

LibX IE: An Internet Explorer Add-On for Direct Library Access

Nathan Edward Baker

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science and Applications

Godmar Back, Chair
Eli Tilevich
Ali Butt

September 6, 2007

Keywords: LibX, Internet Explorer, library, toolbar

Copyright ©2007, Nathan Baker

LibX IE: An Internet Explorer Add-On for Direct Library Access

Nathan Edward Baker

Abstract

Increasingly, students choose to use general search engines for research rather than taking advantage of the resources provided by university libraries. As university libraries offer services such as the careful selection of material and subscriptions to peer-reviewed journals, it is important that the library become integrated into research workflows. Existing technologies on library servers do not provide the level of integration we believe is most helpful to users.

LibX is a browser add-on designed to assist research by making library resources more accessible than they are through the library's own tools. It provides a client-side interface to these library services through the web browser. This integration enhances productivity and augments the user's existing information-seeking behavior.

We extended the existing Firefox version of LibX into a browser-agnostic framework, allowing LibX services to be provided on multiple browser platforms. We created a toolbar and context menu system, written in C#, to extend the existing LibX features to the Internet Explorer web browser. The primary focus of this work is on the software engineering challenges presented in creating this version.

We also designed a new framework for web localization, allowing pages viewed by the user to be modified on the client side by rules written by LibX developers, library staff, or individual users. The framework also provides a way for these rules to be distributed, updated, and composed, enhancing the browsing experience by augmenting it with additional information. The design and behavior of this framework is a secondary focus of this work.

Contents

Abstract	ii
List of Figures	vi
1 Introduction	1
1.1 Motivation	1
1.2 Solution Space	2
1.2.1 Server-Based Solutions	2
1.2.2 Proxy-Based Solutions	3
1.2.3 Client-Based Solutions	4
1.3 LibX for Firefox	5
1.3.1 Existing LibX Implementation	5
1.3.2 Web Localization in LibX	6
1.4 LibX for Internet Explorer	7
1.4.1 Goals	7
1.4.2 Programming Language Choice	8
1.4.3 Creating the Internet Explorer Version	8
1.4.4 Document Overview	9
2 Background	10

2.1	Library-Related Technologies	10
2.1.1	Z39.50 Information Retrieval Protocol	10
2.1.2	Search/Retrieve Services	11
2.1.3	Online Public-Access Catalogs	12
2.1.4	OpenURL	12
2.2	Component Object Model	13
2.3	.NET Managed Code and C#	13
2.3.1	C# and the CLR Overview	14
2.3.2	Memory Management	15
2.3.3	Delegates and Events	15
2.3.4	Attributes	16
2.3.5	The Form Designer	17
2.3.6	Deploying Assemblies and the Global Assembly Cache	18
2.4	Managed/Unmanaged Interoperation	19
2.4.1	Platform Invocation Services	19
2.5	Code Generation Through XML Schemas	22
2.6	Internet Explorer Extension Model	23
2.6.1	Hosting the Browser Control	23
2.6.2	Creating and Installing a Basic Add-On	23
2.6.3	Adding a User Interface	25
2.7	JavaScript	26
2.7.1	Execution Environment	26
2.7.2	Language Features	26
2.7.3	Browser Compatibility	27

3	Toolbar Implementation	28
3.1	Internet Explorer Toolbar	28
3.1.1	Toolbar Design Requirements	28
3.1.2	Toolbar Implementation	30
3.2	JavaScript Integration	35
3.2.1	JavaScript Requirements	35
3.2.2	Getting an Execution Environment	35
3.2.3	JavaScript/C# Interoperation Fundamentals	37
3.2.4	JavaScript Refactoring	38
3.3	Context Menu	40
3.3.1	Design of the LibX Context Menu	40
3.3.2	Officially-Supported Context Menu Extension Methods	42
3.3.3	The Subclassed Window Procedure Method	43
3.3.4	The Context Menu Preferences	46
3.4	Deployment and Updates	49
3.4.1	Initial Deployment	49
3.4.2	Automatic Updates	50
3.4.3	GAC Registration and COM Installation	51
3.5	Implementation Experiences	51
3.5.1	Differences in Browser Extension Philosophies	51
3.5.2	Handling Errors	52
3.5.3	Context Menu	52
4	Web Localization Framework	54
4.1	Requirements	54

4.2	Existing Design	56
4.3	New Design	57
4.3.1	Object Identification and Manipulation	57
4.3.2	Shared State Space	58
4.3.3	Feeds and Updating	59
4.3.4	Sandbox	59
4.4	Implications	60
5	Related Work	62
5.1	Library Tools	62
5.2	Internet Explorer Toolbars	63
5.3	Web Rewriting and Localization	64
6	Summary and Conclusion	67
6.1	Summary	67
6.2	Future Work	68
6.2.1	Internet Explorer Add-On	68
6.2.2	Web Localization Framework	68
6.3	Conclusion	69
	Bibliography	71

List of Figures

1.1	Using a server-based solution to modify information.	2
1.2	Modifying information at a proxy.	3
1.3	Client-side modification of information before it is presented to the user.	4
2.1	The Visual Studio 8 Form Designer.	17
2.2	A FlowLayoutPanel	18
2.3	The Internet Explorer add-on model.	24
3.1	The LibX Firefox toolbar.	28
3.2	The LibX Firefox toolbar, expanded to show multiple search fields.	29
3.3	The edition links menu, shown clicked with its drop-down toolbar.	29
3.4	The LibX IE toolbar with grid lines showing table cells.	34
3.5	Priming the JavaScript environment with the correct variables.	36
3.6	Refactored LibX JavaScript layers	38
3.7	Examples of contextual menu presentation.	41
3.8	The user interface for context menu preferences.	47
3.9	The user interface for updating.	50
4.1	Cues for the VT edition inserted into the pages of two different online booksellers.	54
4.2	Communication between scrapers and actions using the shared state space.	57

Chapter 1

Introduction

1.1 Motivation

Research has shown that the current generation of students prefers using general web search tools such as Google¹ over their own universities' library resources [11]. This preference is due to the increased simplicity of web search engines, which no longer require specialized knowledge or careful crafting of search terms. The phrase 'Boolean operators' is rarely seen anymore, just as the operators themselves are no longer necessary for creating all but the most specific queries. However, web search engines are designed for general searching rather than academic use. These engines usually sort results by some measure of popularity, with no regard to the source of the information or the accuracy of the information itself.

Although a conscientious student may still be able to find useful, well-referenced content and even peer-reviewed articles using a general-purpose search engine, such search results are not the norm. The average search engine results lack academic weight when compared to corresponding resources in a university library, where selectors vet material based on quality and scholarship, and journal articles are peer-reviewed. By relying on references found in a general Internet search, students deprive themselves of this careful selection and preference for higher-quality sources.

Sometimes students are even unaware of all the services a modern library has to offer, leading to the unknown services going unused. For example, a patron might not realize that a library offers a proxy service which allows access to online databases from the user's home computer. Library services can augment both research and regular browsing, but are only useful if users are aware of them.

If students are to be encouraged to pursue scholarly resources for their research rather than giving

¹www.google.com

in to the ease and flexibility of web search engines that do not provide academically-rigorous sources, it is important that libraries provide resources to patrons in ways with which they are familiar. Integrating extra context from library resources (such as bibliographic data or holdings information) into the web page the user is visiting can help augment search results by providing concrete links, making them more appealing to users who are used to using search engines for information gathering.

Even users who realize the advantage of using carefully-selected library materials for research might find traditional library systems do not work with their existing information seeking behavior [11]. For modern users, the existing fact-finding workflow predominantly involves using a web browser to traverse several different sites in search of information. If library services are only accessible through the library homepage, users must return to the library page if they wish to use the library as part of their research. If students are to use library catalogs and periodical systems, those systems must become integrated into existing fact-finding behaviors.

1.2 Solution Space

To solve the problem of integrating library services into research and data-gathering, it is necessary either to reinvent the library services and provide them in a way that is as appealing, if not more so, than search engines or to further encourage students and researchers to alter their existing workflows to incorporate library services in their current state. In our work towards solving this problem, we chose the first approach, opting to provide a compelling user experience for library resources that integrates well with students' information-seeking behavior.

To accomplish this goal, there are many possible alternatives to choose from. In a web-based solution, there are two actors: the client and the server. Resources reside on the server and are transmitted to clients over the Internet as the result of requests. For the web, this communication uses the Hypertext Transfer Protocol (HTTP). In this model, either the client or the server could be modified to provide a solution to the problem outlined above, or a third system could be inserted between client and server as a proxy.

1.2.1 Server-Based Solutions

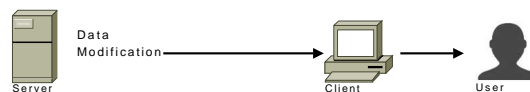


Figure 1.1: Using a server-based solution to modify information.

The first possibility for providing a solution is through a server, as shown in figure 1.1. Search

engines, with which libraries must compete, are server-based. A solution that takes place on the server is one that modifies the response to client requests before it is even sent out. For example, a server-side script which performs a catalog query could present the results to the user by integrating extra contextual results to provide extra information.

The above example illustrates providing services on the library web site itself. Additionally, it would be helpful to integrate library resources into patrons' browsing habits across multiple sites. It would be useful for an institution's server to modify data coming from a book review site, for example, to provide links to the books being reviewed.

This type of interaction is difficult to accomplish using server-based technologies. It would be possible to create a web document on the server which then provides the ability to modify other pages navigated to in a separate frame, but such interaction is not well-suited to a server-based system. Browsing pages through a frame rather than the browser interface is not a natural action.

A server-based solution may be exposed by a third party. One example of this is Google Scholar², a search engine released by Google which is designed for searching academic publications. Google Scholar allows integration of a user's library for the retrieval of search results. However, users must be on the Google Scholar server in order to take advantage of its benefits—once a user navigates away from the site, it can no longer be searched.

As another point against server-based solutions, some libraries may not be inclined to deploy such programs on their own servers, especially if they come in the form of yet another third-party system that must exist beside the other systems they already have installed. Server-based solutions would require agreement across many different organizations and corporations to succeed. Any solution which requires server support must show itself to be a superior proposition technologically and economically if it is to see widespread adoption.

1.2.2 Proxy-Based Solutions

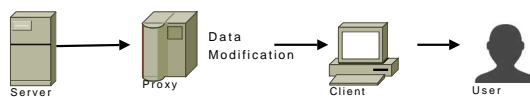


Figure 1.2: Modifying information at a proxy.

As a compromise between server and client, it is possible to integrate a system in the middle, which makes requests to the server on behalf of the user (as a proxy), then manipulates the result before returning it. By routing all traffic from every server the user visits through a single point, proxy solutions overcome the problem with server-based solutions where the server can only modify its

²scholar.google.com

own content. A proxy can also be set up by a third party, meaning that not only does it not have to reside on library servers, it does not even have to be managed by the library at all (though it can be). A proxy solution is shown in figure 1.2.

The primary problem with a proxy is that it provides a single bottleneck for all traffic. In order for a proxy to manipulate responses sent from any server a user may visit, all of a user's traffic must go through that proxy. When applied to the entire user base of a university library, a proxy solution requires a significant investment in hardware and bandwidth to provide a good browsing experience.

Use of a proxy also has privacy implications. Users may not want their browsing history to be available to libraries, and personal information would by necessity be sent through the proxy as well. Even if users trust the library to not reveal or even to discard this information, if the communication in and out of the proxy is not secure then it could be vulnerable to outside interception or interference.

Finally, a proxy must perform special processing. As a third party to the client-server transaction, it has neither the semantic information the server possesses about the site nor the parsed result the web browser generates for the client. A proxy must either parse the returned HTML itself or work with the unparsed text.

1.2.3 Client-Based Solutions



Figure 1.3: Client-side modification of information before it is presented to the user.

The final place for a solution to be installed is at the client. Much like server and proxy systems, a client system can manipulate the web document seen by the user, as shown in figure 1.3. However, a program installed on a client system also provides the ability to create a rich user interface apart from just the content displayed by a web page. In the same way that popular browsers offer a search box allowing users to search the web without actually navigating to a search engine site first, library resources can be exposed outside the context of a web document.

Use of client solutions also helps resolve privacy concerns brought up by proxies, since all the manipulation is done at the client site rather than by routing all requests and responses through a third-party proxy. Personal information can likewise be retained on the user's computer rather than being seen (and potentially saved) by a proxy system.

Although client solutions do not require any library involvement or changes to library server equipment or software, they are also outside the library's control. While a server-based solution delivers its content to anyone connected to the server, a client system requires the user to actively opt-in to the solution. Also, being set up on the client side, the library cannot alter the functionality available to its patrons unless it is involved in the process of creating the solution. Although not requiring library support can shift the burden from library staff onto an interested third-party, it also is unappealing to some libraries due to concerns over this lack of control.

1.3 LibX for Firefox

We know through experience and observation that the web browser is an important part of a normal computer user's workflow. We are also aware of the advantages and disadvantages of each of the approaches listed above. Thus, our solution to the problem is in the form of a client-side system, LibX [5]. Instead of requiring special software on the server, LibX understands the feature set of multiple library services and knows the necessary syntax to use those services. LibX brings library services and resources to the client-side by integrating with existing browsing habits and information retrieval workflows.

The LibX solution solves the problem of library service integration. LibX integrates with software running on the library server, bringing services provided by these servers into the web browser environment. We do not aim to improve the server technologies already existing or replace them with our own system; rather, we expose server-based resources and make them more accessible.

1.3.1 Existing LibX Implementation

The initial version of LibX was a browser-specific extension to the Mozilla Firefox web browser³. While I was not involved in the creation of this Firefox version of LibX, we did retain much of the core logic from the Firefox version in our work.

A Firefox extension consists of a downloadable package that can be installed in the browser. The extension model of Firefox is especially flexible considering that code loaded in extensions and the core code of the browser itself are treated the same, allowing extensions to do anything the browser itself does, and in the same manner.

Extensions are written using the JavaScript programming language, and user interfaces for extensions can be created through the Mozilla Foundation's XML User-interface Language, XUL⁴. The JavaScript is loaded into a privileged environment and executed, while XUL is rendered using the

³www.mozilla.com/firefox/

⁴<http://www.mozilla.org/projects/xul/>

Mozilla Gecko layout engine and presented to the user, interacting with the JavaScript as required.

LibX provides a user interface for querying library catalogs and databases, providing additional information for items the library might contain, and accessing library services related to the user's current task. It interfaces with the catalog software running on most library sites to allow direct searching. It also supports OpenURL [32], which provides a standard for retrieving the appropriate copy of a resource for the user (the 'appropriate' copy being the copy most-accessible to the user, such as the one held by the user's library)⁵.

LibX also integrates with a library's proxy server, if available. A common method of providing institution-wide access to a database to which the institution subscribes is to restrict access to only those IP addresses which the institution has been assigned. Because library users may not be accessing the database from a campus computer, libraries can offer a proxy service. A proxy service allows users to authenticate and thus verify that they should have access to the material and then retrieves that material for the user.

The LibX extension provides three methods of interacting with the user: a toolbar, context menu extensions, and page rewriting. The toolbar provides a convenient place to search all the catalogs, OpenURL resolvers, and other resources made available from the library. The context menu allows for context-specific searching by identifying text that the user has highlighted and displaying appropriate (and customizable) search actions. The context menu also gives proxy options, allowing links to be followed through a library proxy and for the current page to be reloaded via proxy as well. Page rewriting through DOM tree modification, referred to as web localization, is discussed in the next section.

LibX allows libraries to build individualized extensions, referred to as editions, customized specifically for the institution doing the branding. An edition includes only the catalogs offered by the supported institution. Each edition's logos and icons are defined by the institution which supports the edition.

1.3.2 Web Localization in LibX

In addition to the features listed in section 1.3.1, LibX has the ability to alter the document returned from the server to expose library resources or provide additional information to the user. It performs its alterations by manipulating the HTML document that the user sees, inserting extra information or adding extra context.

One way that LibX uses this functionality is in autolinking. LibX scans pages looking for text objects such as international standard book number (ISBN) or digital object identifier (DOI)

⁵Additional information about OpenURL is given in section 2.1.4.

tokens. Once found, LibX converts those tokens into hyperlinks which point to the institution's library. Users who are interested in the media represented by the object can follow the link to the library's own site.

The other common way that LibX alters page content is by inserting cues. A cue is a small icon designed to indicate that there is some sort of additional information available. This indication is more specific than autolinking, as the icon is linked to a library page and is added as the result of page- or site-specific processing and a deeper knowledge of the specific document's structure.

To give an example of cue functionality, LibX offers cues for bookseller Amazon.com⁶. By inspecting the URL and known elements within the current page, LibX can accurately determine the exact identity of the item being offered for sale. A cue can then be inserted into the page at a known location (such as next to the title of the item), providing a place to click that allows the user to see if the institution's library has a copy of that item available. This cue, being graphical, is designed to be more conspicuous than a simple hyperlink, but not so conspicuous that it interferes with the user's workflow.

