Smart Bookmarks:
Automatic Retroactive Macro Recording on the Web

by
Darris Hupp

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

May 11, 2007

Author .....

Department of Electrical Engineering and Computer Science
May 11, 2007

Certified by .....

Robert C. Miller
Associate Professor
Thesis Supervisor

Accepted by .....

Arthur C. Smith
Professor of Electrical Engineering
Chairman, Department Committee on Graduate Theses

Smart Bookmarks:
Automatic Retroactive Macro Recording on the Web

by
Darris Hupp

Submitted to the
Department of Electrical Engineering and Computer Science

May 11, 2007

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

New technology has made the Web more dynamic and personalized, while at the same time interaction with the Web has become more complicated and involved. This thesis presents Smart Bookmarks, a web automation system that allows users to automate complex tasks or easily return to otherwise hard-to-reach dynamic web pages by creating smart bookmarks. A smart bookmark consists of an automatically generated script of recorded browsing commands that returns to a particular web page or web application state. Smart bookmarks can be created retroactively, meaning that the user does not need to explicitly initiate recording before performing a task, but can instead request a bookmark after visiting the destination page; the appropriate sequence of commands need to return to a page is selected automatically from a history of the user's browsing interactions. Smart Bookmarks provides a rich, visual representation of recorded bookmarks in order to clearly illustrate the actions that a bookmark performs, and includes textual descriptions, screenshots, and animated previews of each command. Finally, the system allows users to easily and intuitively edit bookmarks after they have been created, and to share smart bookmarks with other users.

Thesis Supervisor: Robert C. Miller
Title: Associate Professor

Acknowledgements

I would like to thank Rob Miller for his invaluable advice and support. He was a constant source of encouragement and inspiration throughout this process, and I am grateful to have had him as my research advisor.

I would also like to thank all the other members of the User Interface Design group for their technical advice and valuable feedback, and especially Kevin Su, Vikki Chou, and Greg Little for their insight and support.

Finally, I would especially like to thank my parents, Russ and Regina, and my sister, Chelsie. They have been a constant source of motivation and support throughout this project and my entire MIT career. None of this would have been possible without them and their unwavering encouragement and belief in my abilities.

Contents

List of Figures

List of Tables

1 Introduction

As the Web has grown and matured, new technologies such as scripting languages, cookies, and XML have been introduced that have made the Web much more dynamic and personalized than before. At the same time, however, Web interaction has become more complicated and involved [2, 5]. Users are often required to correctly remember and input user name and password combinations and fill out sequences of tedious forms to perform even relatively simple tasks. Furthermore, web pages are often dynamically generated, and so it is not always possible to use traditional bookmark facilities to retrieve them. This is unfortunate, because it is precisely these kinds of pages that require the most effort from users to repeatedly access, and which would therefore benefit the most from a bookmarking facility that allows dynamic content to be retrieved with a single click, as traditional bookmarks provide for static web pages today [6].

This thesis presents a solution to this problem in the form of *smart bookmarks*, which consist of a script of Web interaction commands that can be replayed to return to a particular web page, or to restore a particular state of a web application. An example of a short smart bookmark script that accesses my MIT Webmail inbox is shown in Figure 1.1.

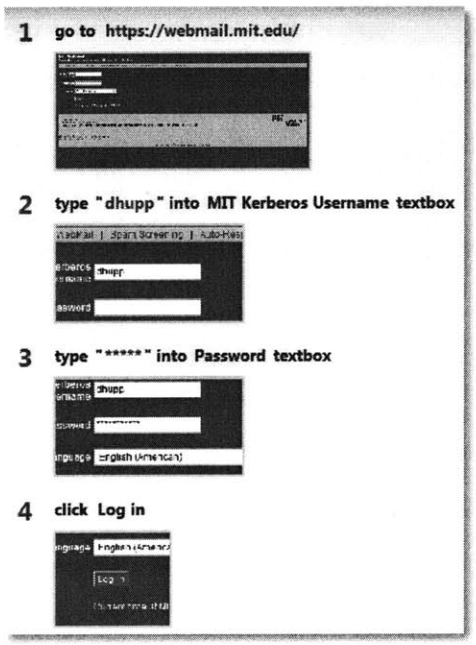


Figure 1.1 A smart bookmark script that accesses my MIT Webmail inbox

In addition to recalling hard-to-reach, dynamic web pages, smart bookmarks are also useful for automating common tasks on the Web. Users may often find themselves repeatedly performing the same tedious sequence of steps to perform a task, such as searching for a flight on a travel website. Smart bookmarks allow users to easily automate these kinds of tasks by creating a script of commands that performs the task when replayed. Because portions of web based tasks are often slightly different each time they are performed, smart bookmarks can also be parameterized, so that users need only enter the one or two parameters that change each time when replaying a bookmark (the departure and return dates in a flight search, for instance).

A third benefit that smart bookmarks provide is that they can allow users to easily share dynamically generated content. For example, a user could customize a laptop he or she wishes to purchase online, save the completed configuration page as a smart bookmark, and send the bookmark to a friend to look over. Smart bookmarks could also be used as step-by-step tutorials that could be sent to someone to illustrate how to perform a task on a website.

The Smart Bookmarks system is implemented as a sidebar inside the Firefox web browser (Figure 1.2). From the Smart Bookmarks sidebar, users can bookmark their current web page, run a bookmark they have already created, or view and edit a graphical representation of an existing bookmark. Smart Bookmarks continuously monitors and records the actions the user makes as he or she browses the Web and stores them as commands in the background. Then, when a web page is bookmarked, it automatically determines the necessary sequence of commands from the user's browsing history needed to return to that page or web application state and saves those commands as a smart bookmark.

Once a bookmark has been created, it can be replayed with a single click. Smart Bookmarks will replay the bookmark's command sequence in the user's web browser to return to the page and state that was originally bookmarked. Users can also indicate that they should be asked about the inputs for certain commands in the bookmark before or during each playback. This would be useful for automating a task where only one or two things change each time the task is performed.

In order to clearly show users what a particular bookmark will do when replayed, Smart Bookmarks displays a graphical representation of the bookmark. In addition to textual descriptions of a bookmark's commands, the graphical view includes a screenshot of each command showing the part of the web page that command acted on when the bookmark was created, as well as an animated preview for each command that more clearly shows its context within the web page as a whole. Smart Bookmarks also allows users to easily edit a bookmark once it has been created, for instance by changing the inputs to a command, copying commands, or removing a command from a bookmark entirely. Users can also dynamically edit or even create bookmarks in real time by explicitly performing the actions they want to capture in their browser. A screenshot showing how an example bookmark appears in the Smart Bookmarks sidebar is shown in Figure 1.3.

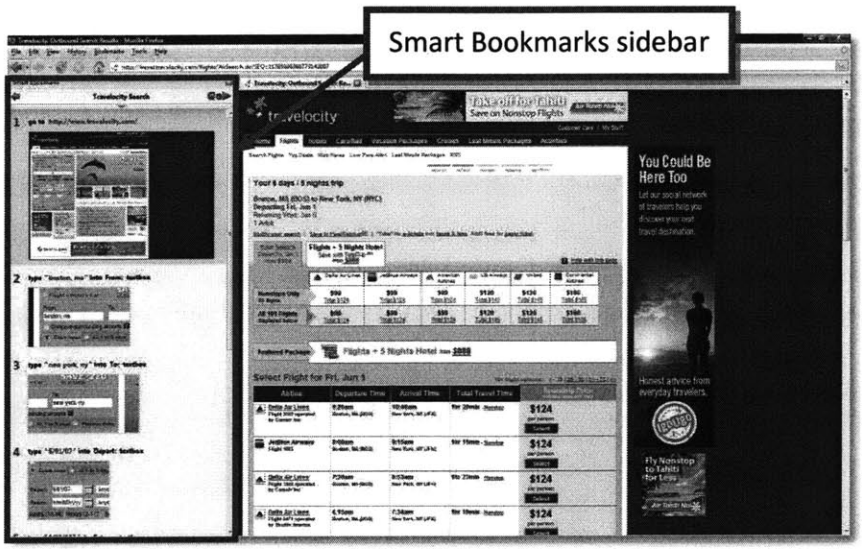


Figure 1.2 Smart Bookmarks is implemented as a sidebar in the Firefox browser

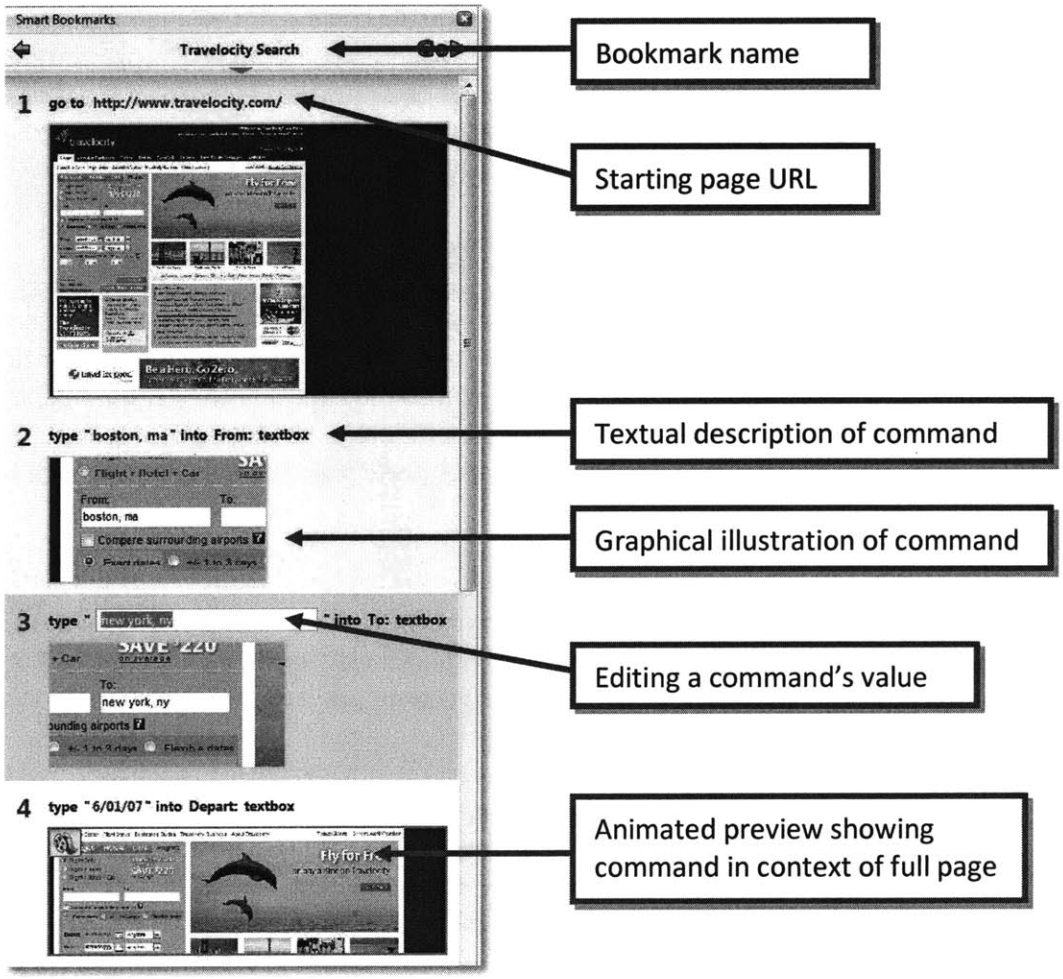


Figure 1.3 An example bookmark displayed in the Smart Bookmarks sidebar

Smart Bookmarks differs from previous Web-based macro recorders [1, 2, 5] in two key areas. First, it utilizes the notion of *automatic, retroactive macro recording*. That is, unlike other macro recorders that use a VCR-like interface that requires users to explicitly specify when to start and stop recording, Smart Bookmarks is able to continuously record users' actions silently in the background as they browse the Web. Users do not need to decide in advance that the task they are about to perform should be recorded. Furthermore, once the task has been performed or the desired web page reached, Smart Bookmarks is able to determine the correct sequence of recorded commands necessary to return to the web page or application state automatically. Users only need to indicate retroactively that they would like to create a bookmark, and Smart Bookmarks will scan back through their command history to find an appropriate starting point for the macro. Automatic, retroactive macro recording conveniently allows users to create macros without any advanced planning and without the need to repeat a task for the sole purpose of recording a macro after determining that one would be useful. It also neatly fits into the "one-click" model of traditional URL-based bookmarks that many Web users are already familiar with.

A second key contribution that Smart Bookmarks provides is its rich, visual representation of recorded bookmarks. A smart bookmark contains many features that help clearly illustrate what the bookmark does and how it does it, including textual descriptions, screenshots, and animated previews of each command. This feedback serves to make smart bookmarks more understandable, and to inspire more confidence in the automation itself, since users can see at a glance immediately after a bookmark is created what it will do when replayed. An additional benefit of this rich visual feedback is that it allows users to easily share bookmarks with each other with less confusion. In fact, two of the primary intended uses for Smart Bookmarks are to allow users to share dynamic or customized content with each other, and to support the creation of step-by-step tutorials, both of which clearly benefit greatly from bookmarks that are easy to understand and follow.

The rest of this thesis discusses the details of the Smart Bookmarks design and implementation. Related work is presented in Chapter 2. Chapter 3 illustrates a few of the main uses of Smart Bookmarks through the use of three example scenarios. The system's user interface is discussed in Chapter 4. Chapter 5 explains the details of Smart Bookmark's implementation. Chapter 6 provides a discussion of some of the challenges inherent in the design of this kind of system, and how Smart Bookmarks attempts to overcome them. An evaluation of Smart Bookmarks and some of its integral components is presented in Chapter 7. Finally, Chapter 8 concludes and discusses possible future improvements and extensions to Smart Bookmarks.

2 Related Work

Smart Bookmarks draws from and builds upon previous related work in several different areas:

- Web automation and customization with scripting languages and syntax-free programming
- Web-based macro recording
- Non-macro-recording Web-based programming by demonstration systems
- Visual macro recording

This section discusses some of the aspects of prior work in each of these areas that Smart Bookmarks incorporates and why, as well as the ways in which Smart Bookmarks differs from, improves upon, or extends some of these ideas.

2.1 Web Automation and Customization

Smart bookmarks are essentially scripts of commands that can be replayed to automate tasks on the Web. Smart bookmark commands are based on the ideas introduced by Chickenfoot [3], a programming system integrated in the Firefox browser that allows users to automate and customize websites using scripts. Chickenfoot is designed so that users can create scripts that manipulate web pages by referring only to the user interface itself; that is, the rendered web page as displayed by a browser. For example, a Chickenfoot script might consist of human-readable commands like `click("search button")`, rather than commands that involve the underlying HTML source of a page.

Using commands written at the user-interface abstraction level provides some important benefits. First, because it is not necessary to manipulate or even look at the HTML source of a web page, scripts can be written more easily and more quickly. Additionally, once written, these scripts are more understandable to both programmers and non-programmers alike. Furthermore, scripts that identify page components using descriptive keywords based on the contents of a site's user interface are more resilient to changes in the web page's structure, since site designers are likely reluctant to modify the outward presentation of their web pages in order to avoid confusing their visitors.

Chickenfoot later took this idea even further by introducing the notion of keyword commands [19], as a syntax-free alternative to traditional scripting languages, including Chickenfoot's own JavaScript based language. For example, instead of writing the command `click("search`

`button")` as above, syntax-free programming frees users to write more natural statements like `click the search button.`

While Chickenfoot is not itself a macro recorder (though it can display commands corresponding to user actions while browsing the Web), its notion of having commands that refer to web page components using only descriptive elements extracted from the visible user interface of a page is clearly beneficial to macro recording as well. Smart Bookmarks internally represents bookmark commands in a manner akin to Chickenfoot commands. Specifically, it uses visible keywords from web pages to refer to components involved in recorded commands, rather than the more direct, but less intuitive and less resilient approach of utilizing aspects of the internal page structures, such as Document Object Model (DOM) XPaths [11, 24] or component IDs. Additionally, Smart Bookmarks displays the textual representation of bookmark commands in the form of syntax-free keyword commands. This makes it easier to see at a glance what a smart bookmark does.

2.2 Web-based Macro Recording

Macro recording is clearly an important component of the Smart Bookmarks system. More specifically, Smart Bookmarks is concerned with macro recording as it pertains to the Web; that is, Smart Bookmarks and previous systems [1, 2, 6, 20] are designed to be able to capture user interaction with web pages inside a browser, and to store those recorded interactions in a form that can be replayed in the future to recreate the user's original actions. The method used to capture and replay those actions, and the form they take once recorded, differs from system to system.

2.2.1 Proactive vs. Retroactive Recording

Much of the previous Web-based macro recording work relies on proactive recording [1, 2, 6], where users must explicitly tell the system when to start and stop recording their browsing actions. Smart Bookmarks, on the other hand, supports recording retroactively: user interaction is continuously recorded in the background, and a macro can be created using a subset of past commands at any time. This eliminates the need for users to know in advance that the sequence of steps they are about to perform could be useful to save as a macro that they can reuse in the future, as might often be the case while browsing the Web. Smart Bookmarks takes this idea a step further by not requiring users to explicitly specify which commands from their recorded history should be included in the macro they want to create. Instead, users need only indicate that they would like to create a smart bookmark that will return them to the web page or state that they are currently viewing, and Smart Bookmarks is able to determine automatically the sequence of commands from their history necessary to include in the bookmark. That said, Smart Bookmarks supports both proactive macro recording and explicit retroactive recording as well, if those methods are preferred by the user for a particular situation. WebMacros [6], an earlier Web-based macro recording system, also proposed a "Get Me Here" feature similar to the retroactive macro recording employed by Smart Bookmarks, but didn't describe how to implement it.

2.2.2 Architectures for Recording User Actions

All macro recorders designed for the Web need a method for detecting events initiated by users in their browser so that those events can be recorded. For instance, the user might click on a link, type a URL into their address bar, or enter information into and submit a form. There are at least two primary methods for detecting and capturing this information: using a proxy that sits between the browser and the Web, and imbedding the system within the browser itself.

Previous Web macro recording systems like LiveAgent [1] and WebMacros [6] use a proxy architecture that sits between the user's browser and the Web, intercepting any requests initiated by the browser and any responses returned from the server. The system can then make any changes necessary to the HTML source to support macro recording and playback before passing it along to the browser.

LiveAgent was one of the earliest Web automation systems, and allows users to automate repetitive tasks through proactive demonstration, by manually starting and stopping recording of their browsing sessions. LiveAgent implements a proxy between the user and the Web that modifies web pages on the fly to incorporate JavaScript event handlers on buttons, links, and other components, which then allow the system to recognize and record user actions.

A later system, WebMacros [6], also users a proxy-based architecture and works by rewriting HTML pages to include annotations that support macro recording and playback. WebMacros replaces all URLs in links and form actions with a special URL containing additional information used by the recording system that it can intercept and interpret when the user clicks a link or submits a form.

One of the main advantages of using a proxy-based approach is that proxies are ideally independent of both the browser and the operating system, which is clearly not possible with an embedded browser solution. Integrating with a cross-platform browser helps mitigate this issue somewhat, however. However, proxies have some significant disadvantages as well. Proxy systems cannot modify web pages that are accessed over a secure connection, since those pages are encrypted by the browser before the proxy is given access to them. This is important since a growing number of sites on the Web today do employ encryption, and macro recorders need to have access to the unencrypted HTML source of a page in order to track users' browsing events. Perhaps an even bigger issue is that many web pages use significant amounts of client-side JavaScript in the design of their user interfaces. Since a proxy has no access to the web page once the HTML response has been passed along to the browser from the server, it is unable to respond to any events caused by user interaction with the client-side JavaScript.

A second approach that avoids some of the problems inherent to proxy-based architectures is to embed the macro recording system inside the web browser itself. This is the approach that Smart Bookmarks takes, as well as other web automation systems like Chickenfoot [3], Koala [20], and WebVCR [2]. Because these systems have access to the browser and its content directly, they are able to see and modify web pages exactly as the user sees them, including

changes that occur due to the use of cookies, client-side JavaScript, and style sheets. Furthermore, embedded systems have no problem accessing or modifying secure web pages, since they will already be decrypted by the browser. Additionally, the issue of being unable to detect and respond to client-side scripting activity is avoided as well.

More specifically, Smart Bookmarks is embedded within the Firefox web browser as a sidebar extension (like Chickenfoot and Koala as well). Smart Bookmarks is able to detect events caused by user interaction with web pages through the use of JavaScript event handlers attached to the Firefox browser window itself. This allows it to seamlessly create smart bookmarks using commands that span multiple tabs and even multiple windows, something some of the previous systems were unable to do.

