that run inside the Web browser. The command language and pattern language are designed to work at the level of the Web site's rendered user interface, rather than requiring knowledge of the low-level HTML.
Example:	Before buying a book from an online bookstore, a user may want to know whether it is available in the local library. A Chickenfoot script can automatically answer this question by submitting a query to the library's online catalog interface, retrieving the location of the book, and injecting it directly into the Web page for the online bookstore, in the user's browser.
Automation:	Yes.
Mashups:	Yes, Chickenfoot can combine multiple Web sites within the browser.
Scripting:	Yes. Users can write scripts in JavaScript using Chickenfoot's command library, in an ordinary text editor.
Natural language:	No.
Recordability:	Yes. Recorded actions are displayed in an output panel, where they can be copied into a script.
Inferencing:	Yes. Heuristics are used to match a script command against the page. For example, click("search") triggers a heuristic search for a clickable element on the page that is labeled with the keyword "search".
Sharing:	Chickenfoot has a wiki where users post scripts and helpful library functions.
Comparison to other systems:	Similar to CoScripter in its ability to automate Web sites, although CoScripter has better support for recording scripts and a wiki integrated into its user interface.
Platform:	Implemented as an extension to Mozilla Firefox Web browser.
Availability:	Freely available at <a href="http://groups.csail.mit.edu/uid/chickenfoot/">http://groups.csail.mit.edu/uid/chickenfoot/</a> . Source code is available under the MIT license.