The LibX Edition Builder

Tilottama Gaat

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
Edward Fox, Committee Member
Eli Tilevich, Committee Member

23 July, 2008
Blacksburg, Virginia

Keywords: LibX, AJAX, Code Generation, Log-based usability evaluation
Copyright 2008, Tilottama Gaat
The LibX Edition Builder
Tilottama Gaat

ABSTRACT

LibX is a browser plugin that allows users to access library resources directly from their browser. Many libraries that wished to adopt LibX needed to customize a version of LibX for their own institution. Most librarians did not possess the necessary knowledge of XML, running scripts and the underlying implementation of LibX required to create customized, functional LibX versions for their own institutions. Therefore, we have developed a web-based tool called the LibX Edition Builder that empowers librarians to create their own customized LibX versions (editions), effortlessly.

The Edition Builder provides rich interactivity to its users by exploiting the ZK AJAX framework whose components we adapted. The Edition Builder provides automatic detection of relevant library resources based on several heuristics which we have developed, which reduces the time and effort required to configure these resources.

We have used sound software engineering techniques such as agile development principles, code generation techniques, and the model-view-controller design paradigm to maximize maintainability of the Edition Builder, which enables us to easily incorporate changing functional requirements in the Edition Builder.

The LibX Edition Builder is currently used by over 800 registered users who have created over 400 editions. We have carried out a custom log-based usability evaluation that examined the interactions of our users over a 5 month period. This evaluation has shown that the Edition Builder can dramatically reduce the time needed to customize LibX editions and is being increasingly adopted by the library community.
I would like to thank Dr. Back for entrusting me with an important project like the LibX Edition Builder and showing faith in me. I am indebted to him for his help, support and guidance throughout the course of this thesis. I am indeed glad to have had this prized opportunity to work with such a learned and helpful advisor.

I would like to thank Dr. Eli Tilevich whose talks about academics, careers and life were a source of education and enjoyment for all of us.

I would like to thank Dr. Fox for his insightful comments about the thesis and about the time and patience he spent in understanding my work.

I would like to thank Kyrille Goldbeck and Annette Bailey for their immense help and valuable inputs about the features of the Edition Builder and the usability analysis. Thanks again to Kyrille for reading one of the most important chapters of this thesis.

I would like to thank all the illustrious professors of the CS department, whose courses I have taken.

I would like to thank my friends Jai, Sneha and all my other friends in Blacksburg for their enjoyable company. I also want to thank Tanushree, my first friend in Pullman, WA.

I would like to thank Ashwin Aji who has been a friend, critic, philosopher, and guide – a constant source of inspiration and support in more ways than one.

I would like to thank my parents for encouraging me to pursue higher education at Virginia Tech and for standing by me through thick and thin. I am also eternally thankful to my grandparents, my aunt and my cousins. And last but not the least; I would like to thank my two lovely sisters for always being there for me.
Contents

List of Figures viii

List of Tables xi

1 Introduction 1

1.1 Background about LibX 2

1.2 Motivation for the Edition Builder 2

1.3 Goals 3

1.4 Results 6

1.5 Contributions 10

1.6 Document Overview 10

2 Background 11

2.1 AJAX 11

2.1.1 Need for AJAX Frameworks 11

2.1.2 Classification of AJAX Frameworks 14
2.1.3 Our Choice – ZK .................................................. 17
2.2 Model View Controller Design Pattern ......................... 18
2.3 XML Processing .................................................. 22

3 Design and Implementation of the Edition Builder ............... 26

3.1 Goals .................................................................. 26
  3.1.1 Web Based Deployment ..................................... 26
  3.1.2 Rich User Interaction ........................................ 27
  3.1.3 Auto-Discovery of Library Resources ....................... 29
  3.1.4 Use of Sound Software Engineering Techniques ......... 30
  3.1.5 Storage Scheme ............................................... 31
3.2 Edition Builder Functionality ..................................... 32
  3.2.1 Overview and General Features .............................. 33
  3.2.2 Edition Life Cycle Management .............................. 35
  3.2.3 Configuring Resources ....................................... 38
3.3 Architecture of the Edition Builder ................................ 41
3.4 Auto-Discovery .................................................. 44
3.5 Maximizing Maintainability using Castor and MVC .......... 48
  3.5.1 Agile Development using Castor ............................ 48
  3.5.2 Application of MVC ......................................... 50
3.6 Storage Scheme ................................................................. 52
   3.6.1 Configuration Repository ........................................... 52
   3.6.2 Database Design ....................................................... 53
3.7 Implementation Experiences ........................................... 55

4 Usability Evaluation .......................................................... 57
   4.1 Effectiveness and Efficiency ........................................... 59
      4.1.1 Cumulative Distribution of Time Taken to Make Editions Live 59
      4.1.2 Complexity of Configurations ................................... 59
   4.2 Intuitiveness of the User Interface ................................... 61
      4.2.1 Number of Live Editions Maintained Per User ................. 61
      4.2.2 Usage of the Help Tooltips ...................................... 63
      4.2.3 Choice of Edition Initiation .................................... 67
   4.3 Usage of Auto-Detection Features .................................... 67
      4.3.1 Usage of the Auto-discovery Feature Per User ............... 67
      4.3.2 Categorization of Auto-detected Resources by Type .......... 69
   4.4 Adoption of the Edition Builder ..................................... 72
   4.5 Community Sharing of Configured Catalogs ....................... 73
   4.6 Conclusion .............................................................. 74