WebVCR [2], another Web-based macro recorder, takes a slightly different approach to the in-browser technique. WebVCR is implemented as a Java applet for use with the Netscape web browser, and uses a VCR-like interface that requires users to explicitly start and stop macro recording. When it is recording, WebVCR automatically inserts event handlers into web pages as they load so that it can record users' browsing activity. WebVCR also coined the term *smart bookmark* to refer to macros that provide shortcuts to web content, though it is unable to generate such macros automatically. As using a Java applet is an in-browser approach, WebVCR is also dependent on the browser itself, the operating system, and more specifically in this case the Java Virtual Machine. Additionally, for security reasons this approach requires users to explicitly confirm the necessary security privileges that the Java applet requires so that it can access and modify web pages outside of its own domain. The applet must also remain open in a separate browsing window while recording so that it can retain state across multiple web pages.

2.2.3 Representing Recorded Commands

While the number of methods for detecting events generated through user interaction with web pages is relatively small, the manner in which those actions are formatted and stored varies widely among all the macro recorders discussed so far.

LiveAgent uses a proprietary format called the HTML Position Definition Language (HPD), which provides a significant amount of flexibility in referring to elements the user has interacted with on the page [1]. For instance, a single link could be represented in terms of its order in relation to other elements on the page, the words contained in the link's text, or a variety of other possibilities. However, while this approach is both flexible and powerful, LiveAgent is not able to infer this information automatically; users must supply the HPD for each element in the macro explicitly using a dialog box.

WebVCR only explicitly stores link traversals and form submissions, and includes information associated with other elements, such as textboxes or radio buttons, with the form they belong to when that form is submitted [2]. For links, it includes information like the URL the link points to, the text displayed by the URL, and its location in the web page DOM. Similar information is stored for form submissions, in addition to the names, values, and locations of any other

elements associated with the form. As some of the information WebVCR uses to identify elements is likely to change frequently as web pages are updated, the system includes heuristics to match steps in the recorded macro with actual page elements during playback. For instance, if a step refers to a link that no longer exists in the current web page DOM, WebVCR will attempt to find a link elsewhere in the document with the same link and displayed text, or barring that any link with the same text but different URL, and so on. These heuristics help protect WebVCR scripts against changes to web page structure.

WebMacros stores recorded macros internally in a relational database [6]. The actual macro steps are stored in one table, which, like WebVCR, consists only of link traversals and form submissions. It includes information like the requested URL, DOM location of the element, and whether the step was a link traversal, form submission, or external page load. A second table stores information related to the elements associated with form submissions. Elements are identified by type (textbox, list box, etc), and by name extracted from the HTML source. Finally, a third table contains the actual values associated with the various form elements. While WebMacros does not make any attempt to protect against changes to the underlying page structure during macro playback, it does attempt to detect when playback is proceeding incorrectly by checking for HTTP error codes and comparing the web pages that are retrieved during playback to corresponding page fingerprints that were calculated and stored with the macro when it was created.

While LiveAgent, WebVCR, and WebMacros all represent recorded commands and their associated web page elements using primarily attributes extracted from the internal HTML source code of the page, Koala (and Smart Bookmarks) takes the approach introduced by Chickenfoot, using keywords derived from the visible user interface of a web page instead. Koala uses proactive macro recording, where users start and stop the recording as necessary, and actions are recorded in the form of natural keyword commands [20]. This is important, since it allows recorded scripts to be more robust against changes to page structure than scripts that rely on the details of the internal HTML, as the user interface of a page is less likely to change frequently over time. Once recorded, Koala command scripts are saved automatically in a wiki, allowing users to easily exchange and improve scripts. This is possible because recorded Koala commands are represented using natural language that is easy for people to read and understand, and are visible in the user interface itself. Smart Bookmarks also stores recorded commands internally using keywords that reference the visible interface of web pages, and displays commands externally to the user using syntax-free keyword commands. Smart Bookmarks takes this one step further, however, and utilizes a rich graphical representation for bookmark commands that should make it even easier for users to understand, use, and modify recorded bookmarks.

2.2.4 Variable and Private Commands

Some previous Web-based macro recording systems, such as WebMacros, support command parameterization, meaning that users can mark particular page elements as being either variable or constant [6]. Variable parameters can then be provided by the user each time the

macro is played, rather than always using the same value. Parameters can also be marked as private, which indicates that the parameter value should not be stored with the macro.

Users must explicitly indicate to WebMacros whether an element should be constant, variable, or private, by selecting from a set of radio buttons associated with the element and inserted into the HTML source of the page by the proxy. Reasonable defaults are provided, however. WebVCR supports the use of variable parameters as well, in addition to private parameters which the system encrypts and stored with the macro [2].

Smart Bookmarks also supports parameterization; however, users specify that a parameter is variable through the Smart Bookmarks sidebar user interface, rather than by modifying the web page itself. Additionally, Smart Bookmarks allows users to choose whether they would prefer to enter the values for variable parameters all together before replaying a bookmark, or through the web page itself during bookmark playback. Smart Bookmarks supports private parameters by automatically detecting password fields and storing those values in Firefox's secure built-in password manager rather than storing them with the smart bookmark itself.

2.3 Other Web-based Programming-by-Demonstration Systems

While not directly related to Smart Bookmarks and other macro recorders, other research has looked into applying programming-by-demonstration on the Web to areas other than macro recording.

One example of such a system is Internet Scrapbook [8], which allows users to create a personal web page using portions of multiple other pages that they can specify graphically using a process called clipping. The personal web page can then be automatically updated by extracting the relevant information from the clipped pages. In order to specify which part of a given web page should be included as a clipping in a manner that is likely to be robust against changes to page structure, Internet Scrapbook uses a combination of patterns derived from the visible page (like Chickenfoot, Koala, and Smart Bookmarks), and patterns involving the structure of the HTML source.

More recent work [16] has taken this idea further by providing users with the ability to interactively specify extraction patterns that the system can use to automatically find similar content, and then present a collection of such content in the form of visual summaries. Other work [17] involves using clipped regions of some web pages as the inputs to clipped regions of other pages to perform computations that no single web page provides on its own. Finally, Sifter [18] is a web browser extension that is able to automatically scrape structured data from web pages and present that data within the context of the original web page in a manner that can be easily filtered or sorted.

2.4 Visual Macro Recording

One key difference between Smart Bookmarks and the other Web-based macro recorders discussed is that they either offer no user visible representation of recorded actions at all

(WebVCR, WebMacros), or display scripts only in a textual format (Koala). Every command recorded by Smart Bookmarks, on the other hand, is represented visually in the user interface. Smart Bookmarks uses a combination of textual descriptions, web page screenshots, and animations to help convey the purpose of each individual command in a bookmark, as well as the bookmark as a whole.

The film strip metaphor utilized by Smart Bookmarks was pioneered by two earlier non-Web based macro recording systems, Chimera [9] and Pursuit [21]. Chimera introduced the notion of graphical histories that visually represent actions performed within a drawing editor, while Pursuit is able to graphically represent commands performed within a desktop environment similar to the Apple Macintosh Finder. Additionally, Chimera supports the creation of macros retroactively, by allowing users to select and copy commands from its graphical history.

3 Usage Scenarios

Smart Bookmarks can be used to automate a variety of common tasks on the Web; these tasks generally fall into one of three areas: accessing dynamic or hard-to-reach web pages, automating repetitive tasks, and saving or sharing customized dynamically-generated content. This section presents a typical scenario from each of these primary usage areas to illustrate how Smart Bookmarks can be used to supplement normal web browsing.

3.1 Accessing a Hard-to-Reach Bank Portfolio Page

Many banks and other financial institutions include an online portfolio feature that allows users to view a variety of information about their various accounts and investments. While this information is likely very useful, it is often also difficult and tedious to access because, like many modern Web pages, the content is dynamically generated and not directly accessible with a standard URL.

As an example, to access my Bank of America online portfolio, I must first browse to the bank's website, select my user ID, and click a sign-in button. On the following page, I need to enter a pass code and click a second sign-in button. Next, I am presented with an overview of my accounts. From there, in order to view my actual online portfolio, I must click on a "My Portfolio" link, which finally takes me to an overview of the portfolio. If I then want to see a different page within my portfolio, a summary of net worth for instance, I would need to click yet another link. So, in order to access one web page, I'm required to perform at least seven actions within my browser. It isn't possible to bookmark any of these intermediary pages using traditional URL-based bookmarks, since they are all dynamically generated based on the login information supplied at the start of the process.

This process can be greatly simplified using Smart Bookmarks. Using a Firefox browser equipped with Smart Bookmarks, I could simply press the *Create Bookmark* button in the toolbar of the Smart Bookmarks sidebar once I've reached my portfolio page (Figure 3.1), and a smart bookmark containing all of the steps necessary to return to that page will be created automatically. A graphical representation of the bookmark will appear in the sidebar (Figure 3.2), which I can quickly glance over to ensure it does what I expect it to do. Any password fields (such as my pass code) are automatically detected and shown using asterisks in the user interface. Additionally, passwords are securely stored using Firefox's built-in password manager and are not saved in a regular smart bookmark file.

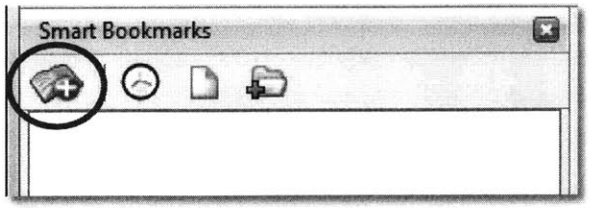


Figure 3.1 Use the Create Bookmark button the Smart Bookmarks sidebar to create a new smart bookmark that returns to the current web page

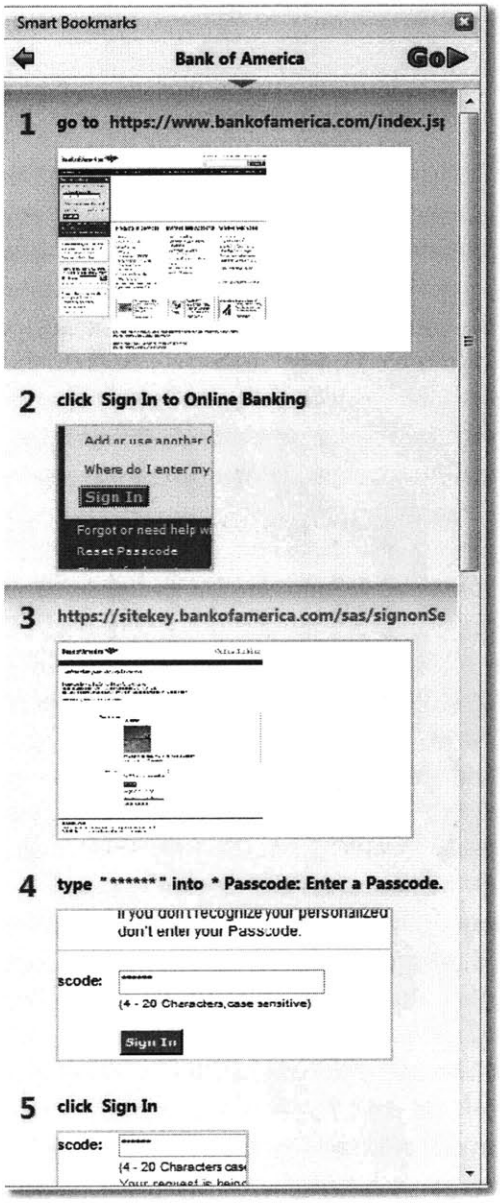


Figure 3.2 A portion of the Bank of America bookmark shown graphically in the sidebar

In textual form, the complete smart bookmark created for this task is shown below:

```
go to https://www.bankofamerica.com
click Sign In to Online Banking
type "******" into Passcode: Enter a Passcode textbox
click Sign In
click My Portfolio
click Net Worth Summary
```

Now, when I want to access my portfolio, I need only select the bookmark in the Smart Bookmarks sidebar and click the *Go* button in the toolbar (Figure 3.3). The system will play the commands in the bookmark in sequence, returning to the portfolio page I originally bookmarked.

Figure 3.3 Replay a smart bookmark by clicking the Go button

3.2 Searching for a Flight on a Travel Website

Travel websites often require users to fill out tedious forms to search for flight fares and other information. Users that travel frequently may often find themselves repeatedly filling in largely the same information each time they look for a flight.

As one example, a user may live in Boston, but have a friend who lives in New York that she visits frequently. She often flies down to New York and back on the weekends, leaving Friday night and returning Sunday afternoon. Each time she searches for a flight, she goes to Orbitz.com, enters *Boston, MA* as the departure city, *New York, NY* as the arrival city, and fills in the dates she wants to leave and return. Also, because she wants to leave Boston in the evening and return sometime in the afternoon, she chooses those options from the appropriate drop down menus as well. Finally, she clicks the *Find Flights* button and after a few moments is presented with a list of possible flights to choose from.

Because she performs this flight search often, and because much of the process is exactly the same every time, she may find it useful to automate the task using a smart bookmark. To do this, she clicks the *Create Bookmark* button in the Smart Bookmarks sidebar, and is presented with the following bookmark in textual form:

```
go to http://www.orbitz.com/
type "New York, NY" into From City name or airport textbox
type "Boston, MA" into To City name or airport textbox
type "08/10/07" into Leave textbox
choose 6p-12a
type "08/12/07" into Return textbox
choose noon-5p
```

```
click Find Flights
```

While most of this process will always remain the same, the departure and return dates will be different each time the user needs to search for this flight. Smart Bookmarks supports this kind of parameterization by allowing users to specify that the values for particular commands are variable, and so should be specified each time a bookmark is played. The system allows users to enter values for variable parameters either all at once right before playback begins, or one by one during playback in the web page itself. In this case of the example above, the user would like to enter the return and departure dates together at the start, so she right clicks on the command *type "08/10/07" into Leave textbox* in the graphical command list for the bookmark, and selects *Choose value -> Before running this bookmark* from the popup menu that appears (Figure 3.4). She then does the same for the command that specifies the flight's return date.

After doing this, when the user plays the bookmark, she will be asked to specify the departure and return dates in the Smart Bookmarks sidebar before the bookmark will begin to play.

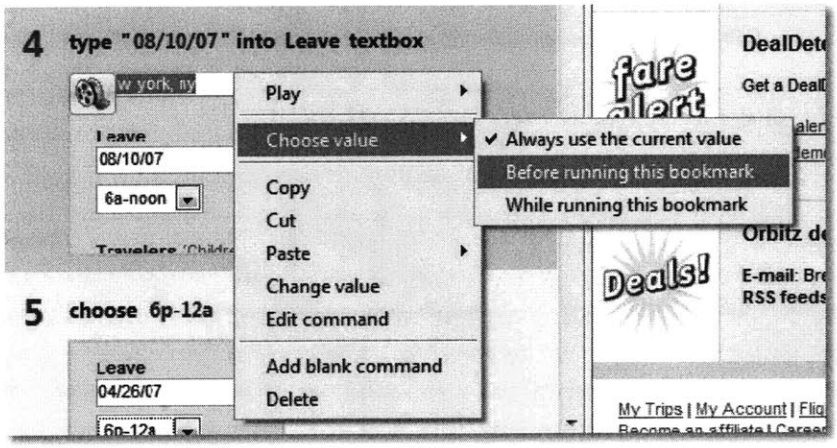


Figure 3.4 Smart Bookmarks allows users to specify that a command's value is variable and should be specified each time the bookmark is played

3.3 Sharing a Custom Laptop Configuration

There are many online computer manufacturers that allow users to customize default systems with a wide variety of different parameters before purchase. As an example, one user needs to choose and customize a computer system, and then send the information to someone else who will actually be purchasing the system. This may be the case because his company or school is paying for the computer, or because he was asked by a less computer-savvy friend to create a system for them. In any case, once the user has chosen a system and modified it with a few dozen options, there is likely no easy way for him to send the resulting custom configuration to someone else. Instead, he may have to provide written instructions to the buyer, describing how the computer should be customized. At some point in this process, mistakes or misunderstandings are likely to occur, and an incorrect system may be purchased.

Smart Bookmarks can be used to get around this problem by allowing the user to save the configuration using a smart bookmark containing the commands that complete the desired customizations. So, after visiting Dell's website, selecting a base model, and making a number of customizations, the user decides he is happy with the laptop configuration he has created and presses the *Create Bookmark* button in the Smart Bookmarks sidebar. A bookmark is generated that is able to recreate the customized laptop he just finished creating:

```
go to http://configure.us.dell.com/dellstore/config.aspx?c=us …
check Intel® Core™ 2 Duo T5600 (1.83GHz, 2MB L2 Cache …
click Go to Next Component
check Genuine Windows Vista™ Business [add $70 or $2/month 1 ]
click Go to Next Component
check 15.4 inch UltraSharp™ Wide Screen WXGA+ Display with …
click Add My Accessories
check Dell Photo AIO 926 - Includes Media Card Reader …
click Choose My Software
check Microsoft® Office Small Business 2007-includes …
click Confirm & Add to Cart
```

The user can save this smart bookmark to a file (Figure 3.5) and email it to whoever is going to complete the purchase. The purchaser can then import the bookmark into his web browser (Figure 3.6), quickly scan its visual representation to see what it does, and play it in order to recreate the same configuration and complete the ordering process.

Alternately, if the purchaser does not also have Smart Bookmarks installed, the user can copy the bookmark to the clipboard and paste it into his email client, where it will appear as a list of textual keyword commands. The purchaser can then follow these instructions by hand to recreate the same configuration.

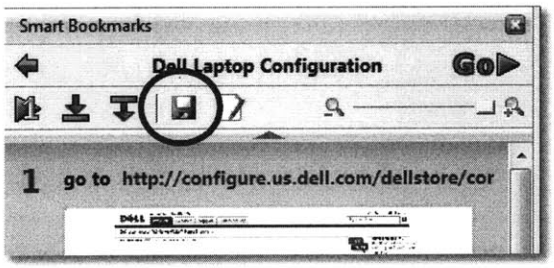


Figure 3.5 Saving a smart bookmark to a file

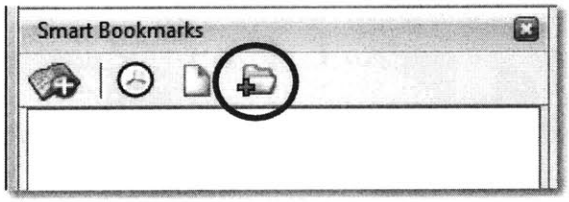


Figure 3.6 Importing a smart bookmark

4 User Interface Design

One of the overarching design goals for Smart Bookmarks was for it to be as close as possible to the metaphor of traditional bookmarks that users are already familiar with, and as easy to use, but be more flexible and more powerful with regard to the tasks that it is able to accomplish. More specifically:

- It should be easy to create a smart bookmark for a particular task, and to replay that bookmark later.
- Users should be able to quickly review a bookmark to visually see what it does.
- The system should support multiple mechanisms for creating and replaying bookmarks if needed or preferred by the user.
- It should be simple and intuitive to edit a bookmark once it has been created.
- In general, the system should be as unobtrusive and integrated into the user's normal browsing experience as possible.

4.1 Overview of the Layout

Smart Bookmarks is implemented as a sidebar for the Mozilla Firefox web browser (Figure 4.1). Because of this, its layout should be instantly familiar to users who have used Firefox's built-in Bookmarks or History features, which are also implemented as sidebars. This also means that Smart Bookmarks can remain onscreen as users browse the Web, or be hidden in the background until the user decides to open it with a simple key stroke.