1.4 LibX for Internet Explorer

As all of the above features were implemented in JavaScript and XUL for Firefox, LibX was platform-dependent. However, some libraries prefer to use Microsoft's Internet Explorer (IE) browser on their own computers, and in addition many library patrons prefer to use IE on their personal computers (Net Applications lists the Q2 2007 share of IE at 78.5%, with Firefox at 14.85%⁷). Because of strong demand plus a strong user base, it was decided to produce a version of LibX for Internet Explorer.

1.4.1 Goals

When specifying the requirements for this project, we set three major goals for ourselves. First, as there was already a significant code base in JavaScript for Firefox, we specified that the IE version should share as much of this code as possible. This goal proved especially important given that, second, the IE version should provide all of the functionality and contain all the features of the Firefox version. Finally, we wanted to fully support the most recent version of the browser, Internet Explorer 7, while also being compatible with the still-widely-deployed version 6 where practical.

Our desire to re-use the JavaScript code was inspired by pragmatism: by keeping program logic

⁶www.amazon.com

⁷<http://marketshare.hitslink.com/report.aspx?qprid=0&qpmr=15&qpct=3&qptimeframe=Q&qpsp=33>

in shared code, we have only one code base to maintain, update, and test. While we knew ahead of time that some code would be only Firefox and some code would be only IE, it was our hope that the majority would be both. Section 3.2 deals with this topic in much greater detail.

We wanted to provide the same functionality to IE users as we do to Firefox users because we wanted the same user experience to be available to users of both browsers.

We chose to support Internet Explorer 7 because it was the most recent version (and is at the time of this writing), and because its feature set is closer to that of Firefox when compared to previous versions. However, we also acknowledged that IE 6 was still in common usage, especially on older computers, and that it would be best to support both if it would not substantially cripple one as a result.

1.4.2 Programming Language Choice

Once we established our goals, we needed to decide what tools and languages we would use to develop LibX. Though the reuse of existing LibX JavaScript was desired, the lack of a XUL engine for Internet Explorer meant that a different system for GUI and IE interaction had to be developed. We narrowed the language options for this development down to a choice between either C# or C++.

Although C++ offered several advantages, our initial preference lay with C#. While C++ is able to directly use many predefined functions and constants for IE-specific and Windows-specific programming which are not present natively in C# and does not require a framework layer, we felt that these advantages did not outweigh those of C#. C# has features and tools that enhance productivity while reducing program complexity, and it is a modern and actively-maintained language as well as an open ECMA standard. Ultimately, we decided to go with C# as the implementation language for the LibX IE functionality.

1.4.3 Creating the Internet Explorer Version

To accomplish the goals we set, we chose to create a toolbar add-on using both the C# programming language to create the IE-specific functionality and JavaScript to utilize existing LibX code. We first created a simple toolbar, ensured that it behaved as other IE toolbars behaved, and developed the UI. We then developed a framework which allowed us to execute LibX JavaScript in our own environment from C#. Finally, we used that framework to develop the context menu system, including the customization interface and the dynamic loading of menu elements.

Though we implemented very basic functionality for web localization, we also designed a completely new system for both Firefox and IE that meets our goals for this aspect of user interaction better

than the existing mechanism. We designed a system that separates object identification, such as finding an ISBN in the text of a web page, from object manipulation, such as autolinking that ISBN. The new system also supports updating scripts independently from the core LibX program and allows aggregation of scripts created by LibX developers, LibX edition maintainers, and users.

1.4.4 Document Overview

The remainder of this document details the development work and technology used in creating a LibX version for IE. Chapter 2 describes the component technologies used by the system in moderate detail. Chapter 3 gives low-level details about the various features of the system, explains what work was done, and describes how each feature was implemented. Chapter 4 describes both completed and ongoing work to provide a general-purpose web localization framework usable by LibX and other applications to modify web page content. Chapter 5 discusses other works, highlights the contributions, if any, of those related works to LibX, and points out the areas where LibX and those other works differ. Finally, chapter 6 concludes and details avenues of future work.

Chapter 2

Background

LibX depends on a number of separate technologies in order to accomplish its objectives. In the following sections, I will describe those technologies which were necessary or useful in the development of the LibX Internet Explorer add-on. Although each section will attempt to provide both a basic understanding of the technology involved and specific, detailed information about the parts that were used in LibX, no section should by any means be considered a definitive reference on its topic.

2.1 Library-Related Technologies

Aside from the technologies used at a lower level to build the program, several protocols and technologies exist in the areas of information retrieval and library science. While we did not use all of the technologies listed below in implementing LibX, those that were not are included because they presented an alternative which was considered and rejected.

2.1.1 Z39.50 Information Retrieval Protocol

The Z39.50 protocol for information retrieval is an ANSI and ISO standard protocol for querying catalogs and viewing the results. The Council on Library Resources and the National Commission on Library and Information Science commissioned the protocol [22] as a specified way for a client to search for resources in an electronic catalog.

Z39.50, while pre-web, resembles most traditional client-server protocols in that it functions through the exchange of commands and results. Z39.50 clients send commands containing attributes of the desired media. The server then creates and caches a result set, paging through it in response to commands from the client. The result set must be explicitly freed when the user

has finished.

Despite its flexibility, the number of revisions and modernizations added to the protocol, and the endorsement by organizations such as the Library of Congress, Z39.50 is not a technology we wanted to add to LibX. Different vendors have extended the protocol in different, vendor-specific ways. Also, while the protocol's Use and Relation attributes¹ allow the user to specify the desired item's properties at a high level of granularity, these fields do not necessarily match with the fields available in the catalog database. Vendors interpret these fields differently, possibly causing wide variations in the accuracy of the same search across multiple vendors [24].

These problems make a consistent implementation in client-side JavaScript difficult. Additionally, standard JavaScript does not provide a sockets library, meaning that integration with LibX would require an outside entity, such as a proxy that performs Z39.50 requests on behalf of LibX. For these reasons, Z39.50 support was not included in the initial release of LibX.

2.1.2 Search/Retrieve Services

To modernize Z39.50 and bring it onto the Web, the Library of Congress sponsored work on Search/Retrieve technology. This work resulted in the Search/Retrieval Web Service (SRW) and Search/Retrieval via URL (SRU) standards [25]. SRU defines a system of encoding a Z39.50-like query into a URL string to be submitted over the web, and SRW defines a web service using XML via SOAP² [23]. Both use the same information but exchange it in different ways.

Both SRU and SRW are built off the Contextual Query Language (CQL) [1]. The language is designed to be human-readable and -writable, while providing language support for specifying relationships and retrieving either specific records or result sets. The Search/Retrieve protocols do not create explicit result lists on the server as Z39.50 does, but rather depend on server caching of queries for efficient paging through large result sets. This feature, while depending on unspecified server-implemented caching for good performance, allowed CQL to drop primitives relating to result set management.

As SRU/SRW were not widely-used (and are still not, at the time of this writing), we rejected them for inclusion in LibX initially. If the technology sees widespread adoption, this decision could be re-evaluated and support for CQL-based technologies added to LibX.

¹Use attributes defines which properties of a book, such as author and ISBN, to search on. Relation attributes define how the given Use attributes should be matched with items in the catalog.

²SOAP is an XML-based messaging standard for web communication.

2.1.3 Online Public-Access Catalogs

Modern library systems use web-based interfaces referred to as Online Public-Access Catalogs, or OPACs. LibX integrates with the library OPAC rather than relying on underlying protocols or additional interfaces. Interfacing with OPAC software directly allows result sets retrieved by LibX to be the same page users are used to, rather than providing our own interpretation and display of the result sets (which would be the case with results returned from Z39.50 or SRW, for example).

OPACs provide an online interface for retrieving catalog information. The majority of modern OPACs are web-based with an interface for use by library patrons. Though web-based OPACs have all the flexibility of web interfaces at their disposal, they are all constrained by the limitations of server-side technologies discussed in 1.2.1, and thus cannot provide the full range of services that LibX provides.

Library OPACs are primarily designed to interface only with the user, and do not generally provide web-service interfaces for external programs. However, the web document returned by the server can be read and manipulated by services such as LibX. Using OPACs allows LibX to initiate searches and query catalogs on behalf of the user without then having to create an interface to present the results.

2.1.4 OpenURL

OpenURL is a standardized system for information (such as bibliographic information) to be codified into a ‘context object’ for machine-to-machine communication. These context objects are created by server systems referred to by Van de Sompel, et al. as *linking frameworks* [32]. By recognizing the origin of the request, a context object can be constructed that resolves to the most appropriate copy of the resource for the user, providing context-sensitivity.

An OpenURL context object, once encapsulated into a link, points to an OpenURL resolver. This resolver can provide a series of links to services available for that resource. If the context object references a book, a library might offer its users the opportunity to place a hold on that book, or view its catalog information and location within the library. If the book is not held by the library, it could offer inter-library loan services.

An example of OpenURL usage is Google Scholar, which allows libraries to register OpenURL resolvers with the service. It then creates the context object for resources it locates and directs them to the registered library, when appropriate. LibX can take advantage of this in the same way, by creating OpenURL links pointing to the user’s configured resolver for items it recognizes.

2.2 Component Object Model

Microsoft's Component Object Model (COM) [30] provides a way to effectively build software systems using components supplied by multiple vendors, using multiple languages, and even running on multiple platforms [33]. It emphasizes the separation between interface and implementation, as objects are exposed to the developer exclusively through interface contracts. Though other technologies have more recently supplanted COM (primarily .NET, as discussed in the next section), many legacy components in Windows and Internet Explorer expose themselves via COM interfaces.

COM uses the concept of interfaces (as separate from implementation) to embody a communication contract between components. An interface exposed by a component is an implicit agreement to provide behavior and handle responsibilities represented in that interface. Each component also provides a method to query for additional supported interfaces it may support. The ability to query one interface for other interfaces the underlying object supports allows for extensibility without breaking compatibility. Interfaces are immutable, forcing new functionality to be exposed through new interfaces rather than additions to an existing interface.

COM interfaces are given globally-unique identifiers, or GUIDs, as defined by the Open Software Foundation's Distributed Computing Environment [17]. Although interfaces also have symbolic names (generally beginning with I, by convention, for Interface), these names are only for programmer convenience. The interface identifier (or IID) is used when querying for an interface, which ensures that the requested interface is uniquely selectable from the available interfaces.

In Internet Explorer, browser components such as the rendered HTML page and the user interface components are exposed via COM. In addition, components which wish to interact with Internet Explorer (such as a toolbar), must also provide COM interfaces. Toolbars such as LibX must register as COM objects which implement the interfaces expected by IE. Fortunately, much of this functionality is provided automatically by the .NET framework, as discussed in the next section.

2.3 .NET Managed Code and C#

.NET provides an approach to platform-agnostic development and interoperability based on compiling source code to bytecode for a virtual machine, similar to the approach taken by Sun's Java language and Java Virtual Machine.

2.3.1 C# and the CLR Overview

Microsoft released C# in 2001 as part of the .NET Framework. .NET is a framework designed to provide support for memory-managed languages and provides unified libraries and namespaces, a dynamic garbage collector, a just-in-time compilation system, reflection, and a method to express program metadata. The Microsoft .NET framework uses the .NET CLR (Common Language Runtime) [16], an implementation of the CLI (Common Language Infrastructure), to provide these services. The bytecode specification supports multiple languages and language paradigms [31], and there exist versions of several languages which use this platform.

When a .NET language is compiled, the emitted result is not machine code but CIL (Common Intermediate Language) bytecode for the CLR [21]. CLR bytecode is hardware-independent and designed for a virtual machine. The .NET CLR does not interpret any bytecode; instead, it invokes the JIT compiler to produce native code before executing a code block. This precompilation causes the startup cost of a CIL program to be much greater than that of a native program, but the resulting native machine code can be cached, causing subsequent load times to be faster.

Files containing executable CIL bytecode are referred to as *assemblies*. An assembly is the basic unit of deployment for a .NET application. In addition, security and access boundaries exist between assemblies—for example, in addition to the standard ‘public’, ‘protected’, and ‘private’ access modifiers, there is an additional ‘internal’ modifier, which makes the class, field, property, or method public to anything in the same assembly and private to anything outside. Assemblies will be discussed further in section 2.3.6.

The CLR memory management system frees the developer from having to explicitly manage memory. Automatic garbage collection reduces complexity and removes an entire class of potential programming errors, including buffer overruns and dangling pointers. This memory management also helps make C# into a higher-productivity language than C++ for applications where complex objects will be stored, passed around, and marshaled across program and language boundaries. Finally, it eliminates memory leaks due to failure to deallocate memory. A broader discussion of .NET memory management is given in section 2.3.2.

The C# language allows (and CLR bytecode directly supports) the ability to perform unsafe memory manipulation. If the developer feels it necessary to manually allocate memory, handle pointers, and then deallocate memory, those facilities are there. It is also significant that we did not need to make use of this ability during the development of the Internet Explorer edition of LibX. While it is reassuring to the performance- or behavior-conscious developer accustomed to writing low-level code that it is possible to drop into unmanaged mode at any time, it is also indicative of the power of the language and runtime that doing so is rare.

2.3.2 Memory Management

The CLR uses a garbage collection scheme to manage memory for programs. Specifically, the CLR garbage collector (GC) works by counting references to an object, removing unreferenced memory, and compacting the heap afterward to reduce the number of active pages [28]. It also works in the presence of multithreading and multiprocessing through a ‘stop the world’ approach to garbage collection³ [29].

Optimizations include the use of generational garbage collection, which marks objects as belonging to one of three generations and adaptively (based on usage patterns) collects only the most recent generations of objects [3]. The compaction step provides additional optimization by reducing paging overhead and working set size. The ability to control the garbage collector directly through the ‘GC’ class that is part of the framework provides developers with additional control when necessary.

The garbage collector also allows weak references to be created. A weak reference does not count when the garbage collector is determining whether an object is reachable. Instead, when the object is collected, the weak reference is set to null. Weak references are an optimization which helps prevent memory leaks where a reference is kept around unnecessarily. To differentiate between weak references and normal CLR references, the latter may be referred to as a ‘strong reference’.

2.3.3 Delegates and Events

Delegates

As those who program in functional languages know (and those who program in modern scripting languages such as Ruby or JavaScript are finding out as well), the ability to treat functions and code blocks as first-class values can enhance productivity and simplify program design. The event-handling mechanism in C# also helps in simplifying complex communication between objects and modules, reducing coupling while maintaining cohesion.

As part of the C# specification⁴, the language has the ability to assign functions to variables called *delegates*. Delegates are similar to type-safe function pointers, behaving exactly as a reference to a function while also providing additional convenience features, such as a method to invoke the delegate on a separate thread without explicitly spawning and managing the thread in the code.

In the 2.0 revision, C# has also included anonymous delegates, which can be passed as closures

³‘Stop the world’ refers to garbage collection algorithms that suspend all processing on all threads while performing a collection.

⁴<http://www.ecma-international.org/publications/standards/Ecma-334.htm>

to any method requiring a delegate parameter. Anonymous delegates allow expanded flexibility over explicitly declared delegates; instead of being its own method, the anonymous delegate can be a code block inside a method scope. Anonymous delegates act as syntax-level closures, allowing access to variables declared in the static scope the delegate was defined in even after control flow has passed out of that scope.

Events

An event is a multicast mechanism for delegates: classes can subscribe to an event by associating a method with it. This association is implemented through overloading the `+=` operator, which adds the method on the right hand side to the list of methods to call when the event is invoked. Using an event-handling paradigm decreases coupling, as the event publisher does not have to worry about how many subscribers there are to the event, and the event subscribers simply have to handle whatever situation the event represents appropriately. This subscription-based mechanism is described in [15] as the Observer pattern.

2.3.4 Attributes

In addition to delegates and events, C# allows programmers to annotate classes, methods, fields, and parameters using *attributes*. Attributes take advantage of the rich metadata expressible by the CIL bytecode [4] [16] [9] to provide additional information from the programmer about a code-based entity. Some attributes may be used by the compiler, or by the runtime marshaler, or by another part of the CLR, while some attributes might be used by other programs via reflection.

Attributes are a form of declarative programming. They provide a language-level system for setting markers or setting options that will then be associated with entities that represent syntax-level elements. Since C# (via the CLR) provides reflection capabilities, programmer-provided attributes can later be accessed through introspection (from the same assembly) or by loading a foreign assembly and inspecting it.

Attributes are needed to describe the interaction with unmanaged code. Here is an example of a declaration of an unmanaged function in C#:

```
[DllImport("user32.dll", SetLastError = true, CharSet = CharSet.Auto)]
public static extern bool InsertMenuItem(
    IntPtr hMenu,
    uint uItem,
    bool fByPosition,
    [In] ref MENUITEMINFO lpmi);
```

The tokens inside the square brackets (`[]`) are attributes. The `DllImportAttribute`⁵ sets information about the method used by the dynamic linker. The `InAttribute` sets information about the `lpmmi` parameter. Though `lpmmi` is passed by reference, the `InAttribute` informs the runtime marshaler that it can treat the call as though the parameter were read-only.

Section 2.4 describes the interoperation between managed and unmanaged code in greater detail.