5 Related Work ................................................................. 78
# List of Figures

1.1 Architecture of the Edition Builder ........................................... 7

2.1 Asynchronous Communication in AJAX ........................................ 13

2.2 Axes for Comparing AJAX Frameworks ....................................... 14

2.3 Client Centric vs. Server Centric Frameworks .............................. 17

2.4 Architecture of ZK .................................................................. 19

2.5 Interaction between MVC ......................................................... 21

3.1 Overview of the Edition Builder Interface ................................. 33

3.2 User Login ........................................................................... 34

3.3 Help Tooltips ................................................................. 35

3.4 Initiating Editions from Scratch ............................................ 36

3.5 Initiating Editions by Cloning .................................................. 36

3.6 Revision Management .......................................................... 38

3.7 Configuring Shortcuts ........................................................... 38
3.8 Configuring Catalogs ......................................................... 39
3.9 Configuring OpenURL Resolvers ........................................... 40
3.10 Configuring Proxy ............................................................ 40
3.11 Configuring Files ............................................................. 41
3.12 AutoDiscovery of OPACs .................................................... 42
3.13 Architecture of the Edition Builder ...................................... 43
3.14 Flow of Information in Auto-Discovery Process ....................... 46
3.15 Use of Castor for Quick Deployment of Changes ...................... 49
3.16 Entity Relationship Diagram for the Database Design ............. 55

4.1 Cumulative Distribution of Time to Make an Edition Live ............ 60
4.2 Number of Configured Shortcuts .......................................... 62
4.3 Number of Configured Catalogs .......................................... 63
4.4 Number of Configured Proxies ............................................. 64
4.5 Number of Configured Files ............................................... 65
4.6 Live Editions Per User ....................................................... 66
4.7 Autodetection of Catalogs - Per User Usage ........................... 68
4.8 Autodetection of OpenURL Resolvers - Per User Usage ............ 69
4.9 WorldCat Registry Search - Per User Usage ......................... 70
4.10 Autodetection Breakdown by Type ....................................... 72
4.11 Number of Public Editions Over Time ........................................... 74
4.12 Number of All editions Over Time ............................................. 75
4.13 Number of Registered Users ..................................................... 76
4.14 Development of Shared Catalogs Over Time ................................. 77
List of Tables

2.1 Differences between SAX and DOM Approaches .......................... 23
2.2 Differences between DOM and Data Binding Methods .................. 24
More and more students are choosing to turn to search engines during their research rather than library resources. The reason for this phenomenon is that students find it cumbersome to navigate to the library homepages to initiate their searches, whereas search engines are more easily accessible [CF04]. Search engines today are exceedingly popular and have been integrated in the day-to-day work flow of students. Moreover, search engines have a user interface that is simpler in comparison with the complex one presented by catalogs or OPACs (Online Public Access Catalogs).

This lack of usage of libraries is a cause of concern for the entire academic community. Libraries contain books and other resources published after being reviewed by peers and selected by librarians, whereas material offered by search engines includes resources such as blogs that do not carry as much academic weight as published material found in libraries. In addition, some resources require authentication from the library. Search engines may not have access to those, and thus may fail to reveal them.
1.1 Background about LibX

LibX [BB06] is a browser plugin that offers library OPAC searches via a toolbar. LibX avoids repeated navigation to the library home page and integrates libraries with the usual work flow for students and researchers. In addition to the toolbar module, LibX provides a context menu module that allows users to create searches by selecting text on web pages. LibX embeds icons called cues as annotations on web pages such as Amazon.com, New York Times Book Review Section, Google, and others. Cues enable a user to relate the material on that web page with resources in the library and find out if it is available in the library. LibX also allows access to library resources from off-campus. LibX supports searches in Google Scholar and redirects users to an accessible copy of the searched resources in the institution library. Thus, LibX is an infrastructure for library services on the client side.

1.2 Motivation for the Edition Builder

Many universities recognized the utility of LibX and wished to adopt LibX for use by their own institution. To adopt LibX, the librarian of that institution needs to customize LibX as per the requirements of that institution. This customized version of LibX is called an edition.

The responsibility of adapting LibX for a university lies with the librarian community of that institution because people working in the domain of libraries can accomplish the task of configuring LibX for their community successfully. To customize LibX, it is necessary to incorporate resources such as OPACs and databases, logos, useful links and others that are relevant to that institution into a configuration file. This task is non-trivial. To customize each resource many settings have to be specified. For example, to configure a single catalog
the name of the server which houses this catalog and the indices to be used with search
terms need to be specified. These settings have to be completed in a format specific to the
underlying implementation of LibX that is known only to the developers of LibX. Conse-
quently, librarians asked the creators of LibX to develop editions for their universities. The
creators of LibX manually created editions for 150 libraries, but the demand for editions was
increasing at a pace which became difficult for the creators of LibX to satisfy. Providing doc-
umentation to librarians about completing settings correctly would not have been sufficient
because librarians would have still needed to edit the configuration XML (Extensible Markup
Language) file with the required settings and run a script that takes this file as input and
creates a version of LibX. Thus for creating an edition for their own library, librarians would
have needed to work with XML, a technology with which most librarians are unfamiliar. It
would have been difficult for them to accomplish building an edition without resorting to
the help of someone familiar with the syntax and the semantics of XML.