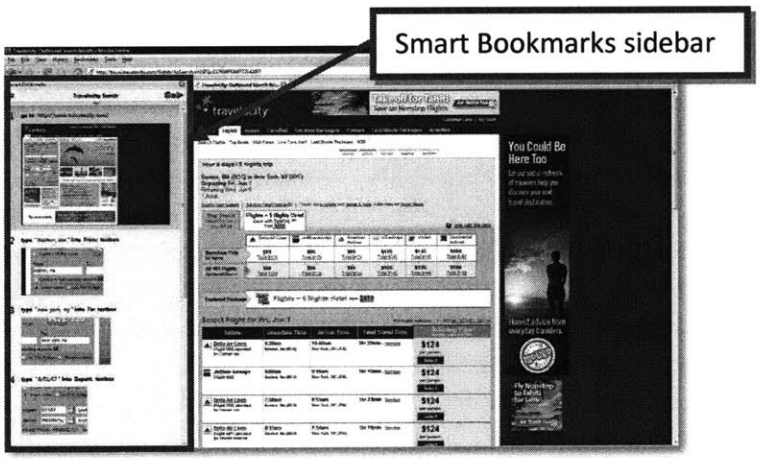
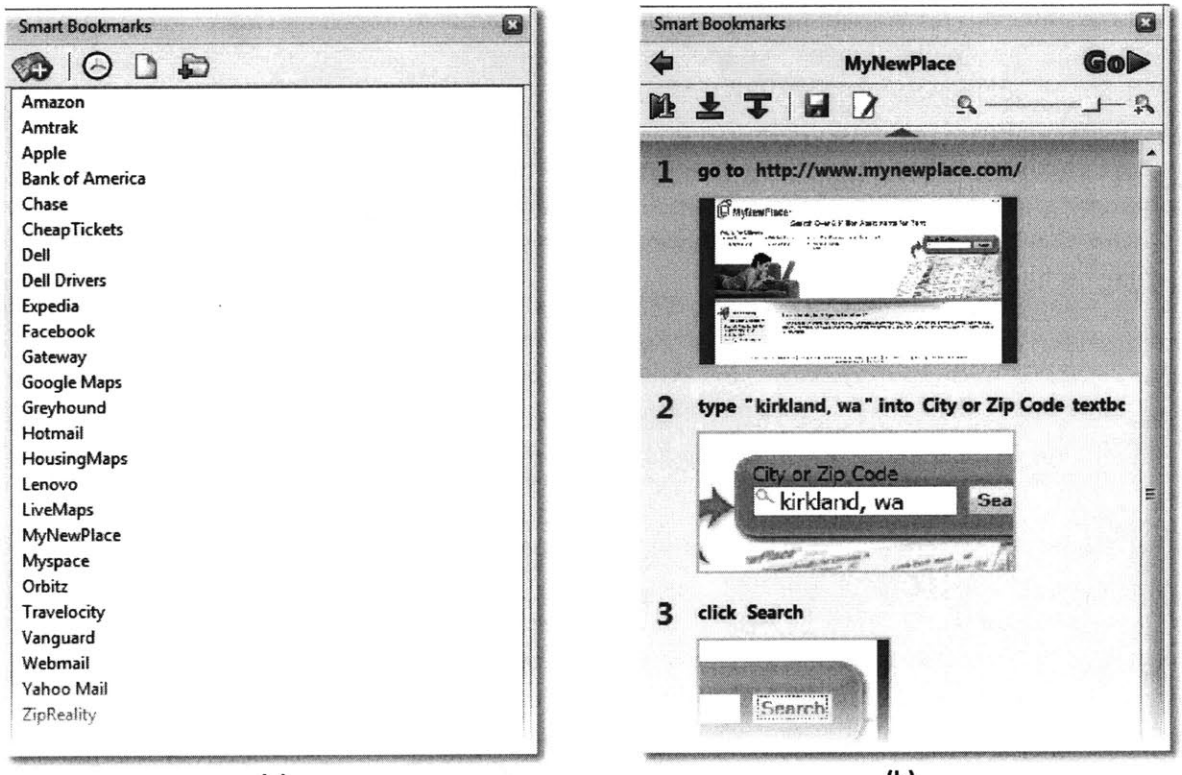


Figure 4.1 Smart Bookmarks is implemented as a sidebar in the Firefox browser

The Smart Bookmarks interface consists of two main components: a *list view* and a *details view*. The list view appears when the user first opens the Smart Bookmarks sidebar, and contains a list of all the user's smart bookmarks by name in alphabetical order (Figure 4.2a). Using the buttons found in the toolbar running across the top of the sidebar, users can create a new bookmark that will return to the page currently visible in their browser, create a new blank bookmark, import an existing bookmark into their list, or view a graphical history of their browsing actions. Users can play any bookmark in their list directly from this view as well. By clicking on the name of one of their bookmarks, users will be taken to a details view, which displays a visual representation of the smart bookmark, as well as a variety of commands for playing or editing the bookmark (Figure 4.2b).

The design follows that of the traditional bookmarking functionality where possible. Like URL-based bookmarks, smart bookmarks can be created with a single click by simply browsing to the page that should be bookmarked. Smart bookmarks are also maintained in an easily accessible list that the user can keep visible as they browse, and a bookmarked page can be revisited with one click as well. However, Smart Bookmarks also prominently integrates graphical representations of the bookmarks created by the system. These visual cues are important because smart bookmarks are, of course, significantly more complex than traditional bookmarks, and so it is necessary to provide as much assistance to users as possible with recalling and understanding the tasks that their bookmarks perform.



(a) (b)

Figure 4.2 The Smart Bookmarks sidebar showing the list of bookmarks this user has created (a), and a graphical view of one of those bookmarks (b)

4.2 Creating a Bookmark

A smart bookmark can be created by clicking the *Create Bookmark* button in the list view toolbar (Figure 4.3). This tells Smart Bookmarks that the user wants a bookmark that can return to the web page or web application state that is currently visible in his browser. The system then creates this bookmark by generating a sequence of commands corresponding to the actions the user performed in his browser to get to the page being bookmarked.

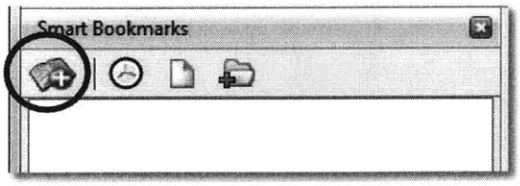


Figure 4.3 Creating a new smart bookmark

A smart bookmark will always begin with a *go to* command instructing the browser to go to a particular URL. This ensures that the bookmark can always be replayed, independent of the current state of the user's browser. If the web page the user wants to create a smart bookmark for is *bookmarkable*, meaning it is always possible to return to the page and its current state by simply entering its URL into the browser's address bar, then this initial *go to* command will be the only command in the smart bookmark and will simply load the bookmarked web page's URL directly. However, if the page being bookmarked is not bookmarkable, Smart Bookmarks will find the most recent web page from the user's browsing history that *is* bookmarkable and construct a bookmark beginning with that page's URL and followed by a sequence of commands that will recreate the bookmarked page. After the bookmark has been created, Smart Bookmarks will automatically open the newly bookmark in details view and display it bookmark graphically. The system will also supply a default name for the bookmark (the title of the web page that was bookmarked), though this name can be freely edited by the user by clicking on it in the sidebar (Figure 4.4).

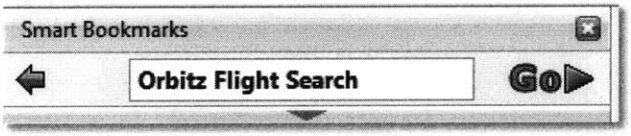


Figure 4.4 Users can change a bookmark's name by clicking on it

Every command included in a smart bookmark is represented by both a textual description of the action performed by the command, and a visual screenshot. For commands that affect a web page component (like clicking on a link or typing something into a textbox), the screenshot captures the portion of the web page that was affected, as well as a small portion of surrounding page in order to maintain context. For *go to* commands, a screenshot of the entire web page is used. An *animated preview* is also associated with each command, which smoothly zooms in to and highlights the component affected by the command from an overview of the entire page.

The command types used by Smart Bookmarks are the same as those used by Chickenfoot [3]:

```
go to URL
click button or hyperlink
type text into textbox
check checkbox or radio button
choose list-item
```

For example, a command that enters *Boston, MA* into the departure city field on the Amtrak website might be:

```
type Boston, MA into Departs textbox
```

The corresponding screenshot associated with this command might look like (Figure 4.5):

Figure 4.5 A graphical representation for the command: *type Boston, MA into Departs textbox*

Smart Bookmarks also allows users to create blank bookmarks with no commands, using the *New Bookmark* button in the list view toolbar (Figure 4.6). Users can use this option to manually create a smart bookmark by explicitly recording their actions as they browse the Web, as discussed in Section 4.5.

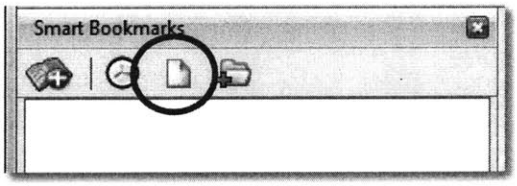


Figure 4.6 Use the *New Bookmark* button to create a new blank smart bookmark

4.3 Graphical Representation of a Bookmark

Smart bookmarks are represented visually in the sidebar using a filmstrip-like metaphor that places each command that makes up the bookmark vertically on top of each other, with the first command in the sequence at the top. Users can scroll through the bookmark to see all the commands if there are too many to fit inside the sidebar at one time.

Each command is labeled with a number, indicating its position in the entire command sequence, and a textual description of the command is shown to the right of this number. A screenshot illustrating what the command does is shown below the text description. An example command is shown in Figure 4.7.

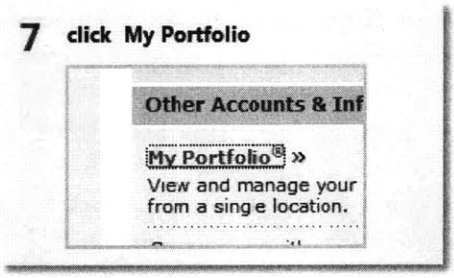


Figure 4.7 How a typical command is displayed in Smart Bookmarks

Users can also view an animated preview of a command by clicking on the *Movie Button* that appears in the upper left hand corner of its screenshot when the mouse is over top of the command. If the smart bookmark begins with an initial *go to* command (as all automatically generated smart bookmarks do), then the animated preview for that command will provide an animated run-through of the entire bookmark, illustrating what the bookmark should do if actually played. This feature could be helpful for users that have forgotten what one of their bookmarks does, or if the bookmark was sent to them from someone else. For other commands, the animated preview will smoothly zoom from an overview of the web page the command acts on, to a close up the component (textbox, link, button, etc.) affected by the command (Figure 4.8). This is useful for showing the context of the command within an entire web page.

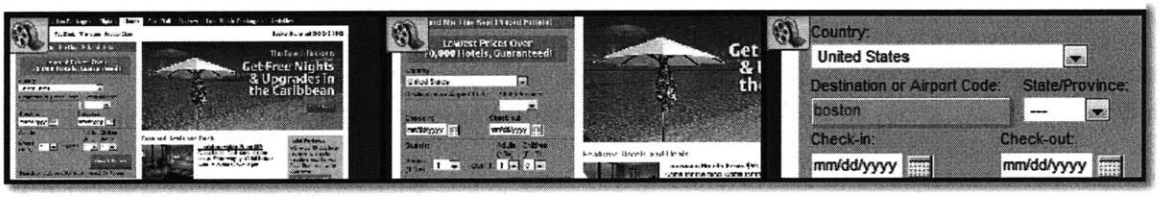


Figure 4.8 A sequence showing a few frames from a command's animated preview

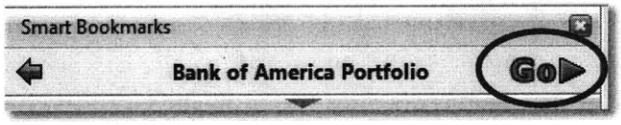
The sequence of commands in a smart bookmark will often span multiple different web pages. Smart Bookmarks indicates that a page change has occurred by inserting a place holder command that is labeled with the URL of the new page, as well as a screenshot of the page. In order to differentiate the page change from an actual command, its label is shown grayed out, and it is marked with a gradient indicating that all commands from that point forward are associated with the new page.

Smart Bookmarks also includes a slider, found in the secondary toolbar of an open bookmark, which allows users to shrink the display of commands from their full, default size to a more compact, text-only format (Figure 4.9).

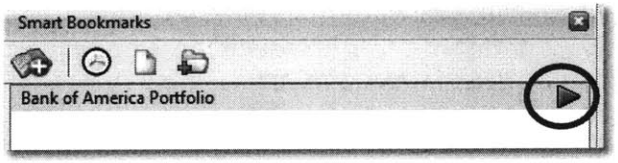
Figure 4.9 Bookmarks can be displayed more compactly using a slider in the toolbar

4.4 Replaying a Bookmark

There are several ways a smart bookmark can be replayed. When a bookmark is open in the sidebar, users can click on the *Go* button in the toolbar beside the bookmark's name (Figure 4.10a) to begin playback. Alternately, a bookmark can be replayed directly from the bookmark list by clicking on the small *play* button next to its name, or by right-clicking on the bookmark and choosing *Play* (Figure 4.10b). When a bookmark is open, the currently selected command is highlighted in green (and can be changed by clicking on another command). When a bookmark is playing, the selected command will automatically change to match the command that is currently playing in the user's browser, allowing users to watch the progress of running bookmarks in the sidebar. Also, when a bookmark is playing, the *Go* button changes to a *Stop* button, which can be used to stop playback if desired.



(a)



(b)

Figure 4.10 Replay a smart bookmark using the Go button (a), or directly from the bookmark list (b)

Smart Bookmarks also provides three additional playback options to support partial playback for debugging or other purposes; these options are available as buttons in the secondary toolbar when a bookmark is open in the sidebar (Figure 4.11). The first button plays only the currently selected command, the second plays down to the selected command from the

beginning of the bookmark, and the last button plays from the selected command to the end of the bookmark. These options are also available through the menu that appears after right-clicking on a command.

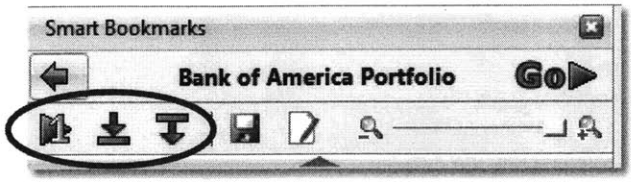


Figure 4.11 Smart Bookmarks also supports several partial playback options for bookmarks

While Smart Bookmarks is designed to be as resilient to changes in web page structure as possible, bookmarks can occasionally break when a web page changes, and fail to play back correctly. Just as traditional bookmarks can break if the web page URLs they refer to change, problems during smart bookmark playback are sometimes unavoidable. Because of this, Smart Bookmarks tries to make it as easy and painless as possible for users to update their bookmarks when errors do occur. If, during normal playback of a smart bookmark, the system is unable to successfully play a particular command, perhaps because the web page component it uses no longer exists, playback of the entire bookmark will stop and the broken command will be highlighted in red in the sidebar. The user can then use any of the editing methods discussed in Section 4.5 to correct the broken bookmark, or continue on by interacting with the web page manually.

One of the main benefits of using Smart Bookmarks is that it allows users to save time by automating common tasks or saving hard-to-reach resources that they may want to access at some point in the future. Because Smart Bookmarks is designed to be as unobtrusive as possible, and may not even always be visible as users browse the Web, there may occasionally be times when the user visits a web site for which he or she has a helpful smart bookmark saved, but may not think to use it. For cases like these, Smart Bookmarks is equipped with a reminder feature. Whenever the user visits the starting page of one of their smart bookmarks by any means (by entering its URL directly into the browser's address bar, or linking to it from an email, for instance) Smart Bookmarks will display a drop-down message at the top of the user's browser window, from which the user can play the appropriate bookmark directly (Figure 4.12). If there are multiple bookmarks that can be played starting from the current page, users can select the one they want using the *More* button in the drop-down message. The most frequently used smart bookmark is chosen initially by default.

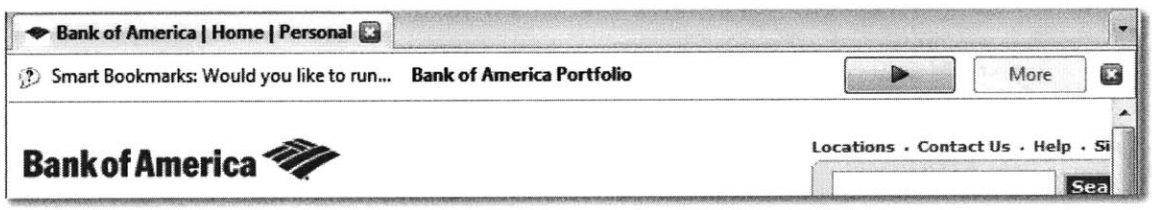


Figure 4.12 An example of a Smart Bookmarks reminder

4.5 Editing a Bookmark

Smart Bookmarks supports a variety of methods for editing a bookmark after it has been created. Users may decide to modify a bookmark for one of several reasons, including: fixing a bookmark that has become broken due to a web page change; making quick changes to parts of a bookmark without the need to completely recreate it; copying or moving commands from one bookmark to another; streamlining or adapting an automatically generated bookmark by adding, removing, or modifying commands by hand; creating a new bookmark manually by specifying each command individually; or creating a new bookmark by explicitly recording actions while demonstrating them in the browser.

4.5.1 Manually Editing a Command

One of the smallest, and perhaps most common, changes users may want to make to their bookmarks is to simply change the input value associated with commands that enter text into a textbox or select an item from a list box. For instance, the user may have a bookmark that logs into a website using a particular user name. If that user name changes at some point in the future, the user doesn't need to recreate the entire bookmark from scratch, but can instead just edit the command that enters the user name so that it supplies the new one instead. The parts of a command's text description that can be changed in this way are highlighted, similar to hyperlinks; users can simply click on these highlighted words to edit them. For instance, if the user has a command labeled *type "Alice" into Username* textbox, she could change the user name to "Mary" by clicking on "Alice" and editing the textbox that appears (Figure 4.13). Alternately, users can make these changes by choosing *Change value* from a command's right-click menu. The editable portions of Smart Bookmarks commands include the URL specified in *go to* commands, the text entered by *type into* commands, and the list item selected by *choose* commands. Also, note that after changing the input value of a command, its associated screenshot will remain, even though it will still show the old value in the textbox or list box. The outdated image is not removed since it likely still serves its intended purpose, which is to provide context for the web page component that the command affects, which should not have changed. However, Smart Bookmarks refreshes all the images in a bookmark each time it's replayed, so a modified command's screenshot will be updated to correctly reflect its text the next time it is played.

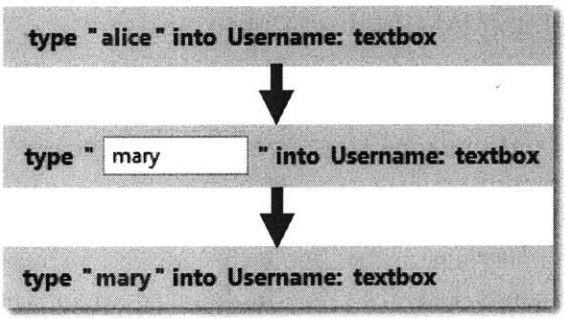


Figure 4.13 Editing a command so that it enters "mary" into the username textbox rather than "alice"

While the ability to update the values of commands is likely sufficient for users' editing needs most of the time, Smart Bookmarks also provides a way to modify the command itself, by directly editing its textual representation. While the previous method of editing only a command's value allows users to change, for instance, the text that gets placed inside a particular textbox, editing the command itself allows users to change *which* textbox gets affected, or even change the command from a *type into* command to a *click* command. To edit a command in this way, users can choose *Edit command* from its right-click menu, and make any changes necessary in the textbox that appears.

Because Smart Bookmarks utilizes Chickenfoot's implementation of syntax-free *keyword commands* [19] to support free editing of smart bookmark commands, it is not essential that edited commands follow any particular syntax. Instead, when a newly edited smart bookmark command is played, its text representation is executed as a keyword command: the system searches for a page component (such as a textbox) and an action on that component (such as typing text into it) that best match the words found in the command. This means, for example, that the command *type alice into username textbox* could be changed to any of the following commands and still likely play successfully and perform the same action as the original:

```
enter alice in username textbox
put alice into username
username = alice
```

When a smart bookmark command has been freely edited in this manner, the system can no longer reasonably assume that the command continues to affect the same web page component as before. For this reason, Smart Bookmarks automatically removes the screenshot associated with a command when it is edited in this manner. As before, however, an updated screenshot will be added the next time the command is played.

In addition to being able to freely edit existing commands, users can manually insert a completely new command into a bookmark by choosing *Add blank command* from the menu that appears after right-clicking on an opened bookmark or any of its commands. After adding a new blank command, users can then edit the command to do whatever they want just as if they were editing any other previously-existing command.

4.5.2 Copying, Moving, and Deleting Commands

It may occasionally be useful to copy and paste commands from one place to another, or delete commands that are no longer needed. Smart Bookmarks supports copying, cutting, and pasting of commands within the same bookmark, between different bookmarks, and even from an external text editor. Users can copy, cut, or delete one or more selected commands from an open bookmark by choosing the appropriate option from the right-click menu. Selected commands are highlighted in green, and multiple commands can be selected by clicking on them while holding down the Ctrl key (to select multiple commands individually) or the Shift key (to select all the commands between the last selected command and the current one). Selection behaves in the same manner as any standard text editor.

To paste commands that have been copied to the clipboard, users can just right click on the point in an open bookmark at which they would like to insert the new commands and choose *Paste before* or *Paste after* from the menu. Commands that are copied and pasted within Smart Bookmarks retain their associated screenshots as well as their text representations. In addition to using the right-click menus, the standard keyboard shortcuts for Copy (Ctrl-C), Cut (Ctrl-X), Paste (Ctrl-V), and Delete (Del) are available in Smart Bookmarks as well.

In addition to copying and pasting commands, entire bookmarks can be copied as well by right-clicking on the bookmark's name in list view and choosing *Copy*, *Cut,* or *Paste* from the menu. Bookmarks can also be deleted by choosing *Delete*, and they can be renamed by either choosing *Rename* from the menu or by clicking on the name itself when the bookmark is open.