2.3.5 The Form Designer

The primary method of creating a user interface in C# is through using the Visual Studio Form Designer. The Form Designer is a code generator which allows GUI elements such as buttons and labels to be positioned on the window (referred to as a form) interactively. As the form and its components are manipulated, the designer generates and updates C# code which produces, at run time, the form seen by the developer at design time. Figure 2.1 demonstrates the form designer interface.

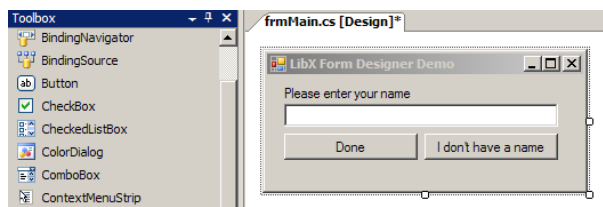


Figure 2.1: The Visual Studio 8 Form Designer.

Because developers are expected to use the form designer to lay out graphical components, positioning and sizing is done at the pixel level. This differs from the model used by some other user interface frameworks such as Tk⁶, Swing⁷, or AWT⁸, which use a layout manager to position and size elements relative to other elements rather than based on absolute values.

In version 2.0 of the .NET framework, the `System.Windows.Forms` namespace introduced two new classes, `FlowLayoutPanel` and `TableLayoutPanel`, and a new property on controls, `DockStyle`. These classes and properties are designed to assist developers in creating fluid, dynamic layouts similar to those created by relative layout managers. The `FlowLayoutPanel` inserts controls sequentially (in a flow pattern) which dynamically orders the controls based on available size. As this is best illustrated visually, figure 2.2 demonstrates a `FlowLayoutPanel`.

⁵As a convention, attribute classes end with the string 'Attribute'. To support this convention, attributes may be referenced without this string, so on a `[DllImport]` reference, if there is no `DllImport` class, the compiler interprets it as a reference to the `DllImportAttribute` class.

⁶<http://www.tcl.tk/>

⁷<http://java.sun.com/javase/6/docs/technotes/guides/swing/>

⁸<http://java.sun.com/products/jdk/awt/>

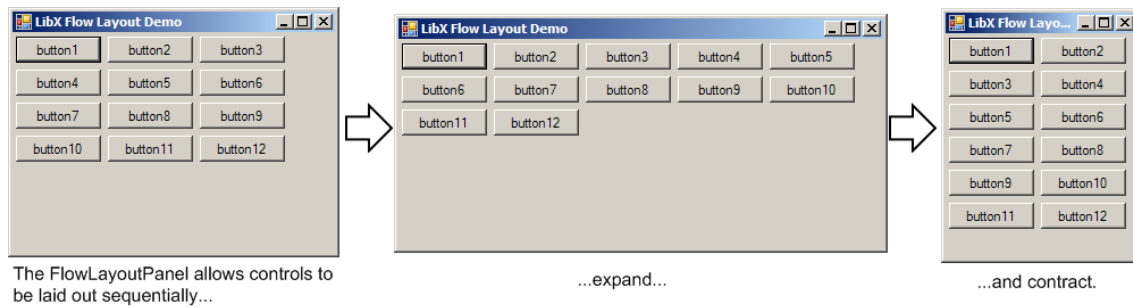


Figure 2.2: A FlowLayoutPanel

The TableLayoutPanel represents a grid of table cells, $n \times m$. Each table cell can hold at most one control. Table columns and rows can be sized based on a percentage of the total panel size, allowing them to expand and contract with the form. This allows a tabular layout that retains the same width and height ratios regardless of form size.

The DockStyle property allows controls to be docked at the top, bottom, left, or right of their container. Controls can also be given a DockStyle of Fill, causing them to take up all space not allocated to other docked controls. Docking provides an alternative to manual positioning and anchoring of form elements.

2.3.6 Deploying Assemblies and the Global Assembly Cache

Version conflicts among dependencies is a frequent problem when developing native code. When one program depends on one version of a dynamically-linked library and another program depends on another version of the same library, these different versions can conflict. If one program keeps a necessary DLL in a location it placed on the system path, it can load that library without specifying its full path. But if another program installs an incompatible version of the same library in a directory earlier on the system path, the first program ends up referencing the second, incompatible DLL, and breaks.

To fix this problem for .NET assemblies, Microsoft created the Global Assembly Cache, or GAC. The GAC provides a central repository for managed assemblies and offers a versioning system, allowing assemblies with different versions to be installed side by side. If an assembly references another assembly, it can reference a specific version of that assembly⁹. If the version is not present the dynamic link will fail, regardless of the presence of newer (and potentially incompatible) versions.

In order to uniquely identify assemblies, the concept of a ‘strong name’ was introduced. A strongly-

⁹While assemblies can be loaded without specifying a version, this re-introduces the versioning risk the GAC exists to prevent.

named assembly includes not just a name, but also a version. As the <name, version> pair is still not guaranteed to be unique (and does not provide any security against malicious assemblies masquerading as good ones), strongly-named assemblies must also be cryptographically signed. A signed assembly can be uniquely identified by its cryptographic signature, and signing the assembly also verifies that the assembly has not been tampered with.

2.4 Managed/Unmanaged Interoperation

Although the .NET framework ports or wraps the most-used parts of the Win32 API, there are times that developers want or need to call into unmanaged code. To support such interoperation, the CLR offers tools the programmer can use to interact with unmanaged code. All unmanaged interaction is considered ‘unsafe’: it must be compiled with the `/unsafe` flag and be granted special security permissions.

2.4.1 Platform Invocation Services

In order to use unmanaged DLLs and APIs directly, the CLR provides Platform Invocation Services, or P/Invoke [2]. These services include runtime marshaling, library loading, and some memory management. Section 2.3.4 above includes a code example that demonstrates the use of attributes on a P/Invoke declaration. That code snippet contains all the code needed to call the Win32 `InsertMenuItem` function.

The first P/Invoke service is runtime marshaling. Different portions of the Win32 API may use different methods of representing a string, for example, and depending on the platform a string may be either ANSI¹⁰ or Unicode¹¹. The runtime marshaler knows how to convert an immutable C# string to many different string types automatically, keeping the programmer from having to worry about whether a string should contain a null-terminator or a length prefix (or both) and what character set it should use.

Though the term ‘marshaling’ commonly refers to serialization of data for either transmission or storage, Microsoft also describes the process of passing managed data into unmanaged code as marshaling. The process may involve serialization or copying, but may also simply involve converting a managed reference into an unmanaged pointer.

Not all functions take primitive types as parameters. Thus, the runtime marshaler can also handle marshaling C# structures and classes into unmanaged code. Usually marshaling native types

¹⁰ANSI format is an 8-bit character set that can only represent up to 255 characters.

¹¹Unicode is a mapping of extended characters for most known languages, and is used internally in both the .NET runtime and Windows operating systems using the NT 5 kernel or above (see <http://unicode.org/> for more information).

requires little effort on the part of the programmer. The following code is an example of a native type which can be trivially marshaled:

```
[StructLayout(LayoutKind.Sequential)]
public struct POINT
{
    public int X;
    public int Y;
}
```

The `StructLayout` attribute informs the system that the struct should be laid out sequentially in memory when passed into unmanaged code (members will be laid out in the order they appear but are not necessarily contiguous, depending on the packing rules and in-memory size of each member). This model is fully composable, and the system will continue to marshal objects contained in objects until everything is properly laid out in memory.

All of COM development revolves around interfaces, and these interfaces must be associated with their unique identifiers. `P/Invoke` then allows COM interfaces (defined as C++ classes) to be mapped to C# interfaces. The following example from `LibX` code demonstrates this association:

```
[ComImport]
[Guid("f1db8392-7331-11d0-8c99-00a0c92dbfe8")]
[InterfaceType(ComInterfaceType.InterfaceIsIUnknown)]
public interface IInputObjectSite
{
    [PreserveSig]
    int OnFocusChangeIS(
        [MarshalAs(UnmanagedType.IUnknown)] object pUnkObj,
        bool fSetFocus
    );
}
```

The `ComImportAttribute` indicates that the interface is exported by a COM object, the `GuidAttribute` provides the COM globally unique identifier for that interface, and the `InterfaceTypeAttribute` tells the marshaler that the interface should appear in unmanaged code as an `IUnknown`¹². Also, this code demonstrates the use of the `MarshalAsAttribute`, which tells the system to send the parameter `pUnkObj` to unmanaged code as an `IUnknown` COM interface. This facility is there to provide hints for the marshaler when given something generic (such as an object).

¹²The `IUnknown` pointer supports only early binding, while `IDispatch` interface pointers support late binding.

Type Libraries

The definitions, such as those given above, are distributed with the .NET Software Development Kit (SDK) as type libraries. A type library contains information about classes and methods exposed by classes contained in an unmanaged library. A type library stores in binary form the interface descriptions contained in an interface definition language (IDL) file. IDL files (and thus type libraries) specify the format of data types expected and provided and also the public methods and properties available to clients of the type described in the file.

Our biggest frustration with the P/Invoke services was, and is, that definitions like the one given above must be either produced manually or statically imported from a type library. As type libraries are designed to be used by unmanaged code, they do not include the extra information given above which informs the runtime how best to marshal calls. We found that when importing type libraries, sometimes the default marshaling behavior would result in crashes¹³.

Having to work around this problem forced us provide our own definitions (like the one above) rather than using definitions imported from type libraries. There are sites such as PInvoke.net¹⁴ that provide these definitions. However, the choice of either relying on type libraries (which might not provide definitions that marshal correctly) or relying on a definition found on the Internet is not an easy choice to make. We decided on the latter approach, because at least with the manual definitions we could tweak them by hand if we wanted.

The ability to automatically generate the interface and method definitions from the IDL files and then include that code in the project (as the xsd code generation tool does, as discussed in section 2.5) would have provided an alternative, allowing us to create patch files with our tweaks that could be applied after the code is generated. However, this option is not available as part of the type library importer tool.

Runtime-Callable Wrappers

The definitions provided through either importing a type library or directly by a developer are used to create a runtime-callable wrapper (RCW). An RCW provides managed code which wraps unmanaged objects. This wrapper code contains statically-generated marshaling code and can also call the runtime marshaler when necessary.

¹³This is documented in <http://www.ddj.com/windows/184405992> as well.

¹⁴<http://www.pinvoke.net>

Interaction with the Garbage Collector

The garbage collector is also designed to be aware of managed/unmanaged marshaling. As garbage collection should not interfere with unmanaged code, a performance enhancement to the GC algorithm keeps it from suspending threads that are executing unmanaged code while a collection is occurring. However, this optimization can cause problems when unmanaged code needs to refer to objects on the managed heap. To solve this problem, the user or the runtime layer can request that an object be ‘pinned’ in the heap, preventing its address from changing even if a collection occurs. Although pinning reduces the efficiency of compaction, it also allows pinned object references to be treated like unmanaged pointers while pinned. Objects which have been pinned must be unpinned before the memory can be freed, making pinned objects behave similarly to traditional unmanaged allocation.

Pinned objects are automatically unpinned after the call completes. If the pointer value is copied and stored in unmanaged code even after the unmanaged function returns, explicit pinning and unpinning will be necessary. Fortunately, in our experience, the majority of Win32 API calls make their own copies of objects that must be retained after the call completes (the exception to this is event handlers, marshaled as function pointers—the unmanaged code does not expect a function pointer to become invalid, so delegates must be pinned and explicit references must be kept).

2.5 Code Generation Through XML Schemas

One facility provided by the .NET framework is the ability to generate classes from XSD schemas¹⁵ and then deserialize XML directly into instances of those classes. The .NET Software Development Kit (SDK) ships with a tool called `xsd`, which will automatically generate the class definitions. This is useful for serializing objects across platforms or for storing data which needs to be user-editable and program-readable.

The tool adds attributes to classes, fields, and properties to customize serialization and deserialization behavior. Additionally, the classes are declared with the ‘partial’ keyword. A partial class in C# can be augmented later by appending the new method, property, and field declarations to the existing class declaration. By declaring the generated classes as partial, developers can add their own methods without having to modify the original code file, which will be replaced if the code is re-generated.

¹⁵An XML Schema Definition, or XSD, contains validation rules for an XML document, providing stronger semantic information than can be inferred from the structure of the document alone.

2.6 Internet Explorer Extension Model

Though the first few versions of Microsoft's Internet Explorer exposed no external API [12], versions since 4 have been programmable through a variety of means. The primary extension model for Internet Explorer is Browser Helper Objects, or BHOs [13] (though Browser Helper Object is the official term, the term exposed through the user interface is add-on, which will be used in favor of BHO in this document).

As with the rest of this paper, all the information in this section pertains only to Windows versions of Internet Explorer.

2.6.1 Hosting the Browser Control

The form on which HTML is rendered is referred to as the browser control, because it can be hosted by third-party applications as well as within IE itself. These third-party applications can use the control to load and render HTML without the Internet Explorer graphical user interface. An example of this is the Maxthon web browser¹⁶, which provides its own web browser UI around the Internet Explorer browser control. Hosting the browser control is not a feature we use in LibX, as our goal was to augment the existing browser, not provide a new one.

2.6.2 Creating and Installing a Basic Add-On

An add-on can be created in any language that can build dynamically-linked libraries in Windows portable executable (PE) format and which supports creating, exposing, and querying COM interfaces. C++ has the most support, as it is able to make use of many header files, COM interfaces, and predefined constants that must either be imported or reimplemented in other languages.

To create an add-on, it is necessary to implement a small set of COM interfaces. For a basic add-on with no UI, only a single interface (IObjectWithSite) with two methods (GetSite and SetSite) must be implemented. Internet Explorer passes its own interface pointer as a parameter to SetSite, allowing control over the browser and access to the document being viewed. All communication with IE and with the document being viewed is done through the COM interface pointer obtained through SetSite.

Once the interface methods have been implemented, the add-on can be installed. To do so, specific keys in the Windows system registry must be created so that Internet Explorer can load the add-on. First, the add-on itself must be registered as a COM class by storing its CLSID (COM class GUID) in the registry. The registry stores certain metadata (both IE-specific and general to COM

¹⁶<http://www.maxthon.com>

IE Process (iexplore.exe)

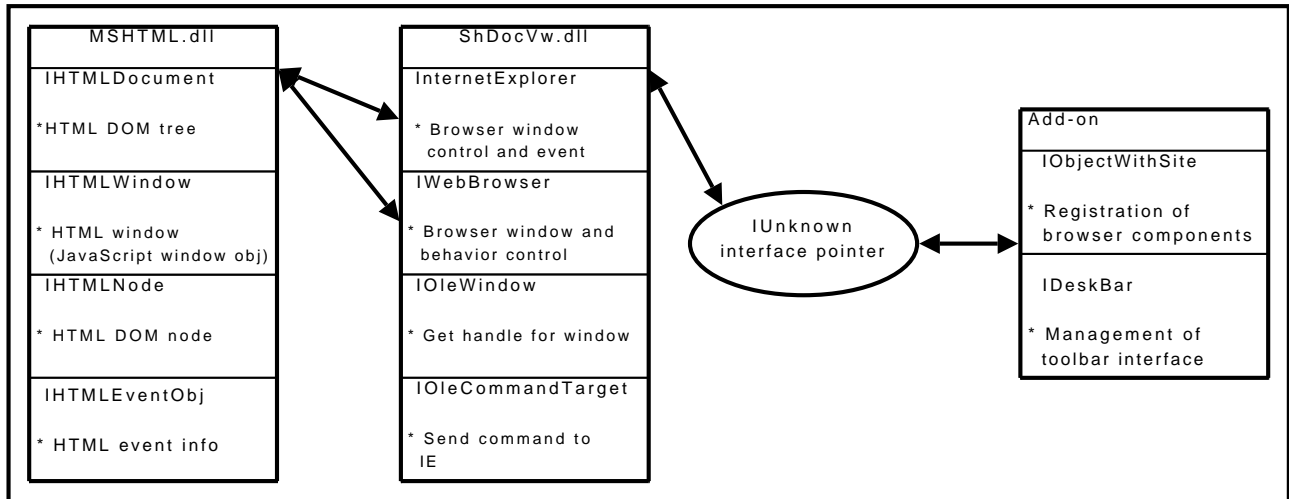


Figure 2.3: The Internet Explorer add-on model.

objects) about the add-in, including which library the class implementation is located in. Once the COM installation has been completed, Internet Explorer has its own list of registry keys at one of these three locations

HKLM\SOFTWARE\Microsoft\Internet Explorer\Explorer Bars

HKLM\SOFTWARE\Microsoft\Internet Explorer\Toolbar

HKLM\Software\Microsoft\Windows\CurrentVersion\Explorer\Browser Helper Objects

These keys enumerate the CLSIDs of the add-ons available to Internet Explorer. When a value has been added to one of these keys, Internet Explorer will automatically look up the CLSID, load the necessary library, and instantiate the class. It will then query the class for IObjectWithSite and, if successful, will call its SetSite method to establish two-way communication between the browser and the add-on.

The choice of registry location determines how IE sees the add-on.