1.3 Goals

To avoid burdening the LibX developers with creating customized versions of LibX for all
universities, a tool that empowers librarians to build their own editions is required. We
developed a tool to instruct librarians in building their edition in a step-by-step fashion and
output ready-made customized editions. We have christened this tool as the LibX Edition
Builder.

The LibX Edition Builder is meant to be used by librarians with rudimentary program-
ning knowledge and skills. Hence, we paid special attention to usability aspects to assist
librarians in configuring their edition correctly and in a short span of time. Extensibility and
maintainability of the code were required in order to facilitate quick deployment of changes and incorporation of user feedback. The following list of goals have helped us maximize usability and maintainability.

**Web Deployment.** The first development goal was to deploy the Edition Builder on the Web. Web applications avoid the need to download and install client-side applications and enable users worldwide to access our tool conveniently.

**Use of AJAX for Rich User Interaction.** The classical model of web applications is based on the use of pages and hyperlinks for navigation. This conventional page and hyperlink based model of web applications is not sufficient for representing the Edition Builder since the user may find it difficult to navigate through the hyperlinks, and not remember what settings she has correctly completed. Consequently, the user would not be able to create a complete and functional edition in a short duration of time. Moreover, each user interaction such as button clicks, mouse movements, and selection of an item in a list would cause a refresh of the entire page, possibly interrupting the user’s flow of thought.

Therefore, it was necessary to use an implementation technology that allows increased responsiveness, interactivity, and consequent user satisfaction. Such a technology is AJAX [Gar05]. AJAX allows us to create Rich Internet Applications (RIA) whose look and functionality mirrors that of desktop applications in terms of sophistication and response time.

**Auto-Discovery of Library Resources.** To create a complete edition, a user has to specify a number of settings correctly. Automatic configuration of settings mitigates the occurrence of errors users may unwittingly commit when specifying these settings. If the Edition Builder can discover as many settings automatically as possible, the customization
required on the part of the users will be minimized. This task is complicated because there is no complete reference database that contains a record of all the library resources an edition may need to include. The OCLC Worldcat Registry is a directory service that contains information about library resources of libraries that are OCLC members, but this information is insufficient to configure those resources in LibX. In addition, The Edition Builder also needed to automatically discover generic resources such as Google and Amazon, apart from OPACs.

**Use of Sound Software Engineering Techniques.** The developers of LibX continually improve the LibX toolbar. It was necessary to incorporate any changes quickly and easily in the Edition Builder. Agile development [HC01] provides the ability to modify the Edition Builder after incorporating user feedback and changed user requirements. Agile development emphasizes principles such as extensible software design, rapid prototyping, and frequent meetings between developers. During our development effort, we followed the principles of agile development to make the Edition Builder quickly adapt to change.

While developing user interfaces, frequent changes need to be made to the user interface implementation. Therefore, it was necessary to use an architecture that enables the separation of concerns between user interfaces, business logic, and the logic pertaining to the data layer and facilitates the deployment of frequent changes.

**Storage Scheme.** The Edition Builder stores configuration information for all revisions belonging to an edition. A *revision* is each change made to the configuration of an edition. The Edition Builder needed to offer a configuration management scheme to users for maintaining the editions and the revisions. We required a storage scheme that enables easy access to these configurations from the Edition Builder for the purposes of revision management.
The configuration files and image files need to be accessed not only by the Edition Builder, but also by other scripts external to the Edition Builder, such as a script to display a list of editions which have been created by users, a script to create the final toolbar as per the configuration file, and other external build tools.

**Evaluation.** A thorough usability evaluation was required to measure the effectiveness and efficiency of the Edition Builder with respect to creating LibX editions, the helpfulness of the user interface as well as the usage of the auto-detection and search features, based on the analysis of user interactions with the Edition Builder.

### 1.4 Results

This section summarizes the approaches we followed to realize the above goals.

**Edition Builder Architecture** Fig. 1.1 describes the architecture of the Edition Builder. There are 2 types of users - edition maintainers who create customized editions of LibX and end users who download and install LibX toolbars on the client side. The Edition Builder consists of the software code that we have developed or adapted to our use. The database and file system are both used to persist Edition Builder data. User information and meta-information about editions is stored in the database. Configuration files of revisions are stored in the file system. Third-party OPAC servers and the OCLC WorldCat Registry are contacted during the auto-discovery of resources.

Five different user interactions are shown in Fig. 1.1 namely, login, edition and revision
management, changing configuration files, auto-discovery, and downloads of LibX. On successful login, advanced features