Finally, users can also copy a bookmark or a set of commands and paste them into a standard text editor or email client, in which case the commands will appear as text. The reverse is also true: textual commands constructed outside of Smart Bookmarks can be pasted into a bookmark as well, in which case they'll appear as keyword commands that will be executed the next time the bookmark is replayed (Figure 4.14).

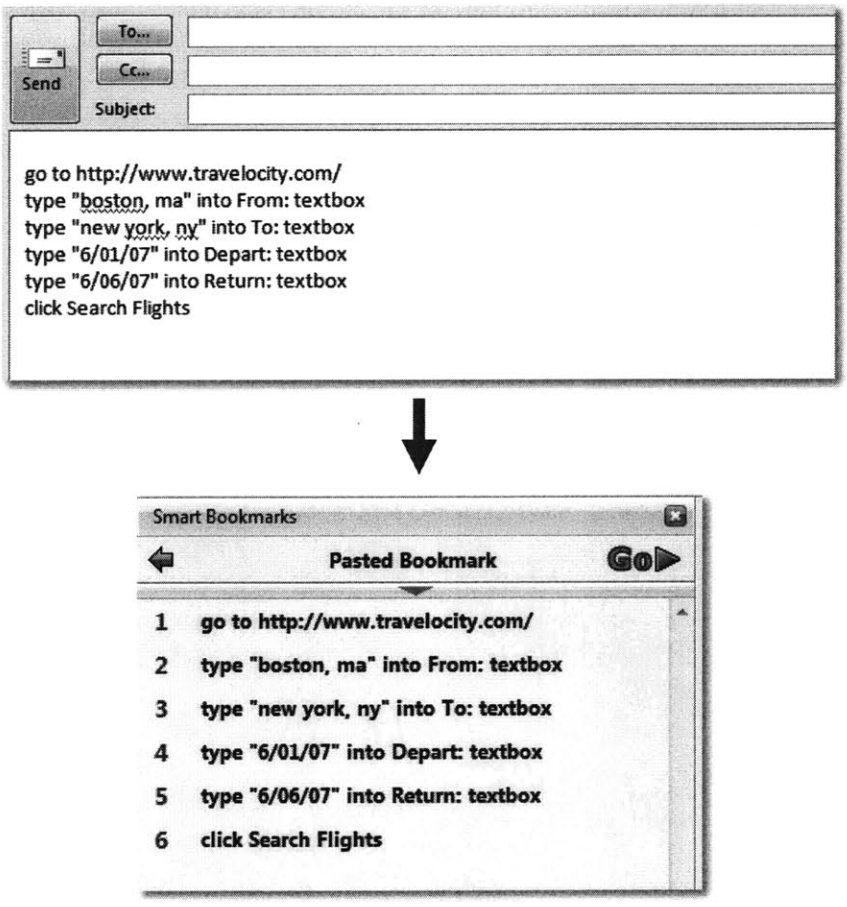
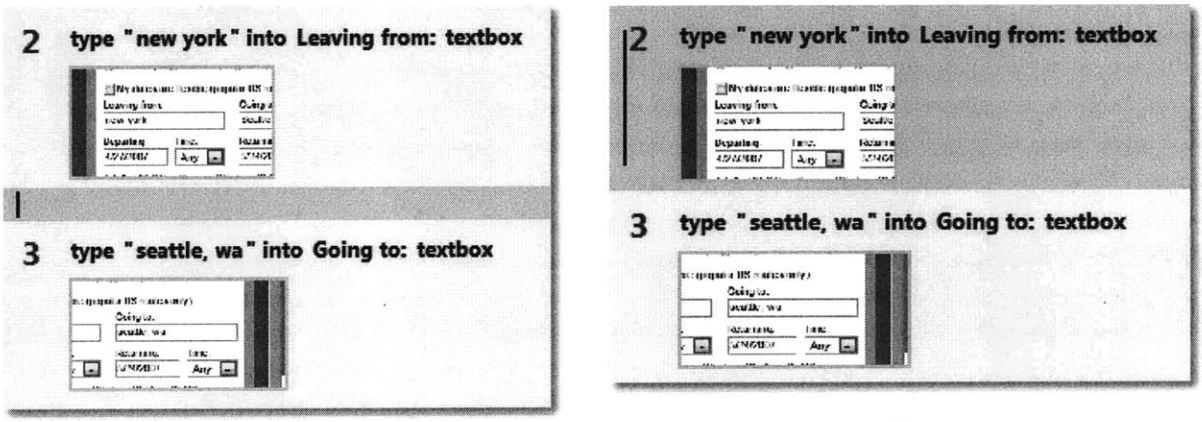


Figure 4.14 Commands can be copied into a smart bookmark from a text editor or email client

4.5.3 Live Edit

Since Smart Bookmarks was designed to be as easy and automatic to use as possible, it supports one additional method for editing bookmarks that allows users to make modifications by demonstrating them in their browser. When in *Live Edit* mode, any actions users perform in their browser automatically get recorded into the currently open bookmark at the location indicated by a flashing cursor (just like the cursor used when editing text) (Figure 4.16a). If the cursor is placed at the same location as one or more existing commands, those commands will be replaced by the next action performed in the browser (Figure 4.16b). The cursor can be moved using the Up and Down arrow keys on the keyboard, or by clicking on a command (or in between commands) in the sidebar. Live Edit mode can be toggled on or off by clicking on the *Live Edit* button in the toolbar when a bookmark is open in the sidebar (Figure 4.15).

Figure 4.15 Toggle Live Edit mode using the *Live Edit* button in the toolbar of an open bookmark



(a) (b)

Figure 4.16 Using Live Edit mode, users can modify bookmarks by demonstrating actions in their browser, which will automatically get inserted into the bookmark at the cursor location (a), or will replace an existing command (b)

Live Edit mode makes it easy for users to fix up bookmarks that break because of web page changes. If, during playback of a bookmark, Smart Bookmarks is not able to play a command for whatever reason, playback of the bookmark will stop and the broken command will be highlighted in red in the sidebar. At that point, the user can enter Live Edit mode and perform the action that the broken command should have performed in the browser. Because the Live Edit cursor will be placed at the location of the offending command by default, it will be

replaced with the corrected version. Of course, the user can move the cursor and edit some other part of the bookmark if need as well.

Another important use for Live Edit mode is to allow users to specify a smart bookmark as a macro by creating a new blank bookmark (see Section 4.2) and then demonstrating what the bookmark should do in their browser. The steps will automatically get captured in the blank bookmark, and users will end up with a smart bookmark that can exactly repeat the actions that they just performed manually. This process is effectively a form of proactive macro recording discussed earlier in Section 2.2.1.

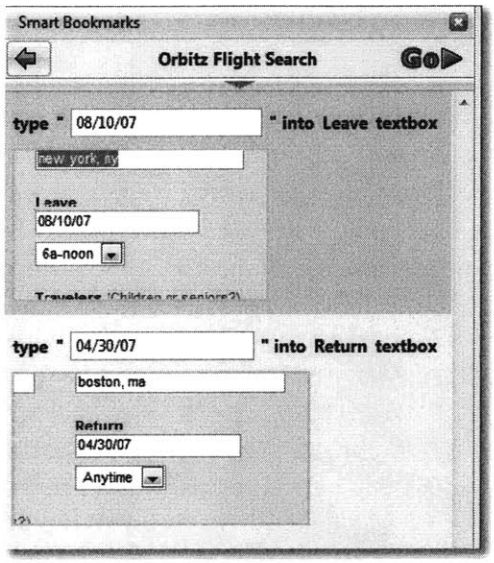
4.6 Parameterizing a Bookmark

Occasionally, users may find themselves wanting to automate a task with a smart bookmark where one or two of the commands should be slightly different each time the task is performed (i.e. each time the bookmark is played). For instance, a user may want to automate performing a search for a book on his local library's website, which requires him to log in first with a username and password. The user could automate this process with a smart bookmark, but needs to be able to specify a different name or author for the book he is searching for each time the bookmark is played.

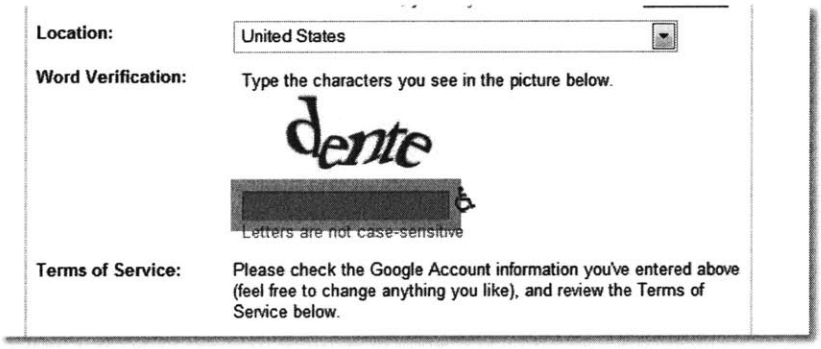
Smart Bookmarks provides this capability by allowing users to specify that the value for a particular command (the text entered into a textbox or the option chosen from a list box) should be provided every time it is played. The system supports two different methods with which users may choose to provide the values for parameterized commands: they can specify a command's value up front, before playback of the bookmark begins (along with all the other commands for which this option was chosen); or they can specify its value using the web page itself *during* playback. The first option is likely preferable the majority of the time, since it allows users to specify everything together up front, without the need to monitor the bookmark as it plays and make choices for each command separately as playback gets to them. However, the second option may occasionally be necessary for cases where users may not know the value to provide for a particular command until they see the web page itself. One common example of this scenario is specifying the text to enter for a command that involves CAPTCHA's, where users are asked to specify the sequence of letters and numbers displayed in a distorted image in order to login to a web page or perform some other action. In this case, it is impossible for the user to know in advance what text this command should enter. Commands can be parameterized using either option by right-clicking on the command and selecting either *Choose value -> Before running this bookmark* or *Choose value -> While running this bookmark.* Likewise, the default option of using a constant parameter value can be selected using *Choose value -> Always use the current value.*

When the user plays a bookmark that contains parameterized commands, Smart Bookmarks will first display a list of the commands for which the user has chosen the first option (values provided up front), if any (Figure 4.17a). Each command will be shown basically as it appears when in the context of the entire bookmark, but with the parts for which the user needs to specify values replaced with textboxes. Default values for each command are provided (the

value that was specified the last time the bookmark was played or when it was originally recorded), but the user has the option to change any of them as desired. Once satisfied, the user should again press the *Go* button to begin playback. While playing a bookmark, if the system comes to a parameterized command for which the user has chosen the second option (values provided during playback), playback will immediately stop, and the affected component will be highlighted in the context of the actual web page to indicate that the user needs to specify a value before continuing (Figure 4.17b). Once the user has entered the necessary text into the appropriate textbox or chosen an item from the appropriate list box, the *Go* button should be clicked again to continue playback.



(a)



(b)

Figure 4.17 Users can choose to enter a command's value before playback begins (a), or during playback (b)

4.7 Graphical Browsing History

In order to support automatic bookmark generation, Smart Bookmarks continuously records the actions users perform in their browser in the background. Users can view the history of

their recorded actions graphically by clicking the *View History* button in the list view toolbar (Figure 4.18). This history is analogous to the standard History feature built-in to many browsers that users are often already familiar with, but much more detailed. While the standard History only includes the web page URLs that the user has visited, the Smart Bookmarks history includes every action the user has performed in their browser, including clicking on links, entering text into textboxes, and so forth. However, the Smart Bookmarks history only includes actions recorded during the current browsing session (since Firefox was opened), rather than an indefinite amount of time in the past. A separate history is available for each tab in each window currently open in Firefox; the history for the currently visible tab is shown when the user clicks the *View History* button.

In appearance and in functionality, the history is mostly the same as a typical smart bookmark (Figure 4.19). The only real difference is that commands in the history cannot be edited or deleted. More importantly, however, users can copy commands from their history and paste them into a bookmark. Thus, Smart Bookmarks also supports manual retroactive macro recording (as discussed in Section 2.2.1), in that users can create a new bookmark by copying and pasting a sequence of commands from some point in their history after performing the corresponding actions in their browser.

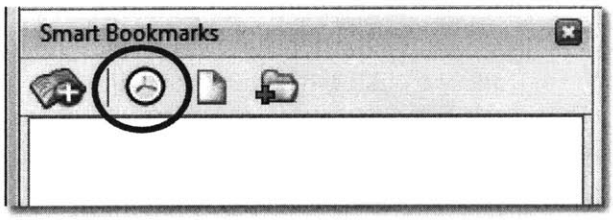


Figure 4.18 Users can view their graphical browsing history by clicking the *View History* button

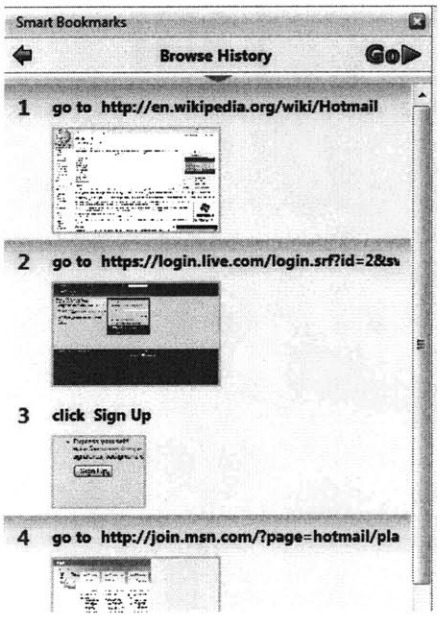


Figure 4.19 The Smart Bookmarks history is similar in appearance to a normal smart bookmark

4.8 Sharing Bookmarks

One of the primary intended uses for Smart Bookmarks is for users to share dynamically generated custom content with each other. This content could be a computer they have configured online, a shopping cart they have created, or a tutorial for how to perform some task on a website. In any case, once users have created a smart bookmark to accomplish the task or return to the content they wish to share, they should first save their bookmark to a file by opening the bookmark in the Smart Bookmarks sidebar and clicking the *Export Bookmark* button (Figure 4.20) in the toolbar. Alternately, they can choose *Save as file...* after right-clicking on the bookmark in list view.

After saving the bookmark to a file, users can email the bookmark to someone else as an attachment. Assuming the recipient of the bookmark also has Smart Bookmarks installed, he or she can then import the bookmark into their bookmark list by clicking on the *Import Bookmark* button in the list view toolbar (Figure 4.21) and selecting the smart bookmark file in the dialog box that appears. Alternately, they can choose *Add from file...* after right-clicking anywhere in list view.

If the recipient doesn't have Smart Bookmarks installed, however, a second option is for the user to instead copy the bookmark to the clipboard, and then paste it as text into a standard text editor or email client. The recipient can then simply use the bookmark as a sequence of instructions that he or she can follow by hand in a web browser.

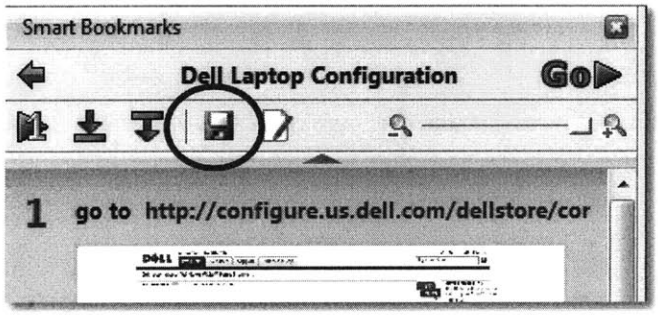


Figure 4.20 Saving a bookmark to a file

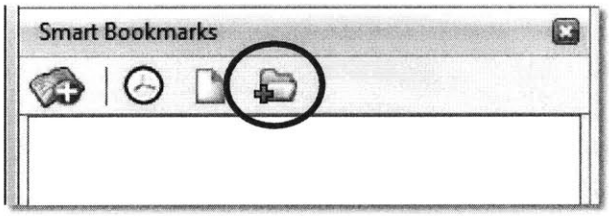


Figure 4.21 Importing a bookmark from a file

5 Implementation

Smart Bookmarks is implemented as an extension to the Firefox web browser, and is written primarily in JavaScript and XUL [12, 26]. However, some portions of the code are written in Java [29] for speed reasons, and interface with the rest of the system using Firefox's LiveConnect technology, which allows JavaScript running inside the browser to communicate with Java objects [23].

5.1 Overview of the Design

The Smart Bookmarks implementation is split into two main components: a backend that manages most of the details of the operation and state of the system; and a user interface component that initializes and controls the system's visible interface, which is provided primarily in the form of a Firefox sidebar. A graphical overview of the architecture is provided in Figure 5.1. The backend is implemented as an XPCOM (Cross Platform Component Object Model) component, which allows Smart Bookmarks to integrate with the rest of the Firefox framework [30]. There exists only one instance of the Smart Bookmarks XPCOM object whenever Firefox is running; if there are multiple Firefox windows open at the same time (and thus multiple Smart Bookmarks sidebars as well), then each window's separate version of the UI interacts with a single instance of the backend XPCOM object.

This setup allows the system to maintain a consistent state that is shared across all open instances of the Smart Bookmarks sidebar, though each sidebar can be used independently. For instance, users could replay a smart bookmark in one browser window while simultaneously creating a new bookmark from a web page in a second window. However, any changes that occur to the state of the Smart Bookmarks system as a whole, such as the addition of a new bookmark, will automatically be reflected in the UI of every open Smart Bookmarks sidebar. Additionally, because anything not directly related to the functionality of the Smart Bookmarks UI runs within the backend XPCOM object, Smart Bookmarks is able to function properly even if no instances of its sidebar is open.

When Firefox first loads for a user that has installed the Smart Bookmarks extension, it will create an instance of the Smart Bookmarks XPCOM object (internally represented as *BookmarkerService)* and register it as a service that other Firefox components can access. At this point, Smart Bookmarks sets up a variety of internal components that it needs to operate, starting with the *BookmarkManager*. The BookmarkManager is the primary component around which most of the Smart Bookmarks backend is designed, and takes care of all of the logistics involved with bookmark creation and recording the user's browsing actions. Whenever a new

Firefox window is opened, a new instance of the Smart Bookmarks sidebar UI is created. That UI component, like all other Firefox components, has access to the Smart Bookmarks XPCOM service, and tells it to register the newly created window with the BookmarkManager. The BookmarkManager is then able to attach a variety of event handlers to the browser window, so that it will be notified when tabs are created or closed, when web pages are loaded, and when the user performs some action with a web page (such as clicking on a link).

Smart Bookmarks records commands on a per tab basis by maintaining a list of *BrowseHistory* objects (one for each tab in any open Firefox window). Each BrowseHistory stores the sequence of commands that are generated from the actions the user performs in its associated Firefox tab. In addition to keeping track of the commands themselves, each BrowseHistory also creates its own *PageManager*, which handles the storage and retrieval of the DOM (Document Object Model) representations of each web page loaded in its tab. These DOM representations are used during the creation of smart bookmarks (see Section 5.4). Additionally, each PageManager sets up an *ImageManager* that handles capturing and storing the various screenshots and other images used to visually represent smart bookmarks in the sidebar.

The BookmarkManager also maintains a *BookmarkStorage* object, which takes care of storing and loading smart bookmarks on the user's hard drive, the *SideEffectsDatabase* (Section 6.1), and the *BrowseTracker*, which is tasked with initiating the bookmark reminder notifications when the user visits certain web pages. Other vital components used in the Smart Bookmarks backend include: *BookmarkRunner*, which supports bookmark playback (Section 5.5); *BookmarkGenerator*, which is used for the automatic creation of smart bookmarks (Section 5.4); *BookmarkPreview*, which provides the animated previews associated with bookmark commands; and of course the various objects used to actually represent smart bookmarks and their commands, including *Bookmark* and *Command* (Section 5.2).