- A browser helper object (BHO) is defined as code which implements IObjectWithSite, is loaded into IE's address space, and is given access to interface pointers that expose IE functionality such as the current document and browser settings.
- A toolbar is a specialized type of BHO that includes a user interface. It is added to the toolbar menu, is expected to implement the IDeskBand interface, and we have observed that IDeskband items are treated differently internally by Internet Explorer¹⁷.
- An Explorer Bar is similar to a toolbar, but is designed for add-ons which need to expose

¹⁷A BHO is considered 'always-on', while a toolbar can be disabled through the View menu. Additionally, different objects are passed into SetSite for BHOs and toolbars.

an interface which is larger or more complex than a toolbar is designed for. Explorer Bars can be displayed vertically as well as horizontally (the search and history bars in Internet Explorer 6 are examples of this).

If additional browser windows are invoked in the same process, the existing copy of the loaded library is used. Reusing the existing library means that all static data are shared between instances of the COM object. Therefore, information that is window-specific must be kept in instance variables, not static variables.

2.6.3 Adding a User Interface

Though the section above describes what is necessary to create a basic add-on, there are additional requirements when exposing a graphical interface to the user. Toolbars are the traditional UI component for a browser extension, though sidebars, floating or dockable toolwindows, and conventional Windows forms are all possible. In the case of a toolbar, a second interface, `IDeskBand`, must be implemented. If the toolbar requires user input via the keyboard, `IInputObject` must also be implemented¹⁸.

Another way of interacting with the user is through the context menu. Like many graphical applications, IE provides a context menu which alters its contents based on which document element is clicked and the state of the document at the time of the click.

IE provides a way to extend the context menu through registry keys, which contain pointers to JavaScript code. This code is executed when the menu item is selected. Alternatively, add-ons can also completely override the context menu, substituting their own. We found that both of these methods are unnecessarily restrictive, in that the former only provides static menu items which cannot be added to or changed at runtime, and the latter is unsuitable for developers who wish to augment the menu, rather than replace it in its entirety. There is no dynamic way to augment the existing menu that is exposed through the Internet Explorer API. For a complete discussion of how we approached and solved this issue in LibX, see section 3.3.

The final option for interacting with the user is direct manipulation of browser content. Manipulating the page being viewed by the user can be done similarly through JavaScript or by using the IE API. There are COM interfaces which mirror the HTML DOM commands used by JavaScript, and these interfaces can be used through COM in any supported language.

¹⁸There seems to be some confusion on the Internet about whether additional interfaces are required for a toolbar. The answer is that `IObjectWithSite` and `IDeskBand` are the only ones which are required, as all other proposed interfaces we have seen are actually superclasses of `IDeskBand`.

2.7 JavaScript

In addition to the C# language, a significant amount of LibX development work exists in JavaScript. Not only does the Firefox extension rely entirely on JavaScript, much of this code is shared with the Internet Explorer version as well. As with C#, focus will be given only to those features which are particularly relevant to LibX. Specifically, only client-side JavaScript as it applies to execution in a web browser environment will be addressed, and only those points about the language that stood out to us as developers will be covered.

The term JavaScript is used here to denote ECMAScript, a standardized scripting language that is implemented as JavaScript by Mozilla (formerly by Netscape and with the support of Sun) and JScript, by Microsoft.

For a comprehensive discussion of the JavaScript language, including features, language details, and implementation guidelines, see [14].

2.7.1 Execution Environment

JavaScript is generally executed on the client-side via a web browser. A script can either be executed inside a sandbox, as is the case for untrusted scripts downloaded from remote sites, or run with full privileges, which is usually the case for scripts installed alongside a browser add-on and run outside the context of a single web page. Sandboxed code does not have access to certain features, such as the ability to retrieve information from a site at a different domain than the current site.

2.7.2 Language Features

JavaScript, like C#, has first-class functions, which may be anonymous. JavaScript functions may also be nested inside other functions. Anonymous functions and nested functions are both closures and are thus able to access variables in their enclosing static scope when called outside that scope.

JavaScript also has object-oriented features, but lacks classes. Instead, it uses objects with inherited properties which may be cloned from a prototype and then modified, rather than instantiated from a class definition.

JavaScript is dynamically typed and executed, differing from static languages such as C++ and C#. The ability to modify both objects and object prototypes by adding properties at will (more fluid than C#'s partial classes, as properties can be added to either a single object or the entire class of objects derived from a prototype) allows developers to augment even built-in classes when

necessary. Also, like many scripting languages, JavaScript includes an ‘eval’ function, which will parse and interpret a string of JavaScript code at runtime.

The JavaScript execution model is single-threaded. The underlying engine is not necessarily single-threaded; however, from the perspective of the developer, there are no threads in the language and no concurrency. For blocking calls such as web requests, completion callbacks can be used to emulate thread scheduling. The ability of threading to hide long-running operations from the user can be simulated by splitting the job into chunks and setting a timer, running each subsequent chunk at a time.

2.7.3 Browser Compatibility

Although the ECMA has standardized JavaScript, the specification lags the popular implementations (Internet Explorer, Firefox) in some areas. As both JavaScript and JScript gain features, a feature added to one is not always present in the competing implementation or is not always exposed to the user in the same way. This leads to incompatibilities between the two scripting engines and, as a result, incompatibility between the ways the two browsers execute the same JavaScript.

JavaScript run as part of an add-on rather than as part of a web page is offered enhanced privileges in Firefox (through the chrome environment), while in Internet Explorer privilege is controlled based on user settings and the trust level of the current site. JavaScript running in an extension must create its own environment rather than using a privileged environment available from Internet Explorer itself. Section 3.2.2 explains how we created our own environment in Internet Explorer.

Browser compatibility problems have been frustrating web designers for years, and trying to implement a cross-browser application using JavaScript will end up frustrating application developers as well. These differences must be taken into account and addressed if JavaScript is to be used as a common language for an application targeting multiple platforms.

Chapter 3

Toolbar Implementation

This chapter details the design and implementation of the LibX IE add-on. Section 3.1 describes the design of the toolbar and its implementation. Section 3.2 explains how we integrated JavaScript and how it interacts with C#. Section 3.3 describes the design of the context menu, the steps we took to improve the configurability, and the alternative implementations methods we attempted. Section 3.4 describes how we deploy the program on client computers. Finally, section 3.5 presents our experiences and lessons learned through the design and development of LibX.

3.1 Internet Explorer Toolbar

3.1.1 Toolbar Design Requirements

The design requirements for the toolbar were already well-known at design time, as our primary goal was to reproduce the toolbar design in Firefox. The LibX Firefox toolbar is presented in figure 3.1 below:



Figure 3.1: The LibX Firefox toolbar.

User Interface

The first requirements specified for the toolbar were those pertaining to the interface: the appearance, the layout of interactive elements on the screen, and the behavior of those interactive elements from a purely graphical perspective.

- The toolbar should span the width of the browser window.
- The central text box should vary in size based on the width of the toolbar.
- The user should not be able to manually vertically resize the toolbar, as the height of the bar should depend on the height of its elements.
- The toolbar must be able to change height to accommodate additional search fields.
- When expanded, the Firefox toolbar appears as shown in figure 3.2.
- When expanded, each row can be individually closed by clicking on the red x button beside that row.



Figure 3.2: The LibX Firefox toolbar, expanded to show multiple search fields.

- Clicking a button with a downward arrow beside it opens a drop-down menu (this behavior can be seen in figure 3.3).
- Clicking the ‘Clear’ button simply clears all search fields.
- The ‘Scholar’ button does not do anything when clicked; rather, it is a drop target, allowing text to be dragged onto it and dropped.
- The search button also supports this feature, while still allowing normal clicks or the enter key to initiate a search using the text in the search fields.

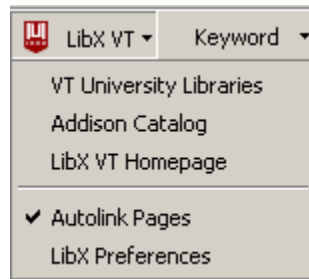


Figure 3.3: The edition links menu, shown clicked with its drop-down toolbar.

The final element in the user interface is library branding. As described in 1.3.1, LibX can be customized on a per-institution basis, creating versions of LibX specific to a certain library (these

versions are referred to as ‘editions’). For example, all the figures above are from the Virginia Tech edition, and thus show a Virginia-Tech-specific logo and have the text ‘LibX VT’ on the edition links button.

Searching

The toolbar is designed primarily around the search feature. We wanted the toolbar to aggregate all the catalogs offered by the library in a single place, where each one could be queried on any (and any combination) of the fields it supports.

Search support extends to Google Scholar. Text can be dragged onto the Scholar button from HTML documents, but text may also come from Adobe PDF documents, for example, opened in the web browser through the Adobe Acrobat Reader. Our Scholar search enhancements use a heuristic approach to determine whether Scholar found the item searched for. If so, and if an OpenURL is provided by Scholar for that resource, that link is followed. If the heuristic indicates that no result is likely to be a match, the scholar results page is displayed. Displaying the OpenURL link page on a probable hit provides fast access to known resources, such as citations in a paper or journal [6].

Navigating

When a search is performed, the search results must be presented to the user. In Firefox LibX, the system can be configured to open results in the current tab, a new tab which is focused (the active tab), a new tab which is unfocused, or a new window. A new, focused tab is the default. This setting also pertains to links selected from the edition links menu.

Since Internet Explorer 6, one of our target platforms, does not support tabbed browsing, LibX installed on IE 6 should allow the user to choose only between reusing the current window or opening a new window, with the latter being the default.

3.1.2 Toolbar Implementation

Articles written by Pavel Zolnikov [34] and Michele Bustamante [10] provided a starting point for our efforts. Since we found little documented support provided by Microsoft’s base class library or in the Microsoft online knowledge base, these resources were very helpful.

Registering the Toolbar

Making Internet Explorer recognize and load our add-on was the first step. The process of registering as an add-on is described in 2.6.2. In brief, the stages are:

- Install the assembly into the Global Assembly Cache (GAC installation is discussed in 3.4).
- Register the toolbar as a COM class.
- Create registry keys to load the toolbar into Internet Explorer.
- Implement the `IObjectWithSite`, `IInputObject` and `IDeskBand` interfaces.

The interfaces listed as the last stage allowed us to customize the interaction between our toolbar and Internet Explorer. A basic implementation of `IObjectWithSite` includes a `SetSite` method that stores the `IUnknown` interface pointer passed in by the browser and a `GetSite` method which returns that pointer. `IInputObject` is required because the toolbar has input controls which use the keyboard, and is discussed later in this section. `IDeskBand` is discussed further below when describing the user interface integration.

Integrating the Interface with Internet Explorer

The key component to implementing the behavior of Internet Explorer toolbars is the `IDeskBand` interface. The following methods were necessary for us to implement:

- **ShowDW**: Called when the toolbar is shown.
- **CloseDW**: Called when the toolbar is closed.
- **GetWindow**: Returns the Win32 Window Handle (hWnd) for the toolbar window.
- **ResizeBorderDW**: Called when the border is resized. Allows the toolbar to modify the proposed new size.
- **GetBandInfo**: Called when IE needs data about the band such as its title, minimum size, and position preference.

The last method, `GetBandInfo`, is called by Internet Explorer when the toolbar layout system believes it is necessary to get information. However, if a toolbar developer wishes to preemptively change a value controlled by Internet Explorer (such as the size of the toolbar), it is necessary to inform Internet Explorer that its cached data are stale, triggering the needed call to `GetBandInfo`.

In order to inform IE that a call to `GetBandInfo` is needed, a command must be sent. This command is sent through the `IOleCommandTarget` interface. This interface exposes two methods:

- **Exec**: Executes a specified command. Discussed further below.

- QueryStatus: Queries the status of a command invoked through Exec.

In brief, Exec categorizes commands as belonging to one of several command groups. When calling Exec, the developer first specifies a command group, then the numeric ID of a command within that group. Additional command options, the parameters to the command, and a pointer which should receive the result of the command (if any) are also passed in.

To do this, it is necessary to retrieve the IOleCommandTarget interface from the IUnknown pointer given by IE in the SetSite call. Then calling the Exec method with the correct parameters¹ results in a call to the GetBandInfo method from Internet Explorer. The following code example demonstrates this:

Getting the IOleCommandTarget interface:

```
public virtual void SetSite(object pUnkSite //IUnknown from IE)
{
    Guid iocT = IID_IOleCommandTarget; //Interface ID for IOleCommandTarget

    //Get the interface pointer
    IntPtr ifPtr = Marshal.GetComInterfaceForObject(
        pUnkSite, typeof(IInputObjectSite));
    IntPtr outPtr = IntPtr.Zero;
    //Query the interface for IOleCommandTarget
    Marshal.QueryInterface(ifPtr, ref iocT, out outPtr);
    Marshal.Release(ifPtr);

    //Get IOleCommandTarget object
    _cmd = Marshal.GetUniqueObjectForIUnknown(outPtr) as IOleCommandTarget;
    if (_cmd == null) {
        throw new Exception("Could not get object for command target");
    }
}
```

¹The parameters are documented by Microsoft at <http://msdn2.microsoft.com/en-us/library/ms690300.aspx>

Calling the method:

```
void ForceGetBandInfo()
{
    Guid dbGuid = IID_DeskBand; //Interface ID for IDeskBand
    object outParm = null;
    object inParm = (object)_id; //The ID for this toolbar

    //CMDID is the ID of the command to execute in this command group
    //OLECMDEXECHOPT_DONTPROMPTUSER is a Win32 constant
    _cmd.Exec(ref dbGuid,
              CMDID,
              OLECMDEXECHOPT_DONTPROMPTUSER,
              ref inParm,
              ref outParm);
}
```

Handling the Keyboard

If the toolbar has input controls, such as a textbox, it is necessary to implement the `IInputObject` interface. This interface exposes the following methods:

- `UIActivateIO`: Handles navigating between controls.
- `HasFocusIO`: Queries whether the toolbar currently has input focus.
- `TranslateAcceleratorIO`: Handles special Internet Explorer shortcut keys.

The important method to implement is the `TranslateAcceleratorIO` method. Internet Explorer receives keypresses before toolbars do, meaning that keys captured by IE will not be sent to the toolbar. It queries the toolbar (using `HasFocusIO`) to determine if input focus is on the toolbar and, if so, calls `TranslateAcceleratorIO`, passing in the Win32 message structure containing the key that was pressed.

If this is not done, users who press the backspace key in the edit control will find Internet Explorer navigating one page back in the history, which is the default behavior of the backspace key. The arrow keys will also not work for moving the cursor around in an edit control. Implementing `TranslateAcceleratorIO` allows a toolbar to indicate to Internet Explorer that it has handled a keypress, preventing IE from acting on the keypress as it normally would.

Developing the User Interface Layout

As mentioned in section 2.3.5, the Visual Studio Form Designer is the suggested method of laying out visual elements for the `.NET System.Windows.Forms` namespace (referred to as WinForms). However, the LibX interface is very dynamic, making the Form Designer unsuitable for our purposes. We opted instead to create the GUI code ourselves.

In doing so, we wanted to mimic the relative layout of the Mozilla XUL engine rather than the absolute layout of the Form Designer. To do so, we took advantage of the `TableLayoutPanel` and `FlowLayoutPanel` classes introduced in `.NET 2.0`.

We used a `TableLayoutPanel` to provide the overall structure to the toolbar. We split the rows into three cells: fixed-size cells on the left and right, and a dynamically-sized cell in the center. Each cell contains a `FlowLayoutPanel` which automatically positions the elements in that cell. The center cell then contains all the elements which are cloned when a new row is created. Figure 3.4 shows the grid lines of the table used to position the elements.



Figure 3.4: The LibX IE toolbar with grid lines showing table cells.

Creating Drop-Down Buttons

Buttons which expose a drop-down menu when clicked are seen in both IE and Firefox. However, there is no `.NET WinForms` control which exposes this functionality to programmers. Buttons with this functionality are used extensively in the LibX UI, so we chose to create this functionality ourselves.

Our approach was to create a subclass of the `Button` control which handles its own click event to display an associated menu. The control also overrides its own paint event to draw a down arrow (like the one shown on the button in figure 3.3), providing an indication to the user that this button offers special functionality. The rest of the behavior of the `Button` class is preserved.

One problem we had to solve was that, in the Firefox code, selecting a menu item frequently causes the text of the associated button to change. In the LibX XUL code each button has its own menu, making changing button text based on menu selection trivial. However, in the IE C# code, multiple drop-down buttons can use the same menu object, avoiding duplication of resources.

To solve the above problem, we made use of the `Tag` property available on each control. `Tag` is

an object property which can contain any data the developer wants to associate with the control. When a button is clicked, the associated menu's tag is set equal to the button which was clicked. As the menu cannot be invoked without clicking a button, assigning the Tag property allows us to programmatically determine which button invoked the menu.