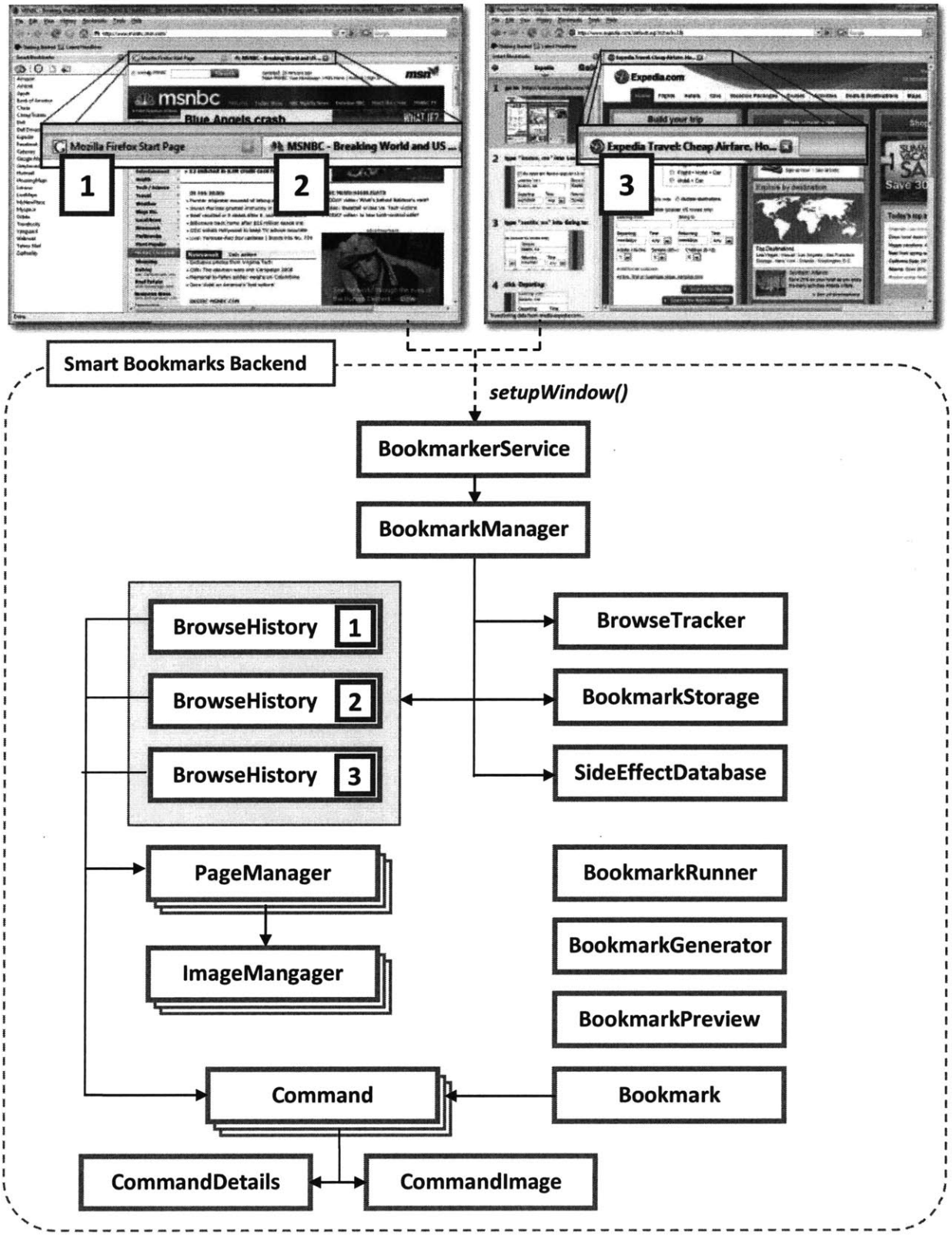


Figure 5.1 An overview of the internal Smart Bookmarks architecture. The boxed "1, 2, 3" labels identify open browser tabs and their corresponding internal BrowseHistory objects.

5.2 Internal and External Representation of Bookmarks

This section describes how smart bookmarks are represented internally while the Smart Bookmarks system is running, and how they are stored externally as files in order to persist between sessions.

5.2.1 Internal Representation

Smart bookmarks are represented internally as *Bookmark* objects. Each Bookmark has an ID, a name, a list of *Commands*, and a variety of properties, including the date and time when the smart bookmark was originally created, how many times it has been played, and a description.

A Command object represents a single smart bookmark command, corresponding to some kind of action that could be performed in the browser (such clicking a link, or entering text into a textbox). Each Command has an ID, its textual keyword command representation, a *CommandDetails* object, a list of *CommandImages*, and a variety of properties that include how the command should be displayed in the sidebar, whether the command should be executed when replaying a bookmark, whether the command has been marked as *unnecessary* (see Section 5.4.2), and how the command's value should be specified (provided before playback, provided during playback, or constant).

A Command's CommandDetails object contains information about the command that the system needs in order to play it when requested, and is usually initialized when the command is recorded. CommandDetails objects contain the following information:

- The command's action type (*go, type into, click, choose,* or *check*).
- Its label, if applicable (the descriptor used to locate the command's target component in a web page, such as a textbox for a *type into* command). *Go* commands do not have a label.
- Its value, if applicable. This will be the URL for a *go* command, *true* or *false* for a *check* command (whether to check or uncheck), the text to enter into a textbox for *type into* commands, or the item to select from a list box for *choose* commands. *Click* commands do not have a value.
- Ordinal information indicating the position of the command's target component in the web page if its label matches multiple components.
- The x,y location and dimensions in pixels of the command's target component in the web page. This information is used when showing a command's animated preview.
- The XPath of the command's target component, which specifies the location of the element in the DOM tree representation of its containing web page. This information is used when determining whether the correct matching component was found when replaying the command.
- The type of the command's target component (textbox, list box, radio button, link, etc).

All of the information provided through a CommandDetails object is separated from the rest of the Command object because it is possible for this information to be unknown for a particular

command. This will be the case any time the user has edited the textual representation of a command, or created a command by pasting text from outside of Smart Bookmarks. In these situations, the command is represented entirely by its keyword command and does not contain a CommandDetails object. The next time the command is played, its keyword command will be used to determine the correct action to perform (using Chickenfoot's keyword command interpreter), and then a CommandDetails object will be created and assigned to the command based on the results of that interpretation. If Chickenfoot is not available because it has not been installed, then Smart Bookmarks ensures that a CommandDetails object is always present by disabling editing of a command's text and pasting of commands from outside of the system.

Finally, Command objects also contain a list of CommandImage objects, which represent the various screenshots and other images associated with commands that are used when displaying bookmarks in the sidebar and to support Smart Bookmarks' animated preview feature. CommandImage objects contain the following information:

- A unique ID
- A path indicating where the actual image file is stored on disk.
- The format of the stored image file. In the current implementation, images are always stored as PNG files.
- The dimensions of the image.
- The image type. When a command is recorded, Smart Bookmarks captures images for two different purposes: to display in the sidebar when a bookmark is open, and to use when showing a command's animated preview. This property refers to which of these purposes this particular image is intended.
- The location of the image in the context of a larger *background* image used during animated previews.

5.2.2 External Representation

A smart bookmark is stored externally on disk as an XML file, with references to any image files it uses stored separately in a directory associated with the bookmark. All the XML files and images for a smart bookmark are automatically saved to a special directory within the user's Firefox profile whenever the bookmark is created or changed. An example of the format of a smart bookmarks XML file is:

```
<bookmark id="1">
   <name>Name of Bookmark</name>
   <description>Description of Bookmark</description>
   <dateCreated>Tue Feb 06 2007 22:34:36 GMT-0500</dateCreated>
   <playCount>5</playCount>
   <imageDirectory>1-1175734866363 Images</imageDirectory>
   <commands>
      <command id="0" display="display always"
               execute="true" prompt="no prompt">
         <keywordCommand>type nyc into from textbox</keywordCommand>
         <details>
            <action>click</action>
            <value>nyc</value>
```

```
            <target x="0" y="0" width="50" height="50">
               <label>from</label>
               <xpath>/HTML[1]/BODY[1]/INPUT[1]</xpath>
               <type>text box</type>
            </target>
         </details>
         <images format=".png">
            <image id="1-0" type="display"
                     x="0" y="0" width="100" height="100" />
            ...
         </images>
      </command>
      ...
   </commands>
</bookmark>
```

The XML tags used above should largely be self explanatory, and mostly refer directly to corresponding properties discussed in the Internal Representation section.

5.3 Recording the User's Browsing Actions

Clearly an important and necessary part of how Smart Bookmarks works is the ability to detect the interactions users make with web pages in their browser, and record those interactions as commands in a form that the system can understand and replay in the future.

5.3.1 Recording Commands

The first part in the process of recording actions users make in their browser is detecting when they occur. Smart Bookmarks accomplishes this by attaching JavaScript event handlers to any open Firefox windows. More specifically, the BookmarkManager is notified any time a new Firefox window is opened, and will then attach *click* and *change* event handlers to the window's *tabbrowser*. The tabbrowser is an internal Firefox component containing all of the browser's open tabs; this means that Smart Bookmarks will receive events that the user performs within web pages contained inside any tab, but not events caused by actions performed on the browser itself.

Click events are generated when the user clicks on a link, button, or some other web page element. *Change* events are generated when the user changes the value of some element, such as entering something into a textbox, choosing an item from a list box, or checking off a checkbox or radio button. When the BookmarkManager detects one of these events, it first runs it through an algorithm that analyzes the event and creates a new Command object that represents the action the user initially performed in their browser that generated the event in the first place. Smart Bookmarks then assigns the newly created command to the BrowseHistory object that corresponds to the tab in which the original event occurred. Whenever a BrowseHistory receives a new command, it sends the command to its associated ImageManager, which is responsible for capturing the images for the command from the web page (see Section 5.6).

The BookmarkManager also registers a ProgressListener on the window, and uses that listener to detect when a web page in the browser begins to load, and when it has finished loading. When Smart Bookmarks detects that a page has begun to load, it creates a new *go* command to represent the event and sends it to the appropriate BrowseHistory object. Smart Bookmarks uses begin load events rather than end load events to trigger the creation of *go* commands since some web pages can take a significant amount of time to completely finish loading, even when the majority of the page is already visible, so it's possible for users to visit and then leave a page before the end load event is generated. In this case, Smart Bookmarks will still be able to capture a *go* command for the page visit. When the page end event occurs, Smart Bookmarks signals the *go* command's associated ImageManager to capture a screenshot of the now fully-loaded web page and store that with the command (Section 5.6), and also to save the DOM representation of the web page to disk so that it can be used during bookmark creation (Section 5.4).

After detecting that an event has occurred, Smart Bookmarks needs to be able to construct a Command object that represents the action the user performed that originally triggered the event. Most of this process is straightforward; the action type can be determined by looking at the event type (click, change, or load) and the target element that the event acts on (textbox, link, radio button, etc.). Most of the rest of the details that make up a Command, such as its value, target type, target XPath, target dimensions, and target location, can be extracted directly from the target element of the event. The label that should be associated with the command is more difficult to determine, since it needs to be able to correctly and reliably identify which component in a web page the command should affect. A command's label should also be both understandable to the user and robust against changes to the structure of a web page. It is for these reasons that Smart Bookmarks follows the ideas introduced by Chickenfoot [3], and uses text taken from the visible interface of a web page as the label whenever possible. If for some reason Smart Bookmarks is unable to determine an appropriate label for a command, the XPath of the target element is used as a fallback instead.

Table 5.1 lists the various element types Smart Bookmarks understands, along with a brief summary of the process used to determine a label for each type. For all elements, before attempting to use the procedures outlined in the table, Smart Bookmarks will use the text content of a Label node that has been assigned to the target element as its label, if one is available. In general, the approach Smart Bookmarks uses to locate appropriate labels is similar to that of Koala [20].

Finally, a visual outline of Smart Bookmark's entire event capturing process is shown in Figure 5.2.

Element Type	Process used to determine a label
Text	The text itself.
Link	The text content of the link.
List box Item	The text content of the list item.
Button	The text content of the button, or if it has none, its value or title attributes.
Image	The image's alt or title attributes.
Radio button or Checkbox	The system looks at all the text nodes following the radio button or checkbox in the web page's DOM, and uses the content of the first one that does not belong to a Script or List Item element as the start of a label. After that, it will continue to append the content of any following text nodes to the initial label until it reaches any element that breaks the flow of the text (such as a Div or Br).
Textbox or List box	The system first looks at the previous sibling of the textbox or list box in the web page's DOM. If the element does not have a previous sibling or the sibling does not have any textual content, then the system will attempt to find an ancestor of the element that does have a previous sibling with textual content. Once such an element is found, a label is constructed by appending the content of any text nodes that are descendents of the element and which also do not belong to a Script or List Item element.
Anything else	If the element is not one of the above elements and is *clickable*, meaning it either has an onclick event handler assigned to it, or the cursor that appears when hovering over the element is the "hand" shape, then the system looks for the first text node or image element that is a descendant of the target element and uses the same procedures as outlined above to determine a label for either of those respective element types. If the element is not *clickable*, then the XPath of the element is used for the label as a fallback.

Table 5.1 A summary of how Smart Bookmarks determines the label that should be associated with particular element types during the command recording process

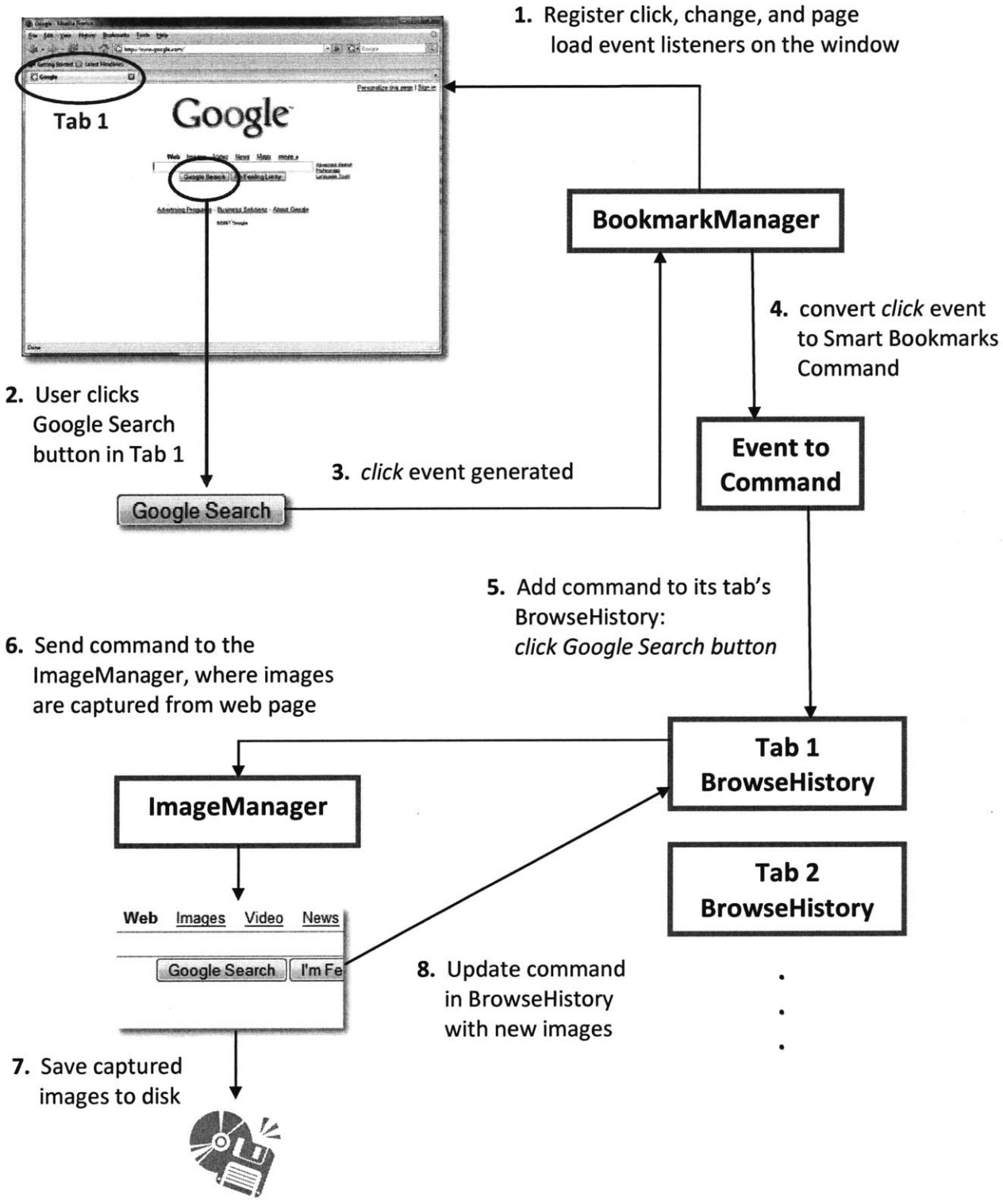


Figure 5.2 An overview of the event-to-command process in Smart Bookmarks

5.3.2 Recording Web Page DOMs

Smart Bookmarks also needs to keep a record of the structure and content of the web pages users visit while browsing the Web, in order to automatically generate smart bookmarks when needed (see Section 5.4). Smart Bookmarks accomplishes this by detecting when a web page finishes loading in the browser, and then notifies the PageManager associated with the tab the web page loaded in. The PageManager constructs a simplified XML version of the web page, based on its DOM, that strips out any information Smart Bookmarks doesn't need, such as Script elements, comments, and element attributes. The resulting XML file is then saved to a special temporary directory within the user's Firefox profile, and is automatically removed when the browser is closed.

5.3.3 Recording Images

Smart Bookmarks' interface heavily utilizes web page screenshots and other images to represent smart bookmark commands in the sidebar and to construct animated previews for each command. These images need to be captured from a web page and stored whenever a command is recorded. This is done in one of two ways, depending on whether the recorded command is a *go* command, or one of some other type (*click*, *choose*, etc.). In either case, the ImageManger associated with the browser tab from which the command was recorded is used.

For *go* commands, two images are captured. The first is a thumbnail screenshot of the entire web page, which is used to represent the command when it is displayed as part of an open bookmark in the Smart Bookmarks sidebar. The second is a full-size screenshot of the entire web page, which is used as the background image during the preview animations of any commands that occur within the web page captured by the *go* command.

The ImageManager is sent two notifications for each *go* command. The first occurs when a web page begins loading in the browser and the corresponding *go* command is created. At this point, the ImageManager simply keeps a record of the *go* command, but cannot actually capture any images for it, since the web page has not finished loading. The second notification occurs once the web page has completely loaded, and at this point the ImageManager captures the necessary images from the browser and associates them with the *go* command it was notified about previously. Smart Bookmarks captures these screenshots by using an off screen *canvas* element, drawing a portion of a web page to the canvas using Firefox's Canvas API, and then converting and saving the resulting image as a file in PNG format.

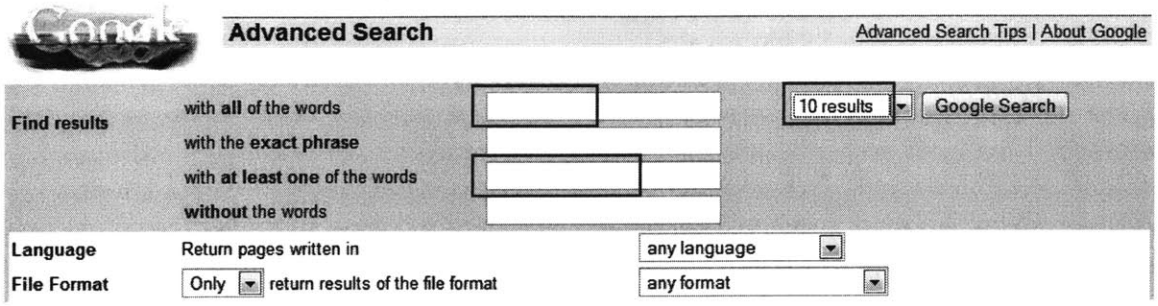
For all other commands, Smart Bookmarks captures two different types of images. The first is a thumbnail screenshot, similar to those captured for *go* commands, which includes the web page component affected by the command (a textbox or link, for instance), plus a 50-pixel border to provide the command's context within a web page.

The second image type is captured for use during animated previews. Because the previews are intended to depict web pages exactly as they looked from one recorded action to the next, with all visual changes intact, it would not be sufficient to simply capture an initial screenshot of the web page and then layer on subsequent command thumbnails, since those thumbnails are

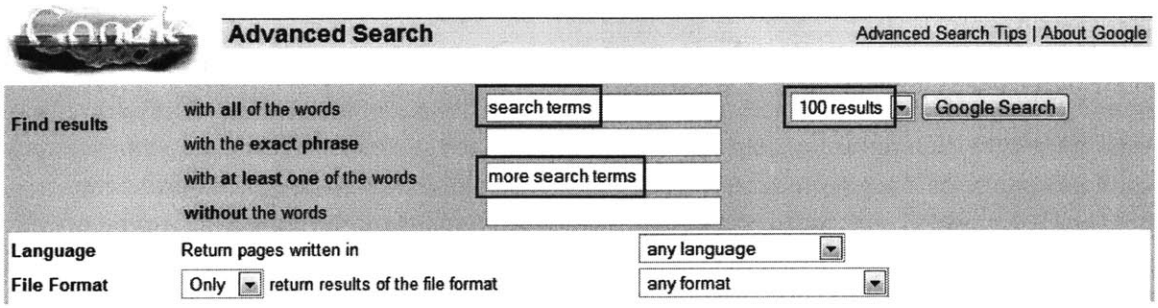
not guaranteed to capture all the visual changes that resulted from performing the command. For instance, selecting a radio button will cause two visual changes to a web page: the selected radio button will be filled in, which should be captured by the corresponding command thumbnail, but in addition to that, the previously selected radio button will become unfilled. This second change is likely not to be captured in the command's thumbnail. An obvious alternative is to recapture a screenshot of the entire web page for every new command. However, because users are likely to generate a great deal of commands over the course of a single browsing session, this solution is clearly not ideal because of the space it would require.