3.2 JavaScript Integration

As presented in section 1.4.1, a goal for our implementation was to facilitate the reuse of the existing JavaScript code. We briefly discussed the option of writing a XUL rendering engine for IE and using nearly the full extent of the Firefox code, but we rejected this as being too complex and not solving the problem of other differences in functionality, such as chrome: URLs. But even though we recognized that we would be unable to use all of the Firefox code, we wanted to keep the core LibX logic in JavaScript, refactoring² as necessary to support both browsers.

This section describes our requirements for a JavaScript environment³ in 3.2.1, followed by our solution in 3.2.2. 3.2.3 contains a discussion of how JavaScript interaction operates. Finally, 3.2.4 details the refactoring work we found necessary.

3.2.1 JavaScript Requirements

To create an environment which mimics the Firefox environment as accurately as possible, we wanted:

- access to properties of the current browser frame, such as 'window' and 'document'.
- security permissions high enough to perform actions, such as open a popup window or reference a local file, which scripts running under restricted privileges cannot do.
- the ability to insert C# objects in the script which we could use for communication and extended functionality.

3.2.2 Getting an Execution Environment

Initially, we considered injecting the LibX JavaScript into the current page's JavaScript environment. Even though this approach would not have allowed execution with full trust, access to the

²Refactoring is the software engineering practice of changing the appearance or implementation of code without changing the task that is performed from the perspective of the calling code.

³By JavaScript environment, we mean an execution context, including a global scope, a set of defined global variables, and security privileges.

website objects is the most important goal. However, the JavaScript engine IE uses is not exposed through the API, and is effectively completely encapsulated.

We could obtain indirect access to the environment by inserting the JavaScript into the current page by programmatically adding a script tag. However, this would cause it to execute under the security context of the current page, which by default is too restrictive, and may be disabled altogether if the user has disabled JavaScript. In addition, if a script on the page encountered an error, it would prevent the LibX code from running.

Instead, we opted to simply instantiate our own environment. By creating a new JScript engine and using the IActiveScript COM interfaces ⁴, we were able to access those properties and set up an environment of our own. We did search for managed equivalents to the COM JScript/ActiveScript components, but nothing we found would give us the same flexibility that the COM objects did.

However, this new environment is in a different global scope and contains none of the global properties of the environment interpreting the script on the current web page. Instead, our solution uses the functionality exposed through the Active Scripting library to insert the proper items (window, document) into the global scope of the new environment. As figure 3.5 shows, we do this by inserting C#-referenced COM objects into the script.

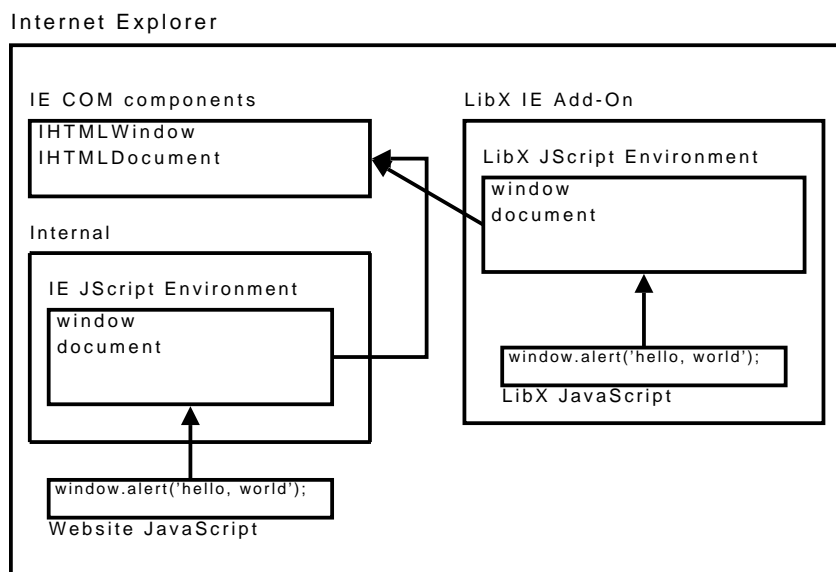


Figure 3.5: Priming the JavaScript environment with the correct variables.

These objects expose the same functionality as the window and document properties seen by web scripts running inside the IE environment. Changes made by web scripts are reflected in the COM objects and then become visible, without any extra interaction, to LibX code.

⁴The IActiveScript COM interfaces expose functionality for interacting with the Active Scripting Engine, which executes languages such as JScript and handles interoperation between these languages and host code.

The mechanism we used to insert these COM objects is the `AddNamedItem` method of the `IActiveScript` interface. This method allows an item to be inserted into the script and (optionally) allows the inserted item's properties to become global properties as well. As the 'window' object is the global object for web pages (allowing 'window.alert' to be written as 'this.alert' or simply 'alert'), allowing the methods and properties of the inserted object to be attached to the global object mimics this behavior, increasing compatibility with code referring to these properties as members of the global object.

3.2.3 JavaScript/C# Interoperation Fundamentals

Calling `AddNamedItem` also allows us to insert C# managed objects into the script. Doing so facilitates communication between the script and the C# script host, as the script can call methods on inserted objects to communicate with the script host or the toolbar itself.

Many primitive types marshal seamlessly between JavaScript and C#. As an example, a string passed into C# from JavaScript can be treated in C# as a managed `System.String`. This simplifies interaction between JavaScript and C#.

Most JavaScript objects, however, are marshaled as opaque COM objects. Interaction with these objects from C# requires the use of reflection⁵. To provide an example of using reflection to dynamically retrieve a property value from an object passed in from JavaScript, the following code snippet is taken from the LibX C# code that provides access to search catalogs:

```
Type t = catalogObject.GetType();
string opts = null;

try {
    PropertyInfo pi = t.GetProperty("options", typeof(string));
    opts = pi.GetValue(catObject, null) as string;
}
```

This code queries the 'options' property from the catalog object to retrieve its value. This same code could be used without modification if `catalogObject` were a C# class with an 'options' property of type `string`.

JavaScript functions, too, can be passed in to C# and then executed easily. They are not converted into C# delegates⁶, but instead are exposed as COM objects which support the `IDispatch` interface.

⁵Reflection is a runtime feature which allows code to dynamically inspect itself, analyze, and modify its own structure.

⁶Although it would be possible to implement JavaScript functions as delegates with a variable number of generic object parameters and an object return type, the marshaling is designed to work with any COM-supporting language, not just C#.

The code to invoke a method is as follows:

```
Type t = typeof(stdole.IDispatch);  
t.InvokeMember("",  
    BindingFlags.InvokeMethod,  
    null,  
    command,  
    null);
```

‘command’ is the function to invoke, passed in from JavaScript. Particularly relevant parameters are the first, which is the method to invoke (the null string implies that the runtime should invoke the default member of the command object, which for a function is the method itself), and the last, which is an object array representing the parameters to the function. InvokeMember returns an object, which is the return value of the function.

Despite the amount of interoperation we took advantage of, we did not exhaust all the possibilities offered by this model. For instance, it is possible to sink managed events into unmanaged code, and invoke them in response to a JavaScript event.

3.2.4 JavaScript Refactoring

Design for Refactored Code

Our design is similar to that of traditional cross-platform applications. The majority of the code is contained in JavaScript files that can run on any browser. When these files need functionality that is not exposed consistently across browser platforms, it calls into a compatibility layer specific to the platform it is running on. The browser can interact both directly with common code or through the compatibility layer, as necessary. Figure 3.6 details this process.

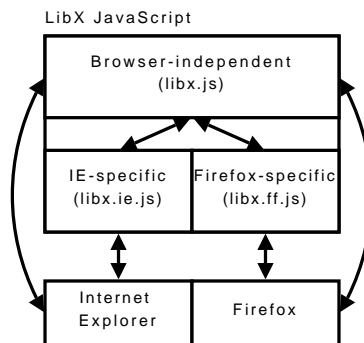


Figure 3.6: Refactored LibX JavaScript layers

Targets for Refactoring

JavaScript using Firefox XUL, the Mozilla UI rendering language for Firefox, required the most refactoring. The original LibX version used XUL to directly store program state. In the absence of XUL, errors and unexpected behavior occur when the JavaScript is looking for document elements that are not present or using a protocol (chrome:) that is not supported by Internet Explorer.

Another area of refactoring was in the interaction between the script and the browser. For example, opening a new window triggered third-party popup blockers when executed through JavaScript in IE. To remedy this, the code to open a new window was exported to C# for IE and called from the compatibility layer.

Differing implementations of JavaScript features also provide an area of incompatibility. An example of this is the XMLHttpRequest object, used in the technology known as AJAX [27]. Microsoft added the functionality to JScript as an ActiveX object (created through `new ActiveXObject("Microsoft.XMLHTTP")`), but as JavaScript does not support ActiveX, the XMLHttpRequest was added as a property of the window object. This difference in implementation meant that there were two differing ways of instantiating this object, requiring code to attempt one, possibly fail, and then attempt the other. Although Internet Explorer 7 has added XMLHttpRequest as a property of the window object, websites which wish to be compatible with Internet Explorer 6 and below still have to have browser checks in their code if they wish to use this feature.

Additionally, different browsers offer different language constructs. Firefox, to give an example, has a `const` keyword, while Internet Explorer does not. Some of the original LibX code used the `const` keyword to provide extra robustness and semantic information when declaring variables that were not expected to change. Variables declared as `const` had to be changed for compatibility with Internet Explorer.

The Refactoring Process

Since architectural refactoring was not the focus of our work, we hoped to minimize the number of sweeping changes made to the code. To accomplish this, we moved all code that needed to be shared but was not compatible with Internet Explorer into Firefox-specific files, and re-implemented the necessary functionality in the IE-specific files. If the majority of a function or object was not browser-specific but at one point it used a Firefox-only feature, that feature would be implemented in the IE layer rather than moving the entire code block into the Firefox layer and reimplementing it all for IE.

Not all Firefox-specific code needed to be reimplemented at all for IE, as code related to creating

and updating the user interface is handled solely in C#. In those cases, the IE layer contains empty stubs for the functions it does not need.

As more features were added to the IE version, incompatible areas of JavaScript which had not been targeted in the initial compatibility efforts were refactored as needed.

3.3 Context Menu

The context menu code was completely refactored both for Firefox and IE. A context menu is a menu that is invoked as a result of either a click of the secondary mouse button or pressing the menu key and usually contains a dynamic list of items which changes based on the context of the click. The LibX context menu was improved to be more context-sensitive and also more user-configurable. The changes also allowed browser-specific context menu operations to be abstracted to the compatibility layer while retaining most of the logic in shared code.

As there are several catalogs, each offering several search options, LibX uses the context menu to narrow down the available queries based both on context and user preference. The context menu is also the primary way through which proxy functionality is exposed to the user.

In this section, the design requirements for the context menu are presented in 3.3.1. We then discuss the documented methods we found for extending the context menu in 3.3.2. 3.3.3 describes the solution we chose to implement. In 3.3.4 we present how we implemented additional graphical interfaces around the context menu.

3.3.1 Design of the LibX Context Menu

The context menu for LibX is designed to allow easy access to the most-used catalogs and search types, as well as to the proxy feature. A hierarchical system is used to determine what the context is and thus which items to show. This system is completely user-configurable, and the configuration persists across sessions through serialization⁷.

The menu system is designed to determine proxy context based on the location of the click. If the clicked item is a link, the context menu offers the ability to follow the link via the configured proxy. Otherwise, the user can reload the current page using the configured proxy. An enhancement, supported by some proxy software (currently only EZProxy⁸), allows a dynamic query to be sent to the proxy server to determine if the current page can be proxied or not. If this feature is supported by the configured proxy and enabled, the context menu reports the status of the query:

⁷Serialization is the process of converting an in-memory representation of an object into a format which can be either transmitted across application and machine boundaries or saved for later loading.

⁸<http://www.usefulutilities.com/>

if the page cannot be proxied, the user is informed of this. If the page can be proxied, the option is given to do so.

The above dynamic querying implied that our design needed to handle changes to the text of menu items while the menu was being displayed. As a remote request might take a perceptible amount of time to complete, we initially display an indication that the proxy is being queried. Once the query returns, the text is dynamically changed to reflect the results of the query. This imposed additional requirements on our implementation of this feature.

Context for searching is determined based on what text is highlighted. If no text is highlighted, no search options are given. If text is highlighted, the system checks to see if the highlighted text is one of several types of tokens, such as PubMed IDs or Digital Object Identifiers (DOIs). The catalogs offered to the user for searching vary based on what type of token is identified. If no token is identified, the fallback (general text) catalog options are presented.

Section 3.3.4 discusses how the catalogs are configured and expands on the hierarchical context determination. Figure 3.7 demonstrates a context menu with no text selected and one with the phrase ‘OpenURL resolver’ selected.

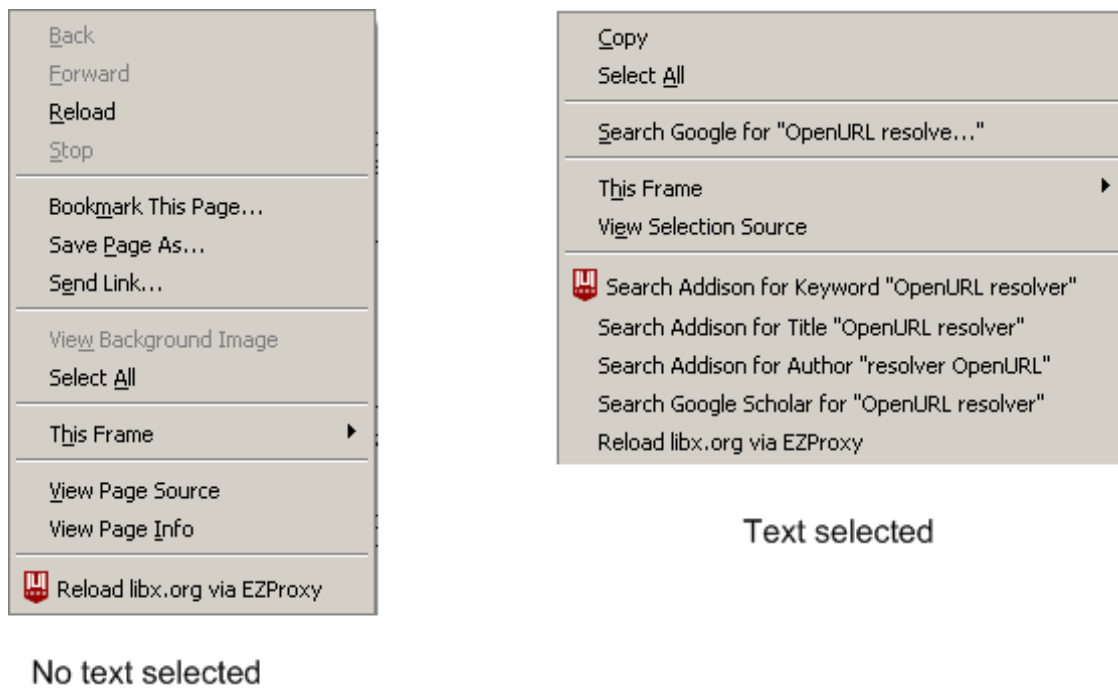


Figure 3.7: Examples of contextual menu presentation.

3.3.2 Officially-Supported Context Menu Extension Methods

As discussed briefly earlier in section 2.6.3, Internet Explorer provides two official, documented mechanisms for using the context menu. These mechanisms are:

- Registering menu extensions in the Windows system registry ahead of time
- Completely replacing the IE context menu with a new one

Static Registry Method

The static registry method involves the creation of registry keys which associate scripts with contexts, allowing menu items to be inserted and optionally displayed based on a combination of certain flags. The registry location in which scripts are registered is:

```
HKCU\Software\Microsoft\Internet Explorer\MenuExt
```

The default value for this key is an HTML file, specified as a URL (usually using the file: protocol, but the HTML can also be a resource compiled into a library using the res: protocol).

The key can also have a value named 'Contexts', which specifies context flags to determine when the menu item should be shown. Examples of context include showing the item only if the mouse has been clicked on an image, or if text has been highlighted on the page.

The file specified in the value of the key is loaded in a new, hidden window. The author of the menu extension is expected to use JavaScript inside the HTML file to perform the action. Once the page has loaded and the script has been processed, the window is disposed.

We rejected this method because it did not allow us to customize either the contents of the menu or the menu text at run-time. While this method is lighter-weight than creating a full add-on, it was neither powerful nor dynamic enough for our needs. The LibX menu is truly context-sensitive in that it customizes both which menu items are displayed and what those menu items actually display at runtime, and this method did not allow us to replicate this in Internet Explorer.

IDocHostUIHandler Method

This method expects the developer to completely replace the context menu with a custom one, gaining complete control over which menu options are displayed; however, the normal Internet Explorer menu items are then completely overridden and thus not displayed. This sort of control is provided for developers who are hosting the Internet Explorer browser control in their own applications and wish to replace the context menu with their own (or remove the context menu

entirely).

In order to use this method, it is necessary to have a class which implements one of the COM `IDocHostUIHandler` or `IDocHostUIHandler2` interfaces. These interfaces expose the `ShowContextMenu` method. `ShowContextMenu` must do everything itself: create the context menu, add items to it, display the items, and handle any resulting clicks on the menu.

The first and largest problem with this method, of course, is that there can only be one `IDocHostUIHandler`. If an add-on adds itself as the handler and then another add-on does the same, the winner will be whichever called `SetUIHandler` last. This is why the UI handler method is designed to be used only by programs hosting the browser from their own application.

Historically, some programs have used this method to augment (rather than replace) the standard IE menu by loading IE's resources and re-creating the context menu that IE would have displayed. However, if internal resources are refactored and relocated, as they were between IE 6 and IE 7, the new version will break the toolbar and then the toolbar must detect which version it is running on and locate the appropriate resources. This is no longer a supported method of extending the IE context menu⁹, and still does not address the issue that only one add-on can extend the menu in this way at a time.

We attempted to use the UI handler method, even producing a proof-of-concept. However, we couldn't develop a robust way to incorporate the original context menu entries, and we could not find a solution to the problem of competition between add-ons. Ultimately, neither of the approved methods allowed us to support the design requirements as stated in section 3.3.1. The solution we chose is discussed in the next section.

3.3.3 The Subclassed Window Procedure Method

As it did not appear that Internet Explorer offered any further options for extension, we looked lower, in the Win32 API. The Win32 API is the collection of functions and structures that can be used to program Windows applications and develop native-looking GUI elements.

Windows Messaging Basics

Win32 works by using a message-passing system. Each Win32 program executes a *message loop*, which receives and dispatches messages targeted to its particular handle. Windows and controls then register message handlers, which are functions with a set signature that perform some action

⁹'No longer' because the method for doing this is actually suggested as a possible solution on the MSDN documentation site at <http://msdn2.microsoft.com/en-us/library/aa770042.aspx>, thus misleading developers into believing that this is an official way to extend the context menu.

based on the message received.

As Win32 works using messages, we posited that we could hook into the message loop of the browser window to inspect all messages received, get a handle to a Win32 menu resource (HMENU) from the menu creation message, and then insert our menu items that way.

Window Procedure Basics

Associated with each window is a window procedure, which is invoked by the message loop when messages for that window arrive. The procedure can then handle the messages directly, dispatch them to child windows, or ignore them. Windows are given classes, and all windows of the same class use the same window procedure. As part of the Win32 API, there exists a function to alter the state of a window, including externally *subclassing* the window by creating a new window procedure for the window and overwriting the old one with the new.

Subclassing a window allows the new window procedure to interpose itself between the messaging system and the original window procedure. Messages may be passed on to the original window procedure without modification, altered and passed on, or not passed on at all. In our approach, we wanted to pass on all messages not related to the context menu. Context-menu-related messages we would inspect ourselves, then pass on to the old window procedure only if they were not handled by our own code.

Handling Menu-Related Messages

Windows includes several messages that are fired to indicate menu-related events. The messages we found useful are:

- Menu Pre-Initialization

The WM_CONTEXTMENU message informs the application that a menu will be displayed at a certain point. This message is important because its parameters indicate the X and Y coordinates of the menu being created.

- Menu Initialization

WM_INITMENUPOPUP provides the menu handle for the menu that is preparing to be displayed. The menu handle is necessary for adding items to the menu. It is here that we can add our LibX menu items.

- Menu Teardown

When we see the WM_UNINITMENUPOPUP message, we know we can remove our items from the menu. Items are inserted and removed each time the menu is shown and hidden,

respectively, because this is how the underlying JavaScript code behaves (this behavior is discussed further in section 3.3.4).

- **Menu Hover Select**

The `WM_MENUSELECT` message is sent when the user hovers the mouse over a menu item. This method can be used for providing tooltips or help text in the status bar.

- **Menu Item Clicked**

The `WM_COMMAND` and `WM_MENUCOMMAND` messages both indicate that a menu item has been chosen from a popup menu. These messages are not guaranteed to be sent when a menu item is chosen, and in fact Internet Explorer 6 and 7 do not send these messages.

Getting the User's Menu Clicks

Whether a message is sent when a menu item is chosen depends on the flags passed to the API call `TrackPopupMenu`. `TrackPopupMenu` displays the menu and enters a local message loop. When it receives either a mouse click or a keyboard event, it determines whether a menu item was chosen or not. If so, depending on its flags, it will either send a `WM_COMMAND` message, a `WM_MENUCOMMAND` message, or will return the ID of the chosen item as its return value, sending no message at all.

While anecdotal evidence suggests that Internet Explorer versions prior to IE 6 sent `WM_COMMAND` notifications when menu items were selected, IE versions 6 and 7 do not. Rather, they instruct `TrackPopupMenu` to return the value of the clicked menu item, where it can be handled locally.

Thus, in order to receive notification that the user has selected an item, we had to hook into the mouse handler and detect the clicks ourselves. This required another set of Windows API calls, centered around `SetWindowsHook`. `SetWindowsHook` sets 'low-level hooks' for events generated by the keyboard and mouse. When the menu is displaying we set a hook, and once the menu has been hidden then we remove it.

The hook allows us to see X and Y coordinates for mouse clicks while the menu is displaying. When the user clicks the primary button, the coordinates are then checked to see if they are in the hit rectangle of any LibX menu item (as provided by the `MenuItemFromPoint` and `GetMenuItemRect` API functions). If so, an event is triggered for the C# menu object itself.

Implications

When we insert and handle our menu items this way, we violate IE's encapsulation by relying on the way that the Win32 API works when handling menus, bypassing Internet Explorer's API

entirely. However, as no suitable dynamic menu extension mechanism exists, we had to decide whether to take this approach or abandon the use of the context menu.

3.3.4 The Context Menu Preferences

In order to put the flexibility of our context menu system into the user's hands, we implemented a graphical interface around its configuration.

Configuration Design

As mentioned above, the context menu system is driven by hierarchical settings. A setting is either always on, in which case it will always be displayed in the context menu, or its category falls somewhere in the hierarchy. Categories sooner in the hierarchy take precedence over categories later on.

The category is determined by the text that is selected. The category of the selected text is determined by walking the hierarchy. The current hierarchy is:

ISBN > ISSN > PMID > DOI > General Text

If text is selected, first it is tested to see if it is a valid ISBN. If so, the ISBN-related menu items are displayed. If not, it is then tested to see if it is an ISSN. This is continued through the hierarchy until it reaches the end. The last item displays general fallback options.

If no text has been selected, the only options which are displayed are the ones which are always on. Currently, the only menu items which are always displayed are the proxy-related menu items. This is because the proxy functionality is independent of what text is selected.

User Interface Design

To allow users to configure the behavior for each item in the hierarchy, we designed a user interface. The setting we are allowing users to change is, for each category in the hierarchy, which menu items should be displayed. Our goal is to embody enough of the design information discussed above into the interface that the user understands intuitively what the significance of the settings is and, when the settings are changed, what impact those changes will have on the behavior of the context menu.

We decided that we would present this information to the user as a set of tabs. Each tab would contain a tree view, and each leaf node in the tree would be a possible context menu item. These items would be user selectable. The expandable and collapsible branches in the tree view would

allow users to focus in on the precise items they are interested in. A sample of this interface can be seen in figure 3.8.

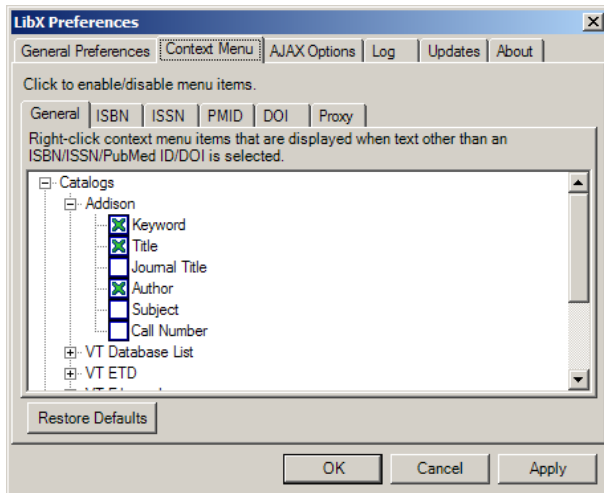


Figure 3.8: The user interface for context menu preferences.

Configuration Implementation

In order to implement the configuration handling, we developed a hierarchy of objects to represent individual context menu entries and entire context menu categories.

At the bottom of this hierarchy is a browser-specific representation of one text label in the context menu—we refer to this as a menu item. In Firefox, this is a XUL node for the context menu, and in IE this is a C# object managed by the LibX IE add-on. This menu item is expected to have certain methods and properties (for instance, methods for setting the text to be displayed by the menu item, or for graying the menu item out).

Containing each menu item is a browser-independent wrapper which we call a menu entry. This menu entry encapsulates the menu item and provides extra data about it used by the context menu system. These extra data are the internal name of the menu entry and the category to which the entry belongs (such as ISBN or DOI).

Holding multiple menu entries is a menu object. This menu object represents a group of menu entries which are related by their category. These menu objects, which are stored in the order of the hierarchy of categories given above, enforce the hierarchical nature of the processing: if the menu object containing ISBN entries matches the selected text as an ISBN, the menu object containing DOI entries is never even iterated over.

Finally, applied to one or more menu objects is a label. This label determines the class of the object. Objects which are in the always-on class are exempt from hierarchical processing, as the

will always be displayed.

There is a large number of potential context menu items. Every search type in every catalog or OpenURL resolver in every category in the hierarchy is a potential menu item. Displaying each of these menu items, even when trimmed based on category, would be unwieldy. Instead, we allow the edition maintainer to pre-select which menu items should be displayed, and then allow the user to change these preferences dynamically.

The context menu preferences object contains a list of selected context menu items. Only selected nodes are included in the hierarchical process. This preferences object is serialized through reflection by recursively enumerating the properties of the base context menu objects and writing them to disk in XML format. The next time the toolbar is loaded, this serialized object is persisted back from disk and its properties are re-populated from the XML.

When the menu is shown, the hierarchy of context menu objects is traversed as described above. Once the correct category of objects is determined, only the objects the user has selected are loaded into the context menu.

After the menu is finished displaying, the menu items are removed from the menu. This allows the menu items which should be shown to change in between menu invocations without requiring the user to restart the browser.

User Interface Implementation

To allow the user to choose which context menu items to display, we implemented the user interface described above and shown in figure 3.8.

We took an approach similar to that of the context menu when implementing the interface. The tabs, tree views, and tree items are all created by browser-specific code and stored in native objects which expose a consistent interface. Much like the context menu items, the user interface consists of an object hierarchy.

A tab contains a single tree view, along with a text label offering an explanation of that tab's functionality.

A tree view contains one or more tree nodes. As the tree view GUI element is different in C# and XUL, a tree view is represented in the JavaScript as a virtual (not visible) root node.

A tree node can contain one or more child tree nodes. If a tree node has no children, it displays a checkbox. If it has children, the checkbox is not displayed, but instead the node can be expanded or collapsed.

This implementation allows browser-independent JavaScript to create the tabs, tree views, and tree nodes in the preferences UI dynamically. As a result, the context menu preferences are as dynamic as the context menu itself, and present the preferences to the user in a manner that mimics the way they are computed internally.

3.4 Deployment and Updates

To install the toolbar onto client computers, we chose the Nullsoft Scriptable Install System (NSIS)¹⁰. Factors that led to this decision were that it is freely available on the Internet, it is an actively-maintained open-source project, and it can build Windows installers on both the Windows and Linux platforms. The last point integrated well with our planned build and distribution system.

NSIS creates graphical wizard which walks the user through the process of installing software. This installation and interface can be customized using a domain-specific scripting language. The install script is used both at compile time to build the installer package and run time to deploy the files to the specified directories and make the necessary system configuration changes.

3.4.1 Initial Deployment

We designed the installer to just contain LibX specific files. The LibX dependencies were placed in a well-known location on the LibX web server and can be downloaded automatically by the installer if they are not present. This optimization was performed both to save space on the server (as each edition builds its own installer, if each installer contains the dependencies, space is wasted on redundant information) and to save user bandwidth if the dependencies are already installed.

The installer then copies the LibX files to the appropriate directories and makes the necessary registry changes (discussed in 2.6.2) to install the toolbar.

The one dependency which is not handled by the installer is the .NET Framework. The reason for this is that the .NET Framework is localized into all the different languages supported by Microsoft Windows and also distributed in different versions (x86, x64, and 64-bit Itanium) depending on the target system's architecture. Determining where on these two axes the target system falls is non-trivial, so given that the .NET framework can be installed through Microsoft's Windows Update¹¹, we could direct users there if the framework is not installed.

¹⁰<http://nsis.sourceforge.net>

¹¹<http://update.microsoft.com>

3.4.2 Automatic Updates

The Firefox version of LibX, like all Firefox extensions, uses a built-in automatic update feature. This feature allows either manually-triggered or automatic, periodic checks for updates. We wanted this feature to be present in the IE version as well to prevent users from having to visit the web site, download the new installer, and execute it manually.

The mechanism we use is designed to mirror the Firefox system in behavior, though the user interface is different. The version on the server is read and compared to the version on the client; if the server version is greater, an update is performed. Due to the optimization of dynamically downloading only the necessary dependencies, updates can use the same installer as the initial deployment, but only the updated components need to be downloaded.

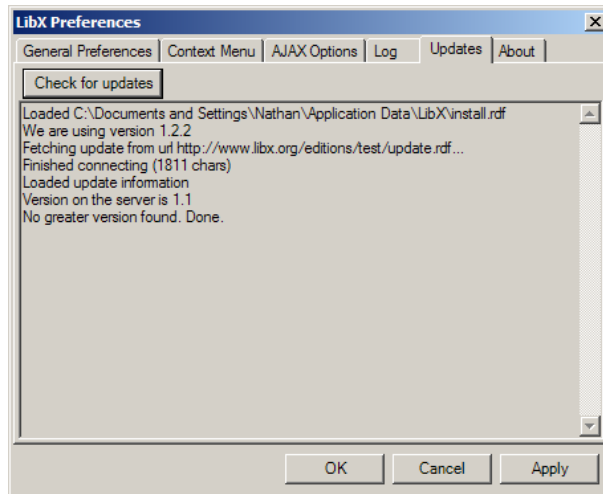


Figure 3.9: The user interface for updating.

Update checks can be triggered manually or periodically through the automatic system. The manual mechanism is exposed through a user interface, as shown in figure 3.9. Periodic checks are also performed once every 24 hours. The last check time is persisted to disk, to ensure that users whose workflow includes keeping the browser open for less than 24 hours will still see periodic updates.

The updates also contain a randomized component. We were concerned that certain events (like the start of a workday, when many people in a large organization might open the web browser within a short period of time) could cause uneven load on the update server. To help distribute updates evenly in time, we each scheduled update time by a random component. This adjustment is not enough to delay updates by more than 24 hours.

3.4.3 GAC Registration and COM Installation

The .NET SDK contains two tools for deploying .NET programs: `gacutil`, which installs and removes assemblies from the global assembly cache, and `regasm`, which registers assemblies as COM objects.

Unfortunately, those tools are not present in an install of the .NET framework¹². However, the functionality provided by both of these tools is available programmatically through the .NET framework itself. Thus, we were able to create our own tool which performs the same task as these two tools. We can then distribute this tool with our installer.

3.5 Implementation Experiences

3.5.1 Differences in Browser Extension Philosophies

Internet Explorer uses a substantially different model for developing add-ons from that of Firefox. The terminology itself evidences this difference: in Firefox, the developer is truly extending the browser, even at the core level. In Internet Explorer, this is not possible. Instead, developers are adding-on extra functionality that is completely separate from the browser itself, only communicating through known interfaces.

We also found that, for Internet Explorer, we spent more time researching and implementing meta functionality. Tasks such as adding the toolbar to the registry or providing implementations for COM functions that are only used to allow IE to manage the add-on took time that in Firefox could have been spent developing functionality which is exposed to the user.

The lighter-weight approach of Firefox lends itself to rapid extension development which is accessible to users who may not even have any formal programming experience. Writing a small extension to do a simple task which improves the browsing experience in a single area is more common for Firefox, while IE, with its heavier-weight model and emphasis on static programming languages and COM support, tends towards a small number of large, multi-feature extensions such as IE7Pro¹³.

¹²The framework consists of only a subset of the tools available to developers through the SDK.

¹³IE7Pro is an extension which is designed to enhance Internet Explorer in a number of ways. It can be found at <http://www.ie7pro.com> and is discussed further in section 5.3.

3.5.2 Handling Errors

Internet Explorer is designed to avoid corruption or crashing of the main browser program by an add-on. However, we found the model chosen to handle the case when errors arise to be unintuitive when compared to error-handling methodologies employed by similar systems.

The least amount of error-related diagnostic information is given on load: if instantiating the add-on class fails, the add-on is simply not loaded at all. In addition, no indication is given to the user that this occurred—as far as we could tell, this information is not exposed anywhere. As the Windows error log contains a section dedicated to application errors, it is surprising that nothing is written there.