Instead, when a command is recorded, Smart Bookmarks captures and saves only the portions of a web page that have changed since it was originally loaded. It does this by comparing an updated screenshot of the web page at the time the command is recorded to the screenshot that was previously recorded for a prior *go* command. The algorithm used to do this takes two images and returns a list of the rectangular regions that differ between the two. Only those regions of the web page are saved and associated with the command. If the window containing a web page is resized between commands, the system will still capture the changed regions as expected as long as the layout of the page is not affected by the resizing, since the entire rendered web page is used during the image comparison rather than only the visible portion. If a web page's layout is changed due to a resizing (regions get bigger or smaller, text flows differently, etc.), then the majority of the page may appear different to the comparison algorithm, and most of the page will be recaptured. This is okay, however, and just means that the effects of the page resizing will be evident during any animated previews involving the page.

The image comparison algorithm used is written in Java for speed reasons, and makes a tradeoff between the number of differing regions returned and the amount of wasted (non-different) space that is included in those regions. For example, at one extreme a single region consisting of the entire image could be returned, with a large amount of wasted space; at the other extreme, potentially hundreds of one-pixel regions could be returned instead with zero wasted space. Smart Bookmarks balances this somewhere in the middle, so that generally only a handful of image regions need to be saved. The algorithm works by first scanning vertically down the two images, and marking continuous vertical regions over which there is at least one pixel difference between the images per horizontal line. This effectively creates zero or more rectangular regions that are guaranteed to contain the portions of the two images that differ. Each of these regions can then be scanned horizontally in the same way to find left and right boundaries for the differing parts within the region. These further constrained regions then replace the original marked regions, and the process continues until every region found by the algorithm is constrained in both directions. The resulting set of rectangular regions is then returned as the specification of the portions of the image that need to be saved. The result of running this process on an example web page is shown in Figure Figure 5.3.



(a)



(b)

Figure 5.3 The results of running the image comparison algorithm on a screenshot of the original web page (a) and a modified version of that page (b); the differing regions as returned by the algorithm are outlined.

5.4 Automatically Generating a Bookmark

When users request a smart bookmark for their current web page, Smart Bookmarks must determine a subsequence of recorded commands from the browsing history that when replayed will return to the web page or state being bookmarked. If the current web page is *bookmarkable*, meaning that it is possible to correctly revisit the page using only its URL, then this command sequence will simply start with the most recently recorded *go* command, plus any additional commands that have since changed the state of the web page. However, if the web page is not bookmarkable (whether because it is dynamically generated, the result of a form submission, relies on session cookies, or some other reason), then the system must find the most recently visited page that *is* bookmarkable, and start the sequence from there. This means that, in principle, a correct sequence of commands for a smart bookmark could simply consist of the user's entire browsing history, since it is guaranteed to start with a bookmarkable web page (the page that Firefox opened with). In practice, however, the entire history is likely to be very long, so Smart Bookmarks aims to find the shortest possible sequence of steps needed to correctly bookmark a given web page.

Smart Bookmarks maintains the user's browsing history separately by browser tab, with each history including only the commands corresponding to actions performed in its respective tab. When a new tab is created, a new history is created as well; when a tab is closed, its history is

discarded. Because it is possible for smart bookmark sequences to span multiple tabs (and thus multiple histories), it is also necessary for Smart Bookmarks to maintain some notion of how commands from separate histories are interspersed. For instance, a user may log in to an online email service, right-click on a message in the inbox and tell Firefox to open it in a separate tab (or window). The user may then decide to bookmark the email message in the new tab. Because the web page containing the email is not bookmarkable, Smart Bookmarks needs to find an earlier page that is bookmarkable, and start the bookmark's command sequence from there. In this case, the smart bookmark should start with the email site's log in page, which is bookmarkable, and from there include the steps that the user used to sign into the site, as well as the command that opened the email message. However, this sequence is now split across two separate command histories, and each may have other unrelated commands both before and after the commands needed for this particular bookmark (the user may have continued browsing to another site after opening the email in the new tab, for instance). Smart Bookmarks accounts for this by creating a link from each newly created browsing history that points back to the last command in the tab that was visible at that moment, if any. This behavior assumes that if the sequence started in a tab depends on a sequence in another tab (and it may not), then the tab that it depends on was visible right before the new tab was created.

In order to find the correct sequence of commands to include in a smart bookmark for a particular web page, the system starts with the most recent *go* command present in the page's browsing history and determines whether that command's associated URL is bookmarkable. The method Smart Bookmarks uses to determine whether a URL is bookmarkable is described in Section 5.5. If it isn't, Smart Bookmarks will continue to try successively older *go* command URLs until it finds one that is bookmarkable. The generated smart bookmark will then include that *go* command and every command that follows it in the sequence. Subsequent *go* commands are included for display purposes only, and are not executed when the bookmark is replayed. If the system reaches the beginning of a browsing history without finding a suitable bookmarkable page, it will follow the history's link back to a prior history and then continue the search from there. In practice, an appropriate starting page should always be found using this method. A diagram illustrating this process is shown in Figure 5.4.

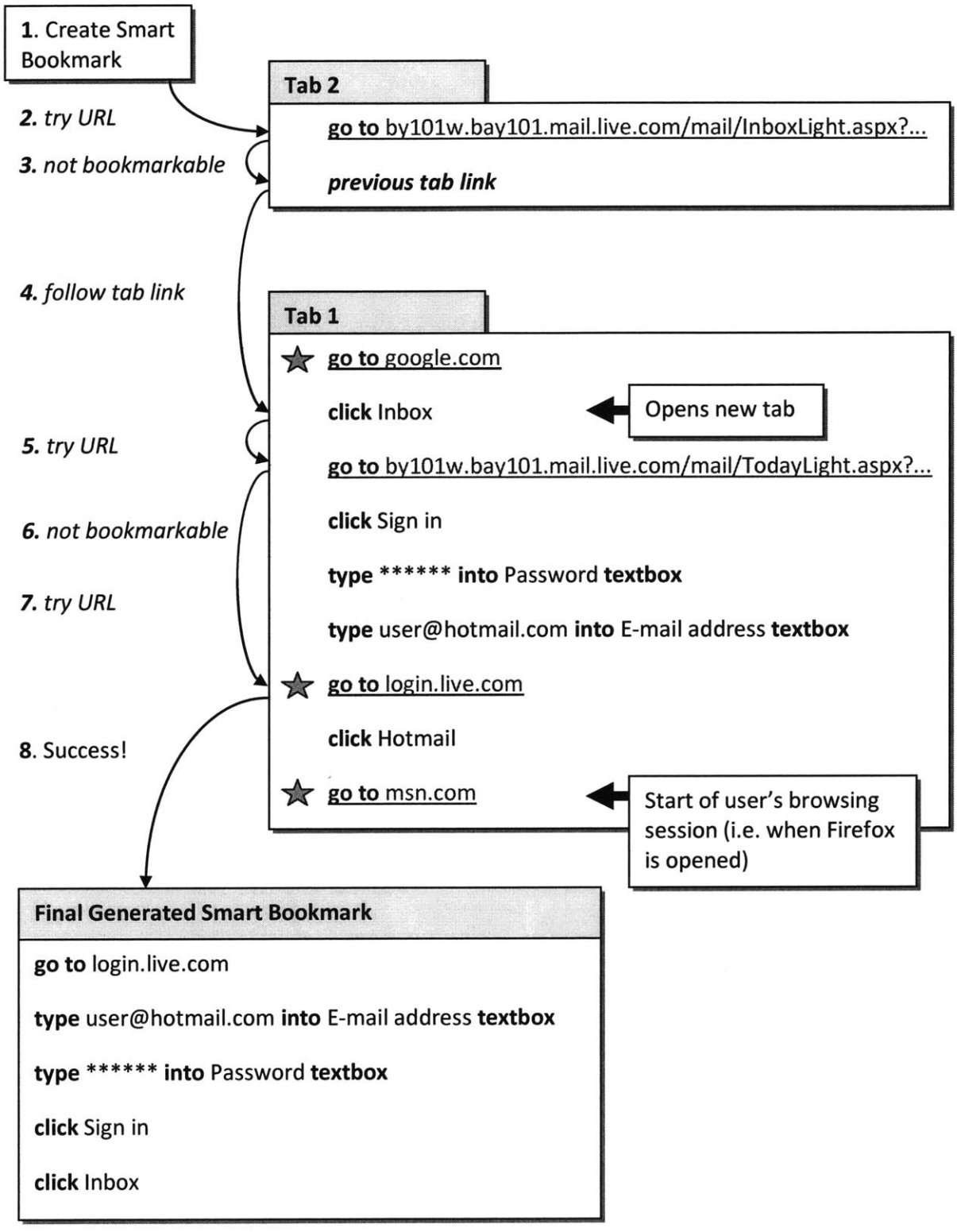


Figure 5.4 How Smart Bookmarks automatically generates a new bookmark (bookmarkable URLs are indicated with stars)

5.4.1 Determining Whether a Web Page is Bookmarkable

Smart Bookmarks considers a web page to be bookmarkable if the version of the web page as seen by the user while browsing is the same as the one the user would see if he or she visited its URL in future. Smart Bookmarks simulates a fresh, future visit to a web page by fetching the page's URL after temporarily removing the user's temporary cookies. Smart Bookmarks considers a cookie to be temporary if it is a session cookie and will expire when the browser is closed, or if the cookie is specified to expire in a time span of less than one month. The fresh version of the web page can then be compared to its original version (which was automatically saved when the user visited the page) by computing the edit-distance between their two DOMs [15]. The edit-distance is determined by the number of insert, delete, and update operations needed to transform one version of the DOM to the other. Smart Bookmarks does not require that the two versions be exactly the same, since even web pages that are bookmarkable are likely to change slightly over even short periods of time (the ads that are shown or a displayed date and time can change, for instance). Instead, the amount of difference between the two versions of a web page, as determined by their edit-distance, must fall within a certain threshold relative to the overall page size in order for them to be considered the same page.

The edit-distance algorithm attempts to match large sub-trees between the two DOMs by computing a hash for every node in both DOMs and then comparing the hashes starting from the roots. Once all the possible matches have been made, the algorithm then matches any remaining unmatched sub-trees in the two DOMs if the ratio of a node's matched children to its unmatched children is above some threshold and the matched children in both DOMs share a common parent. Based on this matching, the set of insert, delete, and update operations needed to transform the original page *P* into the fresh page *F* is computed. The cost of an operation affecting a sub-tree T is measured by the total *weight* of the sub-tree, denoted w_T, where an element node has weight 1 and a text node containing *n* characters has weight *log n*.

Therefore, the overall amount of change between the two pages *P* and *F* is the sum of the costs of the operations needed to transform *P* into *F*, divided by w_P+w_F, the total weight of the two DOMs. This value can thus be interpreted as the fractional amount of the two pages that are different. If *P* and *F* are identical, this measure is 0, indicating no change. If they are completely different, such that the smallest set of edits needed to transform P into F requires deleting of all *P* and replacing it with all of *F*, then this measure is 1. Smart Bookmarks uses a threshold of 0.1 to determine whether two version of a web page are deemed the same, and thus whether the page is considered bookmarkable. Thus, if an original web page and its fresh copy differ by less than 10 percent, Smart Bookmarks will classify the page as bookmarkable. While this threshold may seem somewhat arbitrary, in practice a legitimately bookmarkable URL tends to produce pages that are very similar, and have an edit-distance very close to zero, while URLs that aren't bookmarkable tend to vary substantially (accessing unbookmarkable URLs typically tend to lead to an error page, a completely different page within the same web site, or the start of a form sequence, for instance).

5.4.2 Removing Unnecessary Commands from a Generated Bookmark

Smart Bookmarks detects and records relatively low-level *click* events in addition to higher-level *change* events associated with things like typing text into a textbox or checking off a radio button. Click events are useful and necessary for recording some actions, such as clicking buttons or links. However, performing some actions may result in the generation of a combination of click and change events, and thus multiple Smart Bookmarks commands. For instance, selecting a single option from a list box actually leads to the creation of three separate commands: a *click* command on the list box itself (which displays the list of options), a second *click* command on the chosen option, and a *choose* command resulting from the change event that gets triggered when the value of the list box changes. Clearly the *choose* command is sufficient by itself, and the remaining two *click* commands are unnecessary. Similarly, unnecessary commands often result from the use of textboxes, radio buttons, and checkboxes as well.

Determining when commands like these are not needed and can safely be removed from a smart bookmark is complicated by the fact that web pages occasionally attach their own event listeners to page elements that trigger when the user clicks on the element, for instance. For example, travel websites often display a small calendar when the user clicks inside a date textbox. If the user creates a smart bookmark that depends on that calendar being present, Smart Bookmarks needs to ensure that its display is triggered during playback of the bookmark, just as it would have been if the user had performed the actions manually. Smart Bookmarks does this by simulating click events even when playing back non-click commands, such as *type into*, *choose*, or *check* commands.

A related problem is determining when *click* commands that aren't associated with other actions like selecting from a list box should be included in a bookmark. Including *click* commands that don't do anything, such as clicking on a non-linked image in a web page, clutters up the bookmark and makes it more difficult to understand. However, it is sometimes not obvious whether clicking on an element has an effect on a web page or not. For instance, many web sites support navigation through the use of clickable DIV elements, rather than traditional buttons or links.

Smart Bookmarks uses a combination of factors to attempt to determine the necessity of *click* commands. If the target element associated with the command, or one of its ancestors, includes an *onclick* on *onfocus* event handler, or specifies that a "hand" cursor should be displayed when hovering over the element, then the command is marked as necessary. Additionally, Smart Bookmarks tracks when various HTTP requests are imitated. If a *click* command is immediately followed by one or more of these requests, then Smart Bookmarks assumes that the associated click triggered those requests, and the command is also marked as necessary. If none of these conditions are met, then the *click* command is not included in an automatically generated bookmark. However, all commands, even those that have been deemed unnecessary, are still available in the Smart Bookmarks history (though they are grayed out in order to differentiate them). Therefore, users always have the option of manually

copying one or more of these commands into a smart bookmark if the system incorrectly flags them as unnecessary.

5.5 Playing a Bookmark

Smart Bookmarks replays a bookmark (or any series of commands), by playing each of its commands in sequence. For bookmarks that have been automatically created by the system, the sequence will start with a *go* command that loads a URL in the browser, followed by a series of *click*, *type into*, *choose*, or *check* commands. In general, however, Smart Bookmarks can play back any sequence of commands.

The system plays back non-*go* commands by first determining which element in the current web page the command is intended to affect. For instance, the command *click Google Search button* should logically affect a button in the web page that is labeled *Google Search*. Smart Bookmarks locates the appropriate element for a given command by iterating over each element of the type indicated by its *target type* property (textbox, link, radio button, etc.), and then running the same label-finding algorithm discussed in Section 5.3.1 on each of these elements. Each element in the page is assigned a score based on the edit-distance between its label as generated using the label-finding algorithm and the label associated with the command being played. The element with the highest score is designated the winner. If there are multiple elements with the same or about the same highest score, then the command's *ordinal* property is used to select one. Finally, if the winning element's XPath does not match the XPath associated with the command, then the element's score is reduced by half. If the resulting score is greater than 0.1, then Smart Bookmarks determines that the element is an appropriate match, and the system simulates the action represented by the command (clicking on the element, entering text into it, etc.). Otherwise, playback of the command fails.

A command can fail to play for several reasons. Usually, playback fails because the element the command refers to no longer exists, due to the web page having changed. Occasionally, however, a command can fail temporarily because the element it refers to will be created by JavaScript code embedded in the web page that has not finished running yet. Smart Bookmarks does not attempt to play a given command until the web page it affects has completely finished downloading from the server. However, some web pages that heavily rely on JavaScript are not always entirely loaded when Smart Bookmarks is notified that the page has been fully downloaded. JavaScript code can continue to run and create new page elements after the page has loaded, and it is very difficult to detect when this occurs. In order to mitigate this problem somewhat, Smart Bookmarks waits three seconds after playback of a command has failed and then attempts to play it again. If some JavaScript code that creates the element that the command depends on has finished running in the meantime, then the command will successfully playback on the second attempt. If after three tries playback is still unsuccessful, then the system determines that the bookmark or command sequence is broken; playback of the entire bookmark will stop, and the broken command will be highlighted in red in the Smart Bookmarks sidebar so that the user can correct it or continue on manually.

If no label was found for a command when it was originally recorded, then the system uses the XPath associated with the command to match with an element in the page. However, because XPaths are highly sensitive to changes in the underlying structure of the web page, non-labeled commands may frequently break. If no element exists in the web page at the location indicated by the command's XPath, or if an element exists but it is of the wrong type, then playback of the command automatically fails. Fortunately, Smart Bookmarks is able to determine an appropriate label for the vast majority of commands it records.

A command may potentially also have no detailed information associated with it whatsoever, except for a keyword command. This will be the case if the user has edited the command's text, created it manually, or pasted a textual command into a bookmark from outside of Smart Bookmarks. In this situation, Smart Bookmarks uses Chickenfoot's keyword command algorithm [19] to interpret the command as some action that can be performed on an element in the page (or possibly a *go* command). Smart Bookmarks can then convert this interpretation into the same format used for its automatically generated commands, and continue playback as normal.

Finally, whenever a command is successfully played, all of its associated images and target element information, including the element's XPath, are automatically updated and visually refreshed in the Smart Bookmarks sidebar if visible. This is true even for commands that originally relied on keyword command interpretation.

6 Discussion of Challenges

Smart Bookmarks faces a number of challenges that are inherent in the design of this kind of Web automation system. Specifically, this section discusses the difficulties in protecting users from undesirable side effects caused by automation, and preserving the user's security and privacy when automating websites with sensitive information.

6.1 Dealing with Side Effects

An automatically generated smart bookmark may include commands that trigger side effects. For instance, in certain contexts, simply clicking on a button in web page could send an email, register the user for an account, or even make a purchase with a credit card. If the bookmark is being used to automate a task, these kinds of side effects may be intentional; however, for bookmarks that are intended to simply return to a particular web page, side effects are generally undesirable. Even ignoring the complication of determining the user's intent in these situations, detecting commands that could potentially cause undesirable side effects is a difficult problem in itself. This is because there is no reliable method used to differentiate actions on the Web that cause side effects. That is, from a technical perspective, a form submission that initiates a search for flights on a travel website looks the same as a form submission that completes an online purchase. Systems like Smart Bookmarks can attempt to guess at the intended effect of these actions, but only the web site itself can know with any certainty which actions have side effects and which don't.

With this in mind, one solution to the side effect problem that has been proposed [7] would require web site authors to explicitly annotate actions that have side effects, using a set of standardized side effect attributes. Then a web automation system like Smart Bookmarks could check these attributes during the process of playing a bookmark to ensure that the system does not inadvertently trigger side effects that the user does not intend to occur. While this is clearly an optimal solution, it depends on wide participation by web site authors, and so would be very difficult to actually implement in practice.

Instead, Smart Bookmarks attempts to combat this problem using a combination of two methods: ensuring that users are as aware as possible of what their bookmarks do; and automatically detecting commands that can potentially trigger side effects using heuristics. Smart Bookmarks is designed to make it easy for users to view and understand bookmarks and the actions that they perform, through the use of screenshots, thumbnails, and animation. While this was done in part to make using Smart Bookmarks intuitive to use, the visuals can also help the user see when a bookmark contains commands that might cause side effects they

don't want. For instance, it is difficult for Smart Bookmarks to detect that a bookmark that adds an item to a shopping cart and then clicks a button labeled "Finalize Order" leads to a purchase; however, the user can quickly glance over the same bookmark and easily understand what it does.

However, it may not always be sufficient to depend on the user being aware of every action an automatically generated bookmark performs, especially if the bookmark is long or includes commands unrelated to the task the user was attempting to capture and automate. For this reason, Smart Bookmarks also attempts to detect actions during playback that may cause an unintended side effects whenever possible. Specifically, the system classifies *click* commands that operate on buttons or button-like images as either causing one of a range of possible side effect types, or as being side effect free. The side effect types that Smart Bookmarks can understand and detect are:

- Purchases or other monetary exchanges
- Sending or posting a message or email
- Completing a registration or subscription (for an account, mailing list, etc.)
- Adding or removing something (deleting an email message, adding an item to a shopping cart, etc.)
- Some other, generic side effect

6.1.1 Side Effect Detection User Interface

Before replaying any *click* command that operates on a button or image, Smart Bookmarks passes the command to a classifier that determines which of the above side effect types this command is likely to cause, if any. If the classifier determines that the command is likely to cause an undesirable side effect, then Smart Bookmarks displays a warning dialog box asking the user to provide a confirmation before continuing (Figure 6.1). The dialog provides a description of the specific side effect the system detected, and allows the user to choose one of three options:

- Confirm that the command will indeed cause the indicated side effect, but continue replaying it anyway
- Indicate that the system has incorrectly classified the command and that it does not in fact trigger any side effects
- Confirm the side effect classification, and do not replay the command

Users can also check a checkbox indicating that the system should always apply their choice when considering this particular command. This prevents users from having to repeatedly confirm a misclassified command each time they replay a bookmark, for instance.

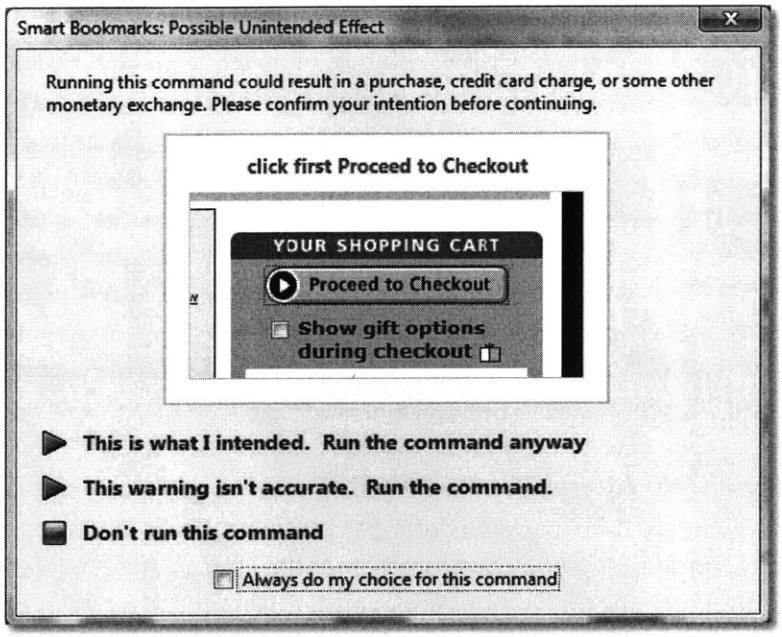


Figure 6.1 Smart Bookmarks will prompt the user to comfirm playback of a command when it detects that it may cause a possible side effect

Users also have the option of explicitly marking an action in a web page as causing a side effect during normal browsing (Figure 6.2). Smart Bookmarks will use this information when a classifying a recorded command that performs that action. Additionally, Smart Bookmarks uses data gathered from actions the user has explicitly marked as side-effecting or non-side-effecting, as well as information collected from the user's confirmation or dismissal of classifications in the warning dialog, to improve the system's ability to correctly classify side-effecting commands automatically.

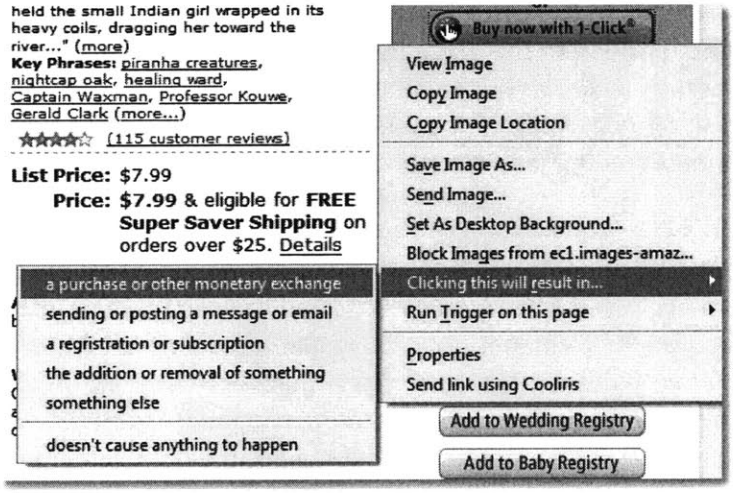


Figure 6.2 Users can mark actions as causing side effects explicitly in their browser

Finally, Smart Bookmarks also highlights the text of commands it has classified as causing side effects in red when they are displayed in the sidebar, in order to make it easier for the user to notice them before actually playing a bookmark (Figure 6.3).

Figure 6.3 Side-effecting commands are highlighted in red in the sidebar

6.1.2 Side Effect Detection Implementation

The classifier used by Smart Bookmarks to detect side effects works in two parts, using a combination of machine learning and collaborative feedback. Whenever users classify a command as side-effecting or non-side-effecting, either using the warning dialog or by explicitly marking a component in a web page, Smart Bookmarks adds a vote for that command's target element in a central database that aggregates feedback from a community of Smart Bookmarks users. Specifically, the database maintains the following information for each element:

- Its label
- Its XPath
- The URL of the web page it's located in
- The values of its various HTML attributes, including its *id*, *name*, *alt*, *title*, and *src* attributes
- The XPath of the form it is contained in
- The values of its containing form's *id*, *name*, and *action* attributes

Whenever a new vote is added to the database for a particular command, the vote is included with the total of an existing element if the above information for the new command's target element matches that of an existing element exactly. Otherwise, the element is added as a new entry in the database with a new vote total. Older entries can be removed after a specified length of time.

When the side effects classifier attempts to classify a command, it first polls the database to see if other Smart Bookmarks users have already classified the command as causing a particular side effect. However, the system does not require that the command match the information for any one element in the database exactly, since much of this information is likely to change slightly over time, even for the same visible element in a page. Instead, the system considers all of the elements included in the database that have the same domain name as the command being considered, and assigns each of those elements a score indicating how closely they match the command in question. So, for instance, a database element that has the same URL, attributes, and containing form as the command, but a different XPath, will be assigned a very high score (but not perfect) score. The score computation algorithm also considers the

individual tokens of a URL separately, so the URL *www.domain.com/a/b/c* will match more closely with *www.domain.com/a/b/d* than with *www.domain.com/e*, even though they are all different.

The total number of votes for a command is then determined by scaling the individual votes for each element in the same domain by a factor equal to its similarity score (which ranges between 0 and 1). So the command will effectively end up with a set of yes and no votes for each side effect type. If the ratio of yes votes for a particular side effect type to the total votes for that type is above a certain threshold, and is reliable (meaning a certain number of votes has been cast), then the system can classify that command as causing the side effect. If for some reason multiple side effect types meet the vote threshold, then the type with the highest percentage of yes votes is chosen.

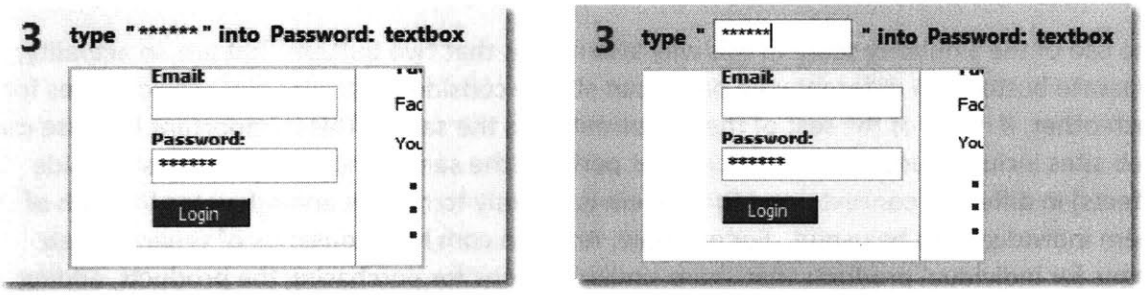
The use of the similarity score in this way also means that two buttons that are, in actuality, separate buttons on different web pages can still be considered similar and provide votes for each other, if much of the rest of their information is the same. This is important because many web sites include hundreds of buttons that perform the same function (with the same side effects) in different contexts, and the system is unlikely to receive enough votes for each of them individually to be useful. For example, Amazon.com has thousands of separate web pages for individual products that share similar buttons for purchasing the products, adding them to wish lists, etc. Because these buttons differ from each other primarily only in the URL of the page they reside in, votes for one can largely be considered the same as votes for another.

The second component of the side effect classification system comes into play when the command being considered does not garner enough votes from the database to make a reliable classification. In this case, a naïve Bayes classifier learned from examples of other buttons in the database is used instead. Whenever a vote is cast for a particular button and added to the database, all of its associated properties (label, HTML attributes, etc.) are parsed for keywords; these keywords are then added to the database as well, along with a yes or no vote for the side effect type in question. These keyword and vote combinations allow the system to learn over time the types of words and phrases that are typically associated with actions that trigger a certain side effect type. For instance, monetary exchange side effects are typically associated with keywords such as *purchase*, *buy*, *shopping cart*, or *check out*. So if the system needs to classify a button that is not yet in the database, but which includes a label like "Complete Purchase," then the system is likely to assume that clicking on this button may cause a side effect.

In the current implementation of Smart Bookmarks, the side effect database, while designed to store community feedback, is used only locally with a single user. However, it can still be useful for classification if the individual user provides the system with enough votes for the naïve Bayes classifier to function effectively. Alternately, the database could be seeded with a certain amount of appropriate data initially when the user first installs Smart Bookmarks.

6.2 Security and Privacy

Smart Bookmarks may occasionally capture commands that include users' private information, such as credit card numbers or passwords. Smart Bookmarks detects passwords automatically, by looking for textboxes that are marked with the HTML attribute *type="password"*. Detected passwords are obscured in the Smart Bookmarks interface in the usual way; for instance, a command containing a password might appear as *type ****** into password textbox* (Figure 6.4a). The screenshot associated with the command is of course already obscured, since it was captured directly from the web site. Users can edit commands containing passwords as usual, with the only difference being that a standard password textbox is used when modifying the password itself (Figure 6.4b).



(a) (b)

Figure 6.4 Smart Bookmarks automatically detects and obscures passwords when displaying commands in the sidebar

Passwords are not actually stored in a bookmark's XML file. Instead, Smart Bookmarks integrates with Firefox's built in password manager, storing any passwords needed by bookmark commands there and automatically retrieving them whenever necessary. This means that users will need to provide their Firefox master password once per browsing session if they want to use a bookmark that contains a password, for security reasons. Passwords are only added to the Firefox password manager when a bookmark is created that uses them; passwords in commands that exist only in the history are not stored in the manager, since they exist only in memory and are not saved to disk until included in an actual bookmark.

Additionally, because Smart Bookmarks does not want users to inadvertently share smart bookmarks containing passwords with someone else, the system will explicitly ask users when they export a bookmark that contains passwords if they would like to include the passwords as plain text in the bookmark file (Figure 6.5). If the user decides not to include passwords in the file, then any commands that use them will be treated as parameterized commands if someone imports the bookmark into their own browser.

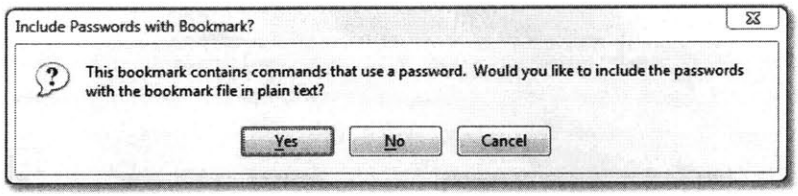


Figure 6.5 Smart Bookmarks warns the user before exporting bookmarks containing passwords

Detecting other private information that is not explicitly marked through the user of a password field, such as social security numbers or credit card numbers, is more difficult, however. Heuristics could be used to attempt to detect when bookmark commands include such information (looking for strings of four 4-digit numbers to locate credit card numbers, for instance), however this may not always be reliable. As with side effects, Smart Bookmarks attempts to mitigate the risk of users inadvertently exposing private information by providing a rich visual representation for bookmarks that clearly shows what they do and what information they contain.

7 Evaluation

This section presents an evaluation of Smart Bookmarks in three categories: the range of web sites and tasks that the system is suited for; the robustness of smart bookmarks against web page changes; and an analysis of the web page comparison algorithm Smart Bookmarks uses to determine whether a page is bookmarkable.

7.1 Automatic Bookmark Generation

To demonstrate the range of web sites and tasks that Smart Bookmarks can support, the system was evaluated against a set of 24 popular websites, using tasks that fall into one of the three usage scenarios described earlier in Section 3: accessing hard-to-reach web pages, automating repetitive tasks, and saving or sharing customized dynamically generated content.

As an example, the bookmark created for the Vanguard task consisted of these actions:

```
go to https://flagship.vanguard.com/...
type "username" into User name textbox
click Log on
type "password" into Password textbox
click Submit
click Performance
```

Each of the tasks was performed using normal browsing in a Firefox browser with the Smart Bookmarks extension installed. When the task was complete, or the desired web page was reached, Smart Bookmarks was used to automatically create a bookmark that could repeat the task. Table 7.1 lists each of the 24 tasks performed grouped into their respective usage scenarios. For each task, the table also includes the amount of time Smart Bookmarks required to generate the bookmark in seconds, and the number of commands it contained. Most bookmarks took only a few seconds to generate, but some needed up to 15 seconds, largely because the web site was slow to respond during the search for the starting point of the bookmark. The average creation time over all the tasks was 4.2 seconds, and the average number of actions was 6.

	Task Description	Time	Steps	Replay	
Accessing hard-to-reach pages					
Hotmail	Log in and view inbox	4.6	5	11 weeks	✓
MIT Webmail	Log in and view sent mail	1.8	5	11 weeks	✓
Yahoo Mail	Log in, view inbox, and mark read	2.0	9	5 weeks	✓
Facebook	Log in and view user profile	1.7	5	11 weeks	✓
MySpace	Log in and view messages	2.5	5	11 weeks	✗
Bank of America	Sign in and view account portfolio	1.7	6	11 weeks	✓
Chase	Sign in and view account summary	3.5	4	11 weeks	✓
Vanguard	Sign in and view performance page	1.7	6	11 weeks	✓
Automating repetitive tasks					
Expedia	Search for a flight	11.1	8	11 weeks	✓
Orbitz	Search for a flight	14.2	6	11 weeks	✓
CheapTickets	Search for a flight	12.4	6	11 weeks	✓
Travelocity	Search for a hotel	7.5	9	5 weeks	✓
Greyhound	Search for and choose bus schedule	1.5	7	11 weeks	✓
Amtrak	Search for and choose train schedule	2.4	7	5 weeks	✓
Dell	Get updated drivers	4.1	5	5 weeks	✓
Zip Realty	Search for real estate	1.3	6	11 weeks	✓
MyNewPlace	Search for apartment rentals	1.8	8	11 weeks	✓
Amazon	Access package tracking information	1.9	6	11 weeks	✓
Sharing Dynamic Content					
Dell	Customize a laptop for purchase	2.3	5	11 weeks	✗
Gateway	Customize a laptop for purchase	13.6	8	11 weeks	✗
Lenovo	Customize a laptop for purchase	2.9	5	5 weeks	✓
HousingMaps	View apartment rentals for an area	1.2	5	5 weeks	✓
GoogleMaps	Locate a specific region on the map	1.4	7	11 weeks	✓
LiveMaps	Locate a specific region on the map	3.4	7	11 weeks	✓

Table 7.1 Smart bookmarks for 24 tasks. *Time* is the time required to generate the bookmark; *Steps* is the number of commands in it; *Replay* indicates which bookmarks could be successfully replayed after the specified number of weeks.

The table only shows successful bookmarks, to demonstrate the range of the Smart Bookmarks system; however it is possible for the system to incorrectly identify a suitable starting page for bookmarks created for some web sites. Some web sites may embed session IDs in the URL of a dynamically generated web page, rather than maintaining state through the use of cookies. In this situation, Smart Bookmarks will incorrectly flag the page as bookmarkable, since it simulates a fresh fetch of a page only by disabling cookies; the session ID contained in the page's URL will remain valid, however, so the original version of the page will still be returned by the server. When the session ID expires a short time in the future, however, the URL will no longer work, and the smart bookmark that uses it will be broken. An example of a site that exhibits this problem is the Apple Store web site.

A second problem related to automatic bookmark generation occurs when long-term (non-session) cookies that affect the layout or content of a web page are created during the process of performing browsing actions that will subsequently be bookmarked. For instance, the user may visit an email site, sign into an account, and then navigate to a page that he or she wishes to bookmark. However, this particular site has placed a cookie on the user's hard drive that will keep the user logged in indefinitely in the future. Now, during the process of bookmark creation, when the system attempts to re-fetch the log in page, the URL now leads directly to the user's inbox, and so the system will incorrectly flag the URL as unbookmarkable since it no longer matches the original log in page that was stored when the user was initially browsing. This situation occurs when a user visits Hotmail and checks the "Save my email-address and password" checkbox when signing in, for instance.

A potential solution to this problem might be to compare the saved version of a page to a fresh version fetched with all cookies enabled, in addition to comparing the saved version to the page when fetched without temporary cookies as usual. If the two versions are different even with all cookies enabled, and the version fetched with all cookies is the same as the one fetched with temporary cookies disabled, then the system might assume that the changes are due to long-term cookies created during the current browsing session, and the page should thus be considered bookmarkable. This possible solution has not yet been tested or implemented in the current system, however.

7.2 Robustness of Bookmarks

In order to gauge the robustness of the 24 bookmarks to web page changes over time, an attempt was made to replay the bookmarks several weeks after they were originally recorded. 18 of the bookmarks were replayed after a period of 11 weeks, and the remaining 6 (which were created more recently) were replayed after a period of 5 weeks. Table 7.1 indicates which bookmarks were successfully replayed. All but three were able to complete successfully. The Dell customization bookmark was unable to perform an action that required selecting a checkbox for a particular model of printer because that printer was no longer listed as a possible customization. The Gateway bookmark failed because the computer model it was created to customize no longer existed. Some unreliability for bookmarks involving specific products or customizations like these can be expected, and is likely unavoidable. Due to the

nature of such tasks, it is also unlikely that users would intend to use them for long periods of time in any case.

The MySpace bookmark did not successfully replay because an advertisement page was inserted at one point in the process that required the user to click on a link to continue. Since the advertisement did not occur when the bookmark was originally recorded, playback was unable to continue without the user intervening. This particular problem could be corrected relatively easily using heuristics that check for links like "click to skip this ad", but a more general solution is more difficult. In any case, no matter what the cause of the error, Smart Bookmarks tries to make it as easy as possible for users to correct problems when they do occur, through the use of the live editing mode and by visually highlighting actions that are found to no longer work.

7.3 Web Page Comparison Algorithm

An evaluation of the web page comparison algorithm used to classify pages as either bookmarkable or unbookmarkable was performed, using the web pages involved in the creation of the bookmarks listed in Table 7.1. A total of 60 different web pages were compared. First, I manually classified the 60 pages by physically looking at the two versions of each page and deciding whether or not the page should be considered bookmarkable or not. This manual classification is then used as the base against which the results of the comparison algorithm are evaluated.

During the process of generating the bookmarks in Table 7.1, a variety of data was recorded for each web page tested: the edit-distance score the page received (ranging between 0 and 1, with 0 being more likely to be bookmarkable); the amount of time the algorithm needed to perform the computation in milliseconds; and the number of nodes contained in the DOMs of both versions of the page. A summary of the results is contained in Table 7.3, and a graph showing the distribution of edit-distance scores across all the web pages tested is shown in Figure 7.1. Pages that I manually classified as bookmarkable are shown in dark blue, and unbookmarkable pages are shown in light blue. A dotted line indicates the score boundary that the algorithm uses to divide bookmarkable and unbookmarkable pages. The algorithm correctly classified all 60 pages that were tested (Table 7.2).

The average score for the set of bookmarkable pages tested was 0.003; 63% of the pages received a score of 0; the maximum score assigned was 0.022. The average score for the set of unbookmarkable pages tested was 0.91; 72% of the pages received a score of 1; the minimum score assigned was 0.141. In general, the scores of bookmarkable pages were clustered very close to zero, with the scores of unbookmarkable pages ranging more; even there, however, all but three pages scored higher than 0.5, and the large majority were either at 1 or very near it.

		Actual	
		Bookmarkable	Unbookmarkable
Predicted	Bookmarkable	24	0
	Unbookmarkable	0	36

Table 7.2 Confusion matrix for the page comparison algorithm. There were zero incorrect matches for the 60 evaluated web pages.

	Count	Score		Time		DOM Nodes	
	Avg	Avg	StdDev	Avg	StdDev	Avg	StdDev
Bookmarkable Pages	24	0.003	0.005	138	318	2866	3827
Unbookmarkable Pages	36	0.910	0.214	73	173	2117	3067
All Pages	60	0.545	0.476	99	241	2421	3377

Table 7.3 Results from an evaluation of Smart Bookmark's web page comparison algorithm, on both bookmarkable pages and unbookmarkable pages. *Count* is the number of pages tested; *Score* refers to the average page similarity score assigned by the algorithm; *Time* is the average time the algorithm took to run in msecs; and *DOM Nodes* is the average number of nodes in the page DOMs.