We determined that silent failure comes from two sources. The first reason the load might fail is the inability to resolve a dependency. If the add-on worked fine on the development machine but failed to load on a clean test machine, the probable reason was missing libraries. If all the dependencies were present, the second possible cause of failure is an exception thrown in the constructor. If an exception is thrown and not handled, it bubbles up to the add-on loader which ignores the exception entirely. Providing a log and initializing the log as the first operation the add-on performs can help counteract this problem.

After initialization, uncaught exceptions and other errors *are* exposed to the user. While we could depend on an operation that went wrong in the constructor silently failing rather than placing the application in an inconsistent state, exceptions thrown by code once the add-on has loaded are indeed sent to the user in the form of a dialog box.

As these errors are designed to assist developers in debugging, the error must be trapped and logged and then the user must be informed through a more user-centric message. Otherwise, the user experience is degraded when the user sees a cryptic error message.

3.5.3 Context Menu

Extending the context menu was the longest and most difficult single task of the development of LibX, and we found very few resources in the official documentation to help us. We were also surprised at the lack of API support from Internet Explorer for accomplishing this task.

Our experiences with the context menu in Internet Explorer and Firefox serve to illustrate the effectiveness of each browser's add-on model. While getting the context menu working in IE required interfacing directly with the Windows API and interposing with low-level operating system hooks, in Firefox the context menu is another piece of XUL, exposed equally to extensions and the core browser.

Allowing extension writers the same access to the browser GUI that the Mozilla Firefox developers themselves have means that Firefox extension developers do not need to depend on the presence of public extensibility APIs to do what they want. While IE provides mechanisms for extending the browser in ways the developers expected, attempting tasks that were not anticipated by the developers and thus not exposed through the API can be difficult. As a result, IE is almost completely inaccessible to the casual developer, while more experienced programmers are equally likely to prefer Firefox's model over IE's.

Chapter 4

Web Localization Framework

One of the key features of the LibX extension for Firefox is the insertion of visual ‘cues’ into the current page (figure 4.1 demonstrates cues on two different pages). As discussed in section 1.3.2 and overviewed in section 2.6.3, these cues provide a way for the user to interact with the page in context-sensitive manner.

Cues are a very page-specific concept. While autolinking (the other form of user interaction through page rewriting) could be performed on any page simply by analyzing its text, inserting a useful cue into a page requires very specific knowledge of both how the page is set up and what semantic content each element conveys. We needed a system to define page-specific rules to rewrite web pages.

4.1 Requirements

A framework to accomplish page-specific rewriting should meet several requirements:

- It should allow scripts to be created easily and updated simply.
- It should provide a simple but flexible interface for mapping URLs to scripts.



Figure 4.1: Cues for the VT edition inserted into the pages of two different online booksellers.

- It should separate the identification of web page elements from the manipulation of those elements.
- It should provide security to protect users from malicious scripts.

Each of these items is discussed below.

Easy Creation and Updating

First among these requirements, users must be able to create and update scripts without much effort. The easier it is to create scripts, the more likely it is that there will be a large number of useful scripts for LibX. Additionally, if edition maintainers can create scripts to augment those provided by LibX developers, the burden of creating and maintaining a comprehensive script library may be shifted more onto the community.

Updating scripts easily and automatically is also important. Sites may change their layout, breaking scripts which depend on the existing layout. Users may also wish to add additional functionality to their own scripts. To support these activities, the ability to seamlessly update scripts to the latest version should be available.

Selectively Executing Scripts

Adding to the idea of simplicity, the framework should provide an easy way to determine which scripts should be run. The ability to quickly identify sites which the script supports (and does not support) is required. It should also be easy to create a script which executes on all pages.

Separation of Identification and Manipulation

The two operations needed by each manipulation are the identification of page objects—text or images of interest—and the manipulation of those objects. We considered that multiple scripts might be interested in a single object, such as an ISBN. To optimize for this case, identification can be separated from action, allowing the identification stage to be reused with multiple actions, or the same action to be performed on multiple types of objects identified. The ability to associate as many actions as desired with the identification of an object reduces code duplication and increases the cohesion of the system.

Security

Finally, security concerns are prominent in the minds of both developers and users. We require that the system protect against malicious (or carelessly-implemented) scripts which steal user data

such as cookies or browsing history, or which attempt to access the user's filesystem outside of carefully-defined areas for storing and retrieving data. We chose these threats as being the most prominent to defend against as they are potentially the most difficult to recover from.

Though the original Firefox LibX had a system in place which accomplished some of these goals, we felt that it could be improved. As part of the work towards creating a more flexible and browser-agnostic JavaScript code base, we developed what we believe to be a much more robust, more interesting, and more general model for executing user scripts on arbitrary pages based on both URL matching and object identification.

4.2 Existing Design

The system which the original LibX Firefox extension used was split into two parts:

- **Autolinking** scans text nodes on each page for text tokens needing links.
- **DoForURL** maps scripts (written using JavaScript) to URL-matching patterns.

Each of these pieces is implemented by comparing an expression against the URL of the current page and executing associated code when a match is found. Autolink is implemented as a DoForURL action which matches all URLs and examines all text nodes on the current page. Other DoForURL scripts are run only when the current page's URL matches the specified pattern for the script.

The existing system allows the user to specify regular expressions that are matched against the URL on page navigation. Users can also provide expressions to exclude pages which would otherwise be matched, preventing false positives. In addition, the matches returned from the regular expression allow portions of the URL to be captured and then used as inputs to the script itself.

The DoForURL system provides security by only executing scripts from trusted sources. A trusted source is defined as either a developer with commit access to the LibX source code or an edition maintainer who chooses to edit the source code for one particular edition. After the package is built, there is no system for integrating additional DoForURL rules. This system provides a high degree of security, but flexibility is reduced as a result.

Although autolink and DoForURL offer enough functionality to allow LibX developers to insert new rules and scripts in to the source, it lacks the flexibility of allowing third-parties to distribute scripts using the same system. It also does not separate object identification from page action, thus preventing new scripts from easily re-using identified objects from already-existing scripts. We felt that it would be possible to provide a new mechanism that more fully satisfied the goals

for the system.

4.3 New Design

The result of our planning was the design for a completely new system. The new framework provides complete separation between object identification and page rewriting actions. It also provides a shared space for communication between scripts without introducing coupling, and features an update mechanism that uses existing web formats. It also uses a sandbox for script execution, providing enhanced security.

The current system is still undergoing implementation as of this writing. While the design is complete at a high level, implementation details are still very much in flux.

4.3.1 Object Identification and Manipulation

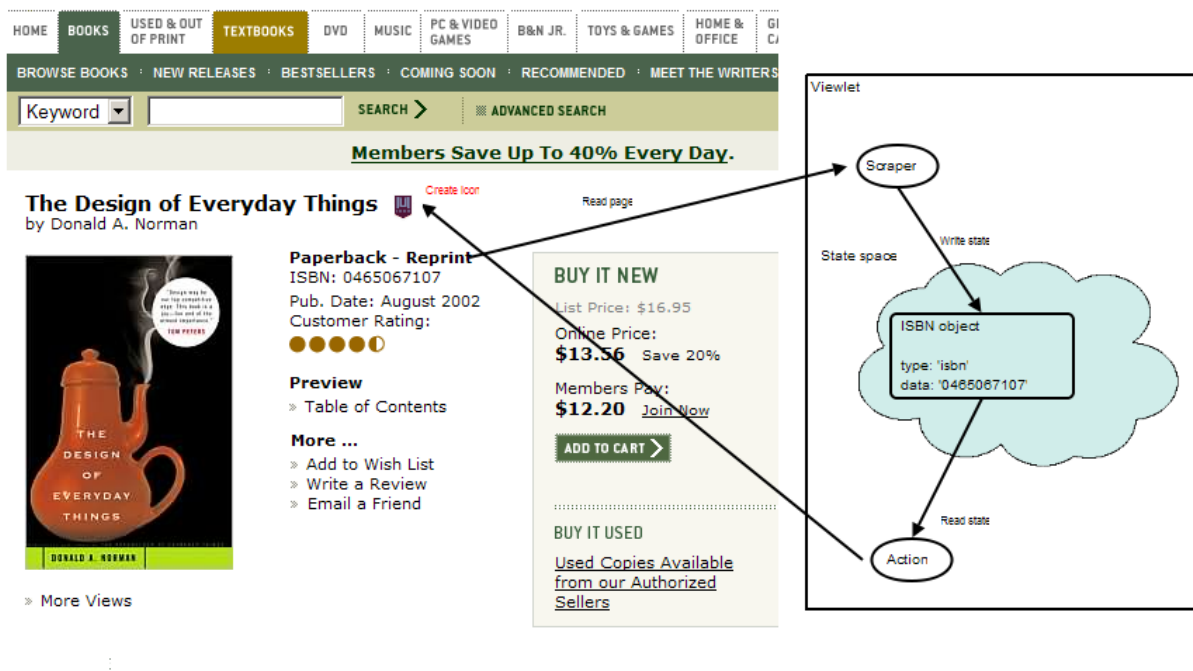


Figure 4.2: Communication between scrapers and actions using the shared state space.

We first specified that the page scanning and object identification portion should be separate from the object manipulation portion. In separating them, we created a composable system of ‘scrapers’, which ‘scrape’ the page looking for specific HTML or text patterns; ‘actions’, which manipulate the HTML or text in certain ways; and ‘viewlets’, which can provide custom mappings between actions and scrapers.

Scrapers run first under most scenarios, because they identify sites of interest on the page for actions to manipulate. Such a site might be the location of an ISBN in the middle of a paragraph, or the HTML element containing the title of a book on an online bookseller product page. Scrapers then give these objects IDs and place them in a shared space where they are accessible both by actions and other scrapers (this space will be discussed in the next section, 4.3.2).

Actions are similar to scrapers, but instead of finding objects, they make use of objects already located. An action might convert an ISBN into a link to the library's page or insert a cue next to a book title located by a scraper. Actions are also composable, so they are able to communicate with other actions. This might take the form of multiple actions creating elements which are then used by another action to create a floating information window that contains the elements created by previous actions. The code for implementing the window only must be written once, and it can aggregate content created by other actions.

A viewlet is a method of combining scrapers and actions and executing them according to a specified order. Viewlets can reference either elements within the feed or external elements made available by others.

Figure 4.2 shows a simple viewlet containing one scraper and one action. The scraper reads and stores the ISBN it locates, and the action retrieves the stored ISBN and then links it on the page.

4.3.2 Shared State Space

A viewlet's scrapers and actions communicate using a shared space for reading and writing state. The shared space provides a medium for data transmission that reduces or eliminates coupling between entities but still allows for composition, where some actions or scrapers build from data gathered or created by others.

Placing an object into the shared space does not impose any requirements on any other part of the system. If the object goes unread by other scrapers and actions, it is simply not acted upon. If multiple actors can consume the object, each can read the object from the space without affecting each other reader.

Figure 4.2 shows an example of how the shared state space is used to communicate. On the example document fragment from the Barnes & Noble website¹, a scraper can locate the ISBN for the book being displayed. It would then insert an object into the state space containing what the number is and its location on the page. Then an action can read that object in and use its information to place a link next to the book's title.

¹<http://www.bn.com>

4.3.3 Feeds and Updating

We wanted script developers to be able to update scripts at will, both publishing new actions, scrapers, or viewlets and editing or removing existing ones. This came from our experiences with the existing system: when a site we processed changed its layout in such a way that the cues were not being properly inserted, updates to fix this could not be pushed out independently. Instead, we required edition maintainers to build a new edition just to apply the updates. By separating the script update system from that of LibX, we could change this behavior.

To implement this change, we decided that scripts would be contained in syndication feeds in Atom format [26]. Syndication feeds, here just called feeds, are an XML-based system for publishing content. They see widespread browser support, with most modern browsers offering the ability to subscribe to feeds as a built-in feature. Feeds are periodically refreshed, and when they have changed, it indicates to the underlying system that they should be reparsed for new information—information such as new articles or stories published for a magazine site, or new versions available for a software site.

These feeds can contain scrapers, actions, and viewlets. Users subscribe to feeds, and when a developer wishes to update a script or provide additional functionality, the feed can be updated. The change will be caught the next time the server is polled, and the new scripts will be downloaded. This mechanism, which integrates with an existing, widely-implemented web technology (Atom), offers a large degree of flexibility for script authors and is separate from the LibX update system, thus fulfilling our requirements for script updating.

4.3.4 Sandbox

The sandbox, our enhancement to the system's security, is a specialized execution environment in which scripts are run. Although client-side JavaScript is itself run in a sandbox, there are features allowed to JavaScript that we do not wish to allow third-party scripts. This is especially the case for browsers such as Firefox, where add-ons are run in a privileged environment and could otherwise steal or even corrupt user data.

Sandboxing restricts the functions available to a script, but there is a trade-off between providing full security and giving power to script developers. We give full document access to scripts, while acknowledging that it would be possible for a script to subtly and maliciously alter the document. However, we deny all access to the Mozilla filesystem functions, reasoning that allowing access to the local filesystem even at a limited level would do more harm than good.

We do not yet have in place a full security model, capable of defending against threats such as malicious scripts reading and transmitting cookie data. We are aware that the problem exists, but

the solution is both complex and not well-defined. The following section discusses implications of this framework, including the possibility of malicious scripts, and presents potential mitigations.

4.4 Implications

Providing the ability for a third party to modify web pages does have potential implications, both positive and negative. It is our hope that providing a flexible, powerful, and accessible framework for integrating third-party content and information into web sites will be both useful and beneficial. However, we do acknowledge that web content providers may wish to retain control over their own information, and our framework removes some of this control.

Undoubtedly the biggest controversy with this technology is that web page authors cannot control the ways their pages are rewritten once the page leaves the server and is sent to the user's browser. Even for scripts distributed with LibX already, booksellers could argue that providing links to library copies of books viewed by the user leads to lost revenue. Even legitimate scripts distributed by online merchants to place links on competitors' sites would probably be viewed as undesirable; a less ethical script which, unbeknownst to the user, redirected clicks on products from one vendor's page to product display pages on a competitor's site would be even less welcome.

A similar problem was solved by search engines when large-scale content indexing and caching was causing site maintainers to be concerned that there existed copies of their data which they had no control over. If a site contains a file named 'robots.txt', this file is parsed by search engines. The file can request the search engine's indexer to not catalog or cache the site.

It would be possible to duplicate this behavior by providing meta tags on each page instructing a framework to not run user scripts on that page. However, implementing such a system might end up limiting the user: if retailers, concerned about competitors' scripts altering their pages, included this tag, helpful scripts such as those provided by LibX (or even those designed to provide more information about the product without drawing customers away from the page) could not run.

While the user is unlikely to intentionally install a script which would then maliciously deface or alter a page in a way that would defeat the original purpose of the page, it is possible for a script to be the equivalent of a Trojan horse, offering an ostensibly useful service which conceals the true, less ethical, purpose.

To combat these types of scripts, it would be possible to maintain a community blacklist of scripts which are known to be detrimental. Since the source code of each script is necessarily available to users, it would be possible for either a developer or an interested user familiar with JavaScript to scan the code manually and suggest to a central list maintainer that a script be blacklisted if it

appears to be either purposefully or inadvertently hindering the user's browsing experience.

It would also be possible to adopt an approach similar to what Firefox uses for its own extensions. Rather than relying on a blacklist, the browser contains a whitelist of domains known to contain trusted, well-tested extensions. Any extension hosted on an unlisted domain is not allowed to be installed, unless the user explicitly places the domain on the list.

These implications are important to consider, and we want to create a safe and positive user experience by providing a completely open model. Making the code of all scripts as well as the code of the framework itself open and available will expose the entire system to scrutiny that can flag malicious behavior and identify programming errors that would lead to inadvertent security holes. By involving the community in the entire process, we hope to help address objectionable scripts and concerns from web developers in an open and fair way.

Chapter 5

Related Work

5.1 Library Tools

There are a very large number of library tools available for enhancing a user's browsing experience. The libsuccess wiki¹ contains a large list of library-related tools for the web browser. However, many of these are specific to a single organization or library rather than being more generally applicable. The existence of the same type of tool for so many different institutions suggests that there is a perceived need for such solutions, and that a general solution which can be localized by multiple universities and libraries is an important service.

A more general tool, and one that popularized the bookmarklet approach to browser-based library integration, is Jon Udell's LibraryLookup tool². LibraryLookup acts as a code generator to create a small bit of JavaScript code which can be stored as a bookmark (this is referred to as a bookmarklet) and executed by selecting the bookmark from the browser's interface. When executed, the JavaScript opens a window navigating to the library listing for that book. As the code for the bookmarklet was generated specifically for the user's library, it can provide localized service. LibraryLookup works on a per-page basis and provides no user interaction except for the single bookmarklet.

Also offered as a bookmarklet (as well as a Firefox extension) is the Windsor-Alberta-Georgia 'WAG the Dog' tool³. This tool is designed to enhance Google Scholar searches by modifying links to use institutional proxy servers and by providing buttons linking to an OpenURL resolver. The bookmarklet and extension redirect Scholar searches to a proxy server, which performs the page modification. While the resolver is configurable, WAG the Dog is designed solely for use with the

¹http://libsuccess.org/index.php?title=Web_Browser_Extensions

²<http://weblog.infoworld.com/udell/stories/2002/12/11/librarylookup.html>

³<http://gslocal.sourceforge.net/>

Google Scholar site.