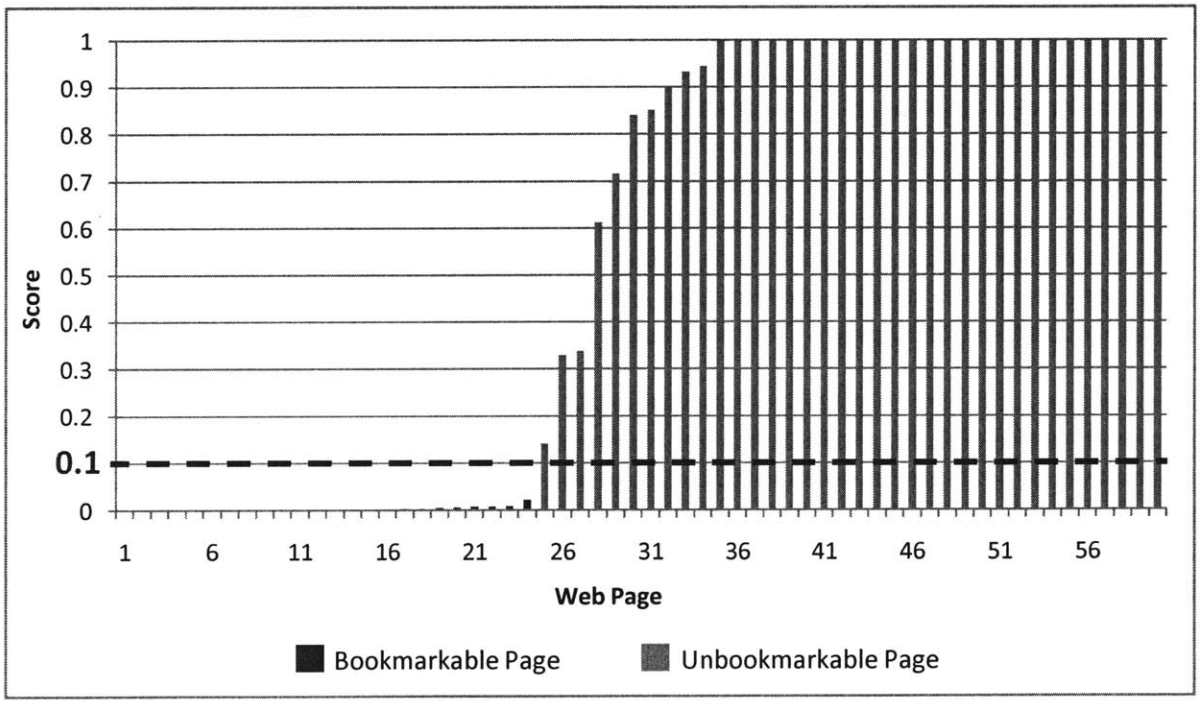


Figure 7.1 A graph showing the distribution of edit-distance scores for 60 evaluated web pages. The system designates pages with scores over 0.1 as unbookmarkable, and those with scores below 0.1 as bookmarkable.

As can be seen in Figure 7.1, a few unbookmarkable web pages scored near the bookmarkable page cutoff of 0.1. In particular, page 25 (an Amazon.com web page) received a score of 0.14 from the page comparison algorithm. This means that the system determined that the Amazon page and its refetched version differed by only 14%, even though the page is unbookmarkable and leads to different content when accessed with cookies disabled. In many of the cases in which an unbookmarkable web page receives an unusually low score, it is because the structure of the constant portions of the web page (header, footer, menus, etc.) is much more complex than the variable content portions of the web page. For instance, the Amazon.com page that received a score of 0.14 consists of a header, footer, and product menu that remain the same across all the pages that make up the Amazon website. The header, footer, and menu are constructed from approximately 1200 individual DOM nodes; however, the actual content on the particular page tested is made up of only 150 nodes. Therefore, even though nearly all of the content nodes differ between the two versions of the web page, the differences as a percentage of the entire page size is relatively small.

One potential improvement to the comparison algorithm that could help resolve this issue might be to compare the DOM of the page being tested to the DOM of the page's domain, and mark any matching nodes. Then, the algorithm could be run as before to compare the web page to a version fetched without cookies, except that any of the previously matched nodes would be ignored. For example, the Amazon web page that was tested could first be compared to the Amazon.com homepage, and the DOM nodes that make up the header, footer, and menu would be marked, since they are the same across all Amazon pages. Then, when the page is evaluated against a refetched version, only the unmatched content nodes would be considered when calculating a score. Since most of these content nodes will be different, the page should receive a score near 1.0.

One other possible improvement to the algorithm that could be considered and evaluated is to factor in the size of nodes as rendered on screen in the browser when determining their weight, rather than only considering the amount of contained text and number of child nodes. Therefore, nodes that render content that takes up a relatively large amount of screen real estate would be given more weight than a potentially more complex node structure that renders to a smaller portion of the screen.

In general, web pages are marked as unbookmarkable when the page that results from accessing its URL without cookies differs from the original version of the page that the user saw. The refetched versions of unbookmarkable web pages generally fall into one of four broad categories: error pages, the beginning of a form sequence, the website's homepage, or some other page such as a login page. The majority (51%) of the 36 unbookmarkable pages that were evaluated led to a website homepage when accessed without cookies. A chart depicting the complete breakdown for the unbookmarkable pages that were tested across these four categories is shown in Figure 7.2.

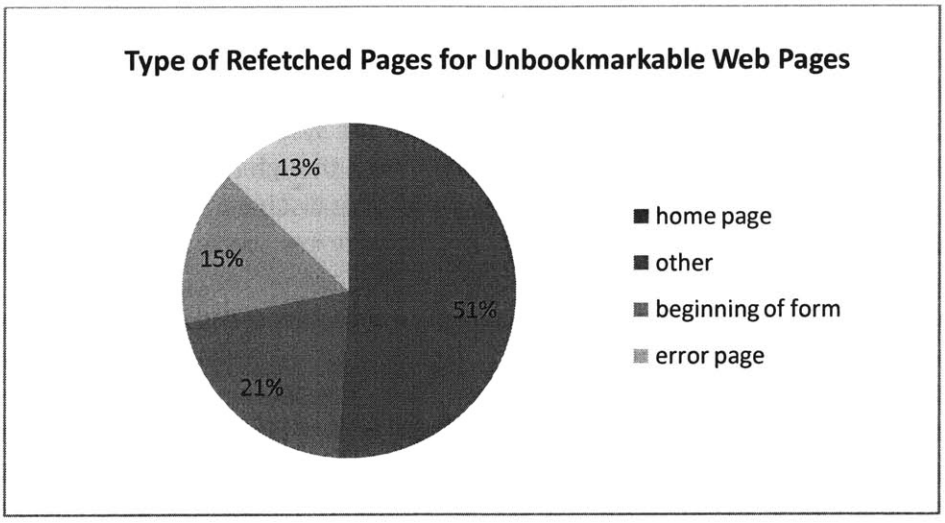


Figure 7.2 A chart that categories the refetched versions of unbookmarkable web pages by the type of content they lead to

Finally, a graph depicting the running time of the algorithm against the number of DOM nodes is shown in Figure 7.3. It is comparable to the expected running time of the algorithm, which is O(n) where n is the combined number of nodes in the two DOMs being compared.

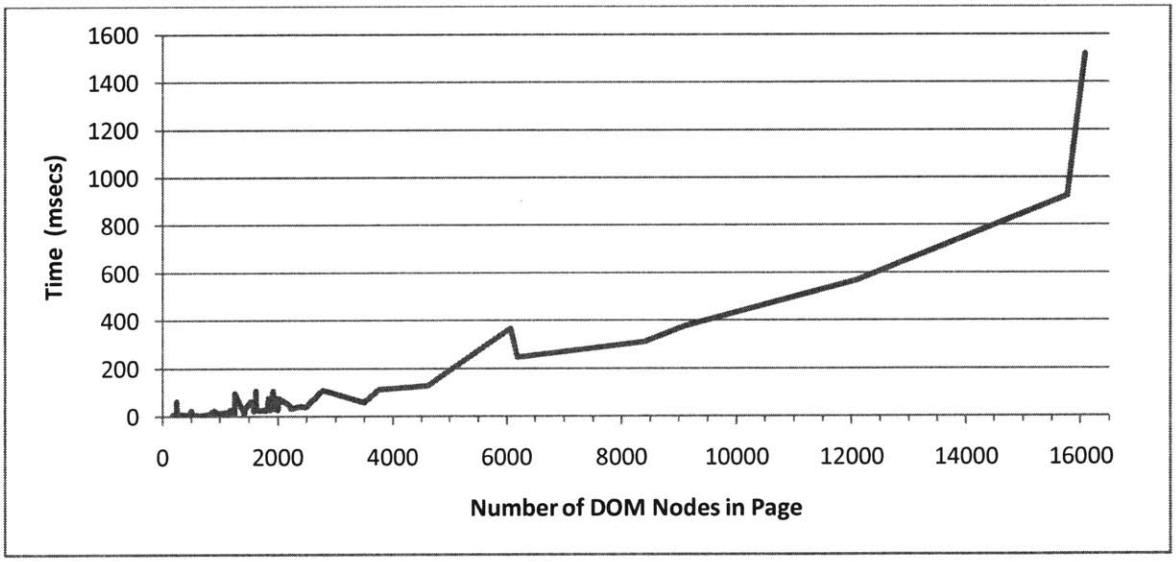


Figure 7.3 A graph depicting the running time of the page comparison algorithm against the number of DOM nodes in the pages being compared.

8 Conclusion

Smart Bookmarks is a web automation system that supports automatic, retroactive macro recording. The system allows users to create *smart bookmarks*, which consist of a script of browsing commands that can be replayed to automate a task or return to a particular web page or web application state. Smart bookmarks can be created to help with a variety of tasks, including: accessing dynamic or hard-to-reach web pages; automating repetitive Web-based tasks; creating interactive tutorials for web pages; and saving or sharing customized dynamically-generated content. Users can create a smart bookmark that returns to a particular web page in a single step by simply bookmarking the destination web page; the necessary script of commands is generated automatically.

Smart Bookmarks includes a highly visual interface for viewing, editing, and replaying bookmarks. Commands are displayed graphically in the Smart Bookmarks sidebar, in a style akin to a story board that users can easily follow to see exactly what a bookmark does. Each command is represented using a combination of textual cues, screenshots, and animation, illustrating as clearly as possible the action the command performs in the context of an actual web page. The visual feedback is intended to make smart bookmarks more understandable, inspire confidence in the validity of the automation itself, and support sharing of bookmarks with less confusion.

Smart Bookmarks incorporates several methods for editing that make it as simple and intuitive as possible for users to change bookmarks they have created, or quickly correct a bookmark that has become broken due to web page changes. In particular, the system includes a *live edit* mode that allows users to modify existing commands or add new ones to a bookmark by demonstrating the actions they would like it to perform directly in their browser. This process attempts to capture the familiarity of a standard text editor, and allows users to see their changes appear immediately in the context of the bookmark they are editing.

Smart Bookmarks provides access to a visual history of the user's browsing actions. This history allows the system to support the traditional model of macro recording, where users can create bookmarks by manually copying and pasting past commands, in addition to automatic bookmark generation. The history's presence also has the potentially beneficial side effect of making the system's operation more transparent, which in turn may increase the user's confidence in its ability to successfully create useful automations.

Finally, Smart Bookmarks introduces a possible partial solution to the problem of reliably identifying actions that may trigger undesirable side effects, through the use of a detection

mechanism based on a combination of machine learning and collaborative feedback provided from a community of Smart Bookmarks users.

8.1 Future Work

This section presents a discussion of possible additions or modifications to the Smart Bookmarks system that would provide some useful new features or improvements to the existing core implementation.

Automatic Bookmark Detection

Smart Bookmarks makes some strides toward the goal of making it as easy and automated as possible for users to create web macros that help them use the Web more efficiently, by not requiring that users know in advance that a task they are about to perform would be useful to record and save, and by not requiring that they manually choose the steps needed to do so.

Ultimately, however, it would be nice if users didn't even need to realize after the fact that recording a task would save them time or effort in the future. Instead, the system itself could monitor the users' browsing history (as it already does), analyze the data for patterns that might indicate oft-performed tasks that might benefit from a smart bookmark, and then offer to generate that bookmark automatically.

For instance, if the user often visits their bank's homepage, signs in, and navigates to a particular page within their account, the system could recognize this pattern and offer to create a bookmark that would take the user directly to the destination page. As another example, the system could automatically generate a parameterized bookmark for a user that always performs a particular flight search on a travel web site (recognizing the input fields that always change vs. the ones that rarely do).

Furthermore, one could imagine a system that is even capable of suggesting bookmarks for tasks the user didn't even explicitly perform. For instance, with the flight search example, the system could suggest bookmarks that perform the same flight search but on a completely different travel web site, perhaps by comparing the task to bookmarks other users have created and placed in some centralized repository.

Better Parameterization

There are many improvements that could be made to the command parameterization options that Smart Bookmarks currently provides. For instance, relating to the previous discussion, appropriate parameters could be automatically chosen as defaults for some commands in a generated bookmark, rather than always requiring the user to mark them manually. This could be accomplished by looking at a history of how the user has performed the task captured by a bookmark in the past.

Additionally, the system could provide a way for users to link the inputs and outputs of certain commands together. For example, the user could create a bookmark that searches multiple

different sites for the same flight; the parameters that change each time (like dates), are the same across each travel web site, so it would be nice if the user only had to supply that information once per bookmark run. Similarly, the "output" of a command (i.e. some results displayed as text on a page), could be used as the input to a subsequent command. This would require some way to visually specify the output region interactively in the web page.

Improved Reminder Feature

Right now, Smart Bookmarks' bookmark reminder feature works in a very simple way: if the user has created a smart bookmark that begins at a particular site, if the user visits that site in the future the system will display a reminder for the relevant bookmark. This could be made much more powerful and elegant, especially if integrated with the kind of automatic bookmark detection feature discussed above.

For instance, there may be many cases where the user performs some specific sequence of steps many times, but doesn't think to create a bookmark for it, or may even feel that the task is simple enough that creating a bookmark would be overkill. A similar situation might exist wherein the user performs a task once and doesn't plan to do so again, or plans to do so very infrequently. The user may not feel it is necessary to create a bookmark for a task that he is only going to perform a handful of times.

In all of these situations, it would be useful if the system provided a way to quickly play a large number of mini bookmarks that haven't explicitly been created and that are suggested within the context of the user browsing a web page. These bookmarks may not even be bookmarks that automate a complicated task, but simply links to a bookmarkable URL, akin to traditional bookmarks. As a personal example illustrating a possible use of this kind of system, I frequently access some web pages by doing a search for the web site name, loading the site from the search results, and then browsing to the page I want by clicking on links within the web site. The page may be bookmarkable using traditional URL-based bookmarks, but it's easier for me to just perform the search process when I want to access it, rather than remembering or taking the time to bookmark the page and then having to find the page in a long list of bookmarks when I want to access it, which may be relatively infrequently. In this situation (and potentially many others), I would find it useful if, instead, when I perform a search for the website's name, I'm presented with an unobtrusive list of possible destinations I may have had in mind within the context of the web page itself. I could then click to go directly to the page I want each time. In this situation, it isn't so much the task itself that is so complicated (I may save only a couple of seconds by doing this), but the context in which I'm presented with the time-saver (i.e. links within the web page itself when I start the process, rather than bookmarks available externally in a side bar that I have to manually open and then find). And over time, and over many different web pages and tasks, these little one or two second time savers could potentially add up.

Integration with Firefox Bookmarks

A less ambitious, more short-term additional feature would be to integrate Smart Bookmarks somewhat with the traditional bookmarking facility provided by Firefox. The names of the smart bookmarks that users create could appear alongside their traditional bookmarks in Firefox's bookmark list (differentiated in some way, of course). The user could then click on a smart bookmark just as if it was a URL-based bookmark, and it would automatically open in the Smart Bookmarks sidebar and begin playing. Deleting, renaming, and organizing bookmarks within the Firefox bookmarking facility could be integrated as well.

More Evaluation

The side effect detection mechanism employed by Smart Bookmarks is a new feature that hasn't been thoroughly analyzed or evaluated, in part because it is difficult to test in any kind of automated way. It is very likely that the specific weighting formula used by the algorithm to automatically classify elements that are not present in the database will need to be adjusted for the detection to operate reliably. This kind of evaluation and refinement will likely require a large sample of both side-effecting and non-side-effecting page elements to analyze. A second component of the detection system that may need to be adjusted is the algorithm used to assign votes to an element by computing the similarities with elements that have already been added to the database by users. This should be analyzed to ensure that elements are receiving appropriate similarity scores based on their amount of similarity in the context of causing side effects within a given web site.

A second useful study would be to evaluate the robustness of automatically generated smart bookmarks to web page changes over time. This kind of study would simply require the creation of a number of bookmarks across a variety of web sites, and then attempting to replay those bookmarks at several periods in the future to see if they still function as expected, and if not, why they broke. A version of this kind of robustness evaluation over a relatively short time scale is discussed in Section 7.2.

References

[1] Krulwich, B. "Automating the Internet: Agents as User Surrogates." *IEEE Computing*, July/August 1997.

[2] Anupa, V., Freire, J., Kumar, B., and Lieuwen, D. "Automating Web Navigation with the WebVCR." *Proc. WWW9*, May 2000.

[3] Bolin, M., Webber, M., Rha, P., Wilson, T., and Miller, R.C. "Automation and Customization of Rendered Web Pages." *ACM Conference on User Interface Software and Technology (UIST)*, 2005.

[4] Bolin, M. "End-user Programming for the Web." MIT MEng thesis, 2005

[5] Safonov, A., Konstan, J., and Carlis, J. "Towards Web Macros: a Model and a Prototype System for Automating Common Tasks on the Web." *Proceedings of the 5th Conference on Human Factors & the Web*, June 1999.

[6] Safonov, A., Konstan, J., and Carlis, J. "Beyond Hard-to-Reach Pages: Interactive, Parametric Web Macros." *Proceedings of the 7th Conference on Human Factors & the Web*, June 2001.

[7] Safonov, A., Konstan, J., and Carlis, J. "End-user Web Automation: Challenges, Experiences, Recommendations." *Proceedings of WebNet*, 2001.

[8] Sugiura, A. and Koseki, Y. "Internet Scrapbook: Automating Web Browsing Tasks by Demonstration." *Proceedings of ACM Symposium on User Interface Software and Technology*, 1998.

[9] Kurlander, D. and Feiner, S. "A history-based macro by example system." *Proc. UIST 1992*, pp. 99-106.

[10] Flanagan, D. JavaScript: The Definitive Guide. *O'Reilly*, 2001.

[11] W3C. "Document Object Model (DOM)." www.w3.org/DOM.

[12] JavaScript 1.5. www.mozilla.org/js/js15.html.

[13] Faaborg, A., Lieberman, H. "A Goal-Oriented Web Browser." *CHI 2006*.

[14] Faaborg, A. "A Goal-Oriented User Interface for Personalized Semantic Search." MIT M.S. thesis, 2006.

[15] Cobena, G., Abiteboul, S., and Marian, A. "Detecting changes in XML documents." *Proc. ICDE 2002*, pp. 41-52.

[16] Dontcheva, M., Drucker, S., Wade, G., Salesin, D., and Cohen, M.F. "Summarizing personal web browsing sessions." *Proc. UIST 2006*, pp. 115-124.

[17] Fujima, J., Lunzer, A., Hornbæk, K., Tanaka, Y. "Clip, connect, clone: combining application elements to build custom interfaces for information access." *Proc. UIST 2004*, pp. 175–184.

[18] Huynh, D.F., Miller, R.C., and Karger, D. "Enabling Web Browsers to Augment Web Sites' Filtering and Sorting Functionality." *Proc. UIST 2006*, pp. 125-134.

[19] Little, G. and Miller, R.C. "Translating Keyword Commands into Executable Code." *Proc. UIST 2006*, pp. 135-144.

[20] Little, G., Lau, T., Cypher, A., Lin, J., Haber, E., Kandogan, E. "Koala: Capture, Share, Automate, Personalize Business Processes on the Web." *Proc. CHI 2007*, to appear.

[21] Modugno, F. and Myers, B. "Graphical representation of programs in a demonstrational visual shell – an empirical evaluation." *TOCHI*, 4(3), Sept. 1997, pp. 276-308.

[22] Chickenfoot. groups.csail.mit.edu/uid/chickenfoot/.

[23] LiveConnect. http://www.mozilla.org/js/liveconnect/.

[24] W3C. "XML Path language (XPath) Version 1.0," 1999.

[25] Mozilla. XUL Reference. http://developer.mozilla.org/en/docs/XUL_Reference.

[26] XULPlanet. http://www.xulplanet.com/.

[27] Mozilla. Mozilla Development Center. http://developer.mozilla.org/en/docs/Main_Page.

[28] Bauer, M. and Dengler, D. InfoBeans-Configuration of Personalized Information Services. *Proceedings of ACM IUI 99.*

[29] Sun. Java SE 6.0 API Specification. http://java.sun.com/javase/6/docs/api/

[30] Mozilla. XPCOM. http://www.mozilla.org/projects/xpcom/.