Another tool of interest is the OCLC⁴ OpenURL Referrer software⁵, which is offered as a Firefox browser extension. This software locates COinS (Context Objects in Spans) in the current webpage and links them to a configured OpenURL resolver. By allowing the resolver to be user-configurable, it provides localization for sites (such as Google Scholar) which use COinS to help identify library information in the document.

The final tool in this section is Zotero⁶, developed at George Mason University. Zotero is a tool for collecting and annotating references through the web by capturing bibliographic information. It is another tool designed as an extension for the Firefox browser. Zotero works primarily through configured rules for known websites from which information can be extracted.

Library-related tools such as these replicate functionality offered in LibX. We have not, however, seen a tool offering the full range of functions LibX provides. Most of these tools are also exclusive to Firefox, with the exception of bookmarklet-based tools, which sacrifice a user interface but gain the ability to be used across multiple browser platforms.

5.2 Internet Explorer Toolbars

Though most library resources are targeted towards the Firefox platform, there are a number of toolbars available for Internet Explorer. Among the more general toolbars are those offered by general web search providers. Google, Yahoo!, and MSN all provide toolbars which enhance existing functionality and add new features to the browser. While not specific to library functionality, these toolbars are all freely-available for Internet Explorer.

OCLC did, however, collaborate with Yahoo! to produce a version of the Yahoo! Internet Explorer toolbar enhanced with support for the OCLC WorldCat bibliographic database⁷. This toolbar is focused on searching, and also allows users to locate libraries near to them geographically. It also provides the features of the standard Yahoo! toolbar, such as pop-up blocking and general web search.

Another add-on for Internet Explorer is OAses⁸, a toolbar for searching open access resources such as the Directory of Open-Access Journals⁹ and the Project Gutenberg repository of electronic text¹⁰. It does not possess localization functionality, but instead focuses on searching resources

⁴Online Computer Library Center

⁵<http://www.openly.com/openurlref/>

⁶<http://www.zotero.org/>

⁷<http://www.oclc.org/toolbar/>

⁸<http://www.psyplexus.com/oases/>

⁹<http://www.doaj.org/>

¹⁰<http://www.gutenberg.org>

which are freely available to everyone.

The majority of related Internet Explorer toolbars are primarily focused on searching. Most toolbars also provide additional functionality such as pop-up blocking and integration with whatever specialized services the toolbar vendor offers.

5.3 Web Rewriting and Localization

There exists a large variety of tools available for modifying, remixing, aggregating, and presenting web content in user-driven ways. These tools are designed to provide services and information not found on any one web page, but instead taken as a composite and localized to be relevant to the specific user viewing the data.

A driving force behind this sort of tool is the idea of the semantic web [7]. The concept of the semantic web is that web pages should be annotated by metadata containing the semantic significance of data using some format such as the Resource Description Format (RDF). This metadata should be then machine-parsable, allowing computer systems to better analyze the content of web pages at a semantic level, providing more accurate results when searching and the ability to automatically locate and aggregate requested data from previously unknown sources.

One example of a program that provides this functionality is Annotea¹¹ [20]. Annotea is a project which allows users to bookmark and annotate fragments of a website and then share those annotations through RDF files. These annotations can be presented for human interpretation or used by computer systems to recognize desired data. A client implementation currently exists for the Amaya browser¹².

Another semantic web project is Thresher [18]. Thresher is a system designed to work inside the Haystack semantic web browser¹³, and allows users to interactively locate desired semantic content. Generalized DOM scrapers are then created to locate and extract that content. It allows both users and content providers to mark up their documents with semantic information using RDF.

Related to the Haystack browser is the Piggy Bank project¹⁴ [19], also developed at the MIT AI labs. Piggy Bank allows users to extract data from web sites, save them, and combine them through RDF. It uses screen scraping through queries on the HTML DOM (rather than regular expression parsing on the HTML text itself), and allows extracted data to be tagged, aggregated, and published. It aims to help accomplish the semantic web goal by allowing one particular datum to be bookmarked and then incorporated in another web site's information, enhancing the value

¹¹<http://www.w3.org/2001/Annotea/>

¹²<http://www.w3.org/Amaya/>

¹³<http://groups.csail.mit.edu/haystack/>

¹⁴http://simile.mit.edu/wiki/Piggy_Bank

of both data.

Not overtly part of the semantic web movement but similar in concept is the idea of personalized home pages. Sites such as My Netscape¹⁵ and iGoogle¹⁶ allow users to mix in information such as syndicated content through RSS and Atom feeds, calendar events, news items, and frequently-visited sites and create a personalized home page for each user.

Other programs aim to achieve aggregation and localization through deep knowledge of page content rather than through semantic metadata. For instance, the Greasemonkey¹⁷ Firefox extension allows user scripts (such as those found on the community site Userscripts.org¹⁸) to be executed on pages based on URL matching. Instead of depending on metadata available through RDF or similar means, Greasemonkey executes site-specific scripts to perform specific tasks and modify pages based on existing rules.

Another system that performs similarly to Greasemonkey is the Chickenfoot Firefox extension from MIT¹⁹ [8]. Like Greasemonkey, it provides a JavaScript environment for executing user scripts that modify the page based on certain rules.

For Internet Explorer, there exists an add-on called IE7Pro²⁰ which provides equivalent functionality to that of Greasemonkey and Chickenfoot for Firefox. User scripts (obtainable from sites such as IEScripts²¹) are executed in a special environment to modify known pages identified through URL matching. IE7Pro, which also works with IE6, is designed to be an all-around enhancement to Internet Explorer, and provides additional functionality as well.

Another idea behind localizing web information is in aggregating and syndicating search information, as specified in OpenSearch²². OpenSearch is a collection of various formats used to publish and aggregate search results. It is implemented by several search engines, such as the Amazon A9 search²³, and in addition is supported by browser clients such as Firefox and Internet Explorer 7 to provide syndicated searching and results retrieval.

These works all offer functionality similar to the web localization features of LibX. Greasemonkey and IE7Pro in particular offer the same level of functionality that we have developed for our web page rewriting design. However, all of these tools are specific to a single browser, while LibX provides a platform for multiple browser implementations. LibX also offers a large number of

¹⁵<http://my.netscape.com/>

¹⁶<http://www.google.com/>

¹⁷<https://addons.mozilla.org/en-US/firefox/addon/748>

¹⁸<http://userscripts.org/>

¹⁹<http://groups.csail.mit.edu/uid/chickenfoot/>

²⁰<http://www.ie7pro.com/>

²¹<http://iescripts.org/>

²²<http://www.opensearch.org/>

²³<http://www.a9.com>

first-party scripts which provide direct access to library resources, while many other frameworks listed here depend primarily on user-created content for their usefulness.

Chapter 6

Summary and Conclusion

6.1 Summary

LibX is a browser add-on that addresses the problem of students choosing to use general web search engines, which generally produce results with questionable academic integrity, rather than library resources while doing research. Our goal was to take the existing Firefox implementation of LibX and, while reusing as much of the existing JavaScript implementation as possible, develop it into a browser-agnostic framework. By finding a way to run the JavaScript code from C# in a privileged environment and then use this environment in an Internet Explorer add-on, we accomplished this goal. Though some parts are written mostly in C#, others are entirely or almost-entirely shared, non-browser-specific JavaScript code. The result is a browser toolbar and extension that has all the features of the Firefox browser and shares a significant portion of the JavaScript code.

While performing this task, we also developed the concept of client-side web page rewriting done in LibX into a general-purpose web localization framework. The framework separates object location from the actions performed on the located objects. It also provides a mechanism for composing the location of objects and the actions in various ways for optimum flexibility. Finally, it allows third-party developers to not only create their own scripts but combine existing scripts in new ways, or to use existing object identifiers but associate new actions with them.

6.2 Future Work

6.2.1 Internet Explorer Add-On

The LibX Internet Explorer add-on provides many rich avenues for future work. A possible area of development is the development of an abstraction for UI manipulation from JavaScript, allowing more of the interface code to be unified across both browsers. Another task that would increase the amount of code sharing is the emulation of the Firefox chrome space, allowing `chrome:` URLs to be interpreted correctly.

We also believe that our context menu extension mechanism is of general interest, apart from LibX. While it is already very general to support LibX's flexible menu system, it could be removed from LibX and further generalized as a support library. Used by any arbitrary add-on, it could allow developers to simply and dynamically augment the IE context menu both from .NET code and from JavaScript, providing a feature IE does not offer natively.

Finally, LibX provides an interesting subject for user interaction analysis. More research could be done to determine how well our existing system of context and cues integrates with user workflows, which would both allow LibX's user interface to be streamlined based on user input and provide additional data regarding the utility of integrating this style of interaction into the web browser.

6.2.2 Web Localization Framework

The web localization framework presents even more varied opportunities for exploration and research. One area that will undoubtedly be important going forward is a further investigation of the attitude of web developers towards a system that can alter the presentation and content of their sites without their input.

In addition, usability studies could help find the most effective way to augment the contents of a web page to provide an optimal browsing experience and integrate information from other sites into what the user sees. With such a large amount of flexibility in terms of information that *could* be aggregated and presented to the user, it is important to know with greater certainty what types of information would be helpful to the user and how that information should be presented.

On the software development side, we believe that the web localization framework is of general interest outside of LibX. We would like to see either a version of LibX independent of any library or a stand-alone cross-browser add-on that LibX would integrate with but which could also provide object location and page rewriting services without LibX even being installed. While we believe that the services LibX provides are helpful and unobtrusive, we acknowledge that not all libraries

offer a LibX edition. Even those who do not have access to a LibX-supporting library, however, could easily find a general web script execution tool useful.

We also believe that expanding the security model to perform both static and dynamic analysis of script content would help further guard against malicious script writers. This type of system would decrease the possibility of a harmful payload being hidden by a seemingly-useful script. A more complete dynamic analysis solution would also expand the types of attacks we could protect against, such as attempts to crash the web browser or perform denial-of-service attacks against a web site.

6.3 Conclusion

With web-based applications becoming more popular, the browser is also becoming an increasingly attractive target for extension development. Much like programming for multiple OSes, unfortunately, programming for multiple browser platforms tends to result in an application that only supports the ‘least common denominator’—only the feature set that is easy to implement in every browser. Despite this tendency, we neither had to remove features from the Firefox version nor cut any features from the Internet Explorer version when developing LibX.

We did find that, while JavaScript is supported across most browsers, it is hardly a *lingua franca* for add-on developers. Feature support and browser integration vary too much between browser platforms once it becomes necessary to use features outside the well-specified ECMAScript standard for client-side JavaScript.

We believe, however, that the solution we have produced in LibX for Internet Explorer strikes a good balance between achieving a concise and comprehensible implementation and keeping the control logic general enough and common enough that it can be shared between browsers, lightening the maintenance load. It is our hope that what we have designed adds meaningfully to the state of the art, and that, as an open-source project, helps provide guidance and inspiration to others attempting to solve the same problems.

We also hope that the work we have done on interacting with the user proves to be a significant contribution to the understanding of how the Web facilitates user interface design. By making use of client-side technology to integrate communication between user and program into the standard workflow, we hope the result is a feature set which feels natural and well-integrated to the user while being unobtrusive when its services are not needed.

Ultimately, these developments will put more power and control into the hands of the users, where it belongs. By providing users with easy access to resources, an intuitive interface, and the power to customize their web environment and share those customizations with others, we try to make the

full range of resources available from university and local libraries more accessible to the patrons of those libraries. In doing so, we improved the experience of using library resources and provided a way through which other services and resources can be exposed naturally to the user.

Bibliography

- [1] CQL: Contextual query language. <http://www.loc.gov/standards/sru/specs/cql.html>.
- [2] Platform invocation services. <http://msdn2.microsoft.com/en-us/library/Aa712982.aspx>.
- [3] A. W. Appel. Simple generational garbage collection and fast allocation. *Software: Practice and Experience*, 19(2):171–183, 1989.
- [4] Giuseppe Attardi, Antonio Cisternino, and Diego Colombo. CIL + metadata – > executable program. *Journal of Object Technology*, 3(2):19–26, 2004.
- [5] Annette Bailey and Godmar Back. LibX – a firefox extension for enhanced library access. *Library Hi Tech*, 24(2):290–304, April 2006.
- [6] Annette Bailey and Godmar Back. Retrieving known items with LibX. *The Serials Librarian*, 53(4), 2007.
- [7] Tim Berners-Lee. Semantic web road map. 1998. <http://www.w3.org/DesignIssues/Semantic.html>.
- [8] Michael Bolin, Matthew Webber, Philip Rha, Tom Wilson, and Robert C. Miller. Automation and customization of rendered web pages. In *UIST '05: Proceedings of the 18th annual ACM symposium on User interface software and technology*, pages 163–172, New York, NY, USA, 2005. ACM Press.
- [9] Don Box and Chris Sells. *Essential .NET: The Common Language Runtime*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
- [10] Michele Leroux Bustamante. Build a managed BHO and plug into the browser. *15 Seconds*, 2004. <http://www.15seconds.com/issue/040331.htm>.
- [11] Grant D. Campbell and Karl V. Fast. Panizzi, lubetzky, and google: How the modern web environment is reinventing the theory of cataloguing. *Canadian Journal of Information & Library Sciences*, 28(3):25–38, September 2004.

- [12] Steven J. Davis and Kevin M. Murphy. A competitive perspective on internet explorer. *American Economic Review*, 90(2):184–187, May 2000. available at <http://ideas.repec.org/a/aea/aecrev/v90y2000i2p184-187.html>.
- [13] Dino Esposito. Browser helper objects: The browser the way you want it. *Microsoft Developers Network*, 1999. <http://msdn2.microsoft.com/en-us/library/bb250436.aspx>.
- [14] David Flanagan. *Javascript: The Definitive Guide*. O’Reilly, 5th edition, 2006.
- [15] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley Professional, January 1995.
- [16] Jennifer Hamilton. Language integration in the common language runtime. *SIGPLAN Notes*, 38(2):19–28, 2003.
- [17] Jr. Harold W. Lockhart. *OSF DCE: guide to developing distributed applications*. McGraw-Hill, Inc., New York, NY, USA, 1994.
- [18] Andrew Hogue and David Karger. Thresher: automating the unwrapping of semantic content from the world wide web. In *WWW ’05: Proceedings of the 14th international conference on World Wide Web*, pages 86–95, New York, NY, USA, 2005. ACM Press.
- [19] David Huynh, Stefano Mazzocchi, and David Karger. Piggy bank: Experience the semantic web inside your web browser. *Journal of Web Semantics*, 5(1):16–27, 2007.
- [20] José Kahan and Marja-Ritta Koivunen. Annotea: an open RDF infrastructure for shared web annotations. In *Proceedings of the 10th International World Wide Web Conference*, pages 623–632, 2001.
- [21] Serge Lidin. *Inside Microsoft .NET IL Assembler*. Microsoft Press, Redmond, WA, USA, 2002.
- [22] Clifford A. Lynch. The z39.50 information retrieval protocol: an overview and status report. *SIGCOMM Computer Communication Review*, 21(1):58–70, 1991.
- [23] Sally McCallum. A look at new information retrieval protocols: SRU, OpenSearch/A9, CQL, and XQuery. *72nd IFLA General Conference and Council*, 2006.
- [24] Paul Miller. Z39.50 for all. *Ariadne*, Issue 21, 1999.
- [25] Eric Morgan. An introduction to the search/retrieve url service (SRU). *Ariadne*, Issue 40, 2004.
- [26] M. Nottingham and R. Sayre. The atom syndication format. <http://tools.ietf.org/html/rfc4287>.

- [27] Linda Dailey Paulson. Building rich web applications with ajax. *Computer*, 38(10):14–17, 2005.
- [28] Jeffrey Richter. Garbage collection: Automatic memory management in the microsoft .NET framework. *MSDN Magazine*, 15, 2000.
- [29] Jeffrey Richter. Garbage collection: Automatic memory management in the microsoft .NET framework, part 2. *MSDN Magazine*, 15, 2000.
- [30] Dale Rogerson. *Inside COM*. Microsoft Press, Redmond, WA, USA, 1997.
- [31] Jeremy Singer. JVM versus CLR: a comparative study. In *PPPJ '03: Proceedings of the 2nd international conference on Principles and practice of programming in Java*, pages 167–169, New York, NY, USA, 2003. Computer Science Press, Inc.
- [32] Herbert Van De Sompel and Oren Beit-Arie. Open linking in the scholarly information environment using the OpenURL framework. *D-Lib*, 7, 2001.
- [33] Sara Williams and Charlie Kindel. The component object model: A technical overview. *Microsoft Developer Network*, 1994.
- [34] Pavel Zolnikov. Extending explorer with band objects using .NET and windows forms. *The Code Project*, 2002. <http://www.codeproject.com/csharp/dotnetbandobjects.asp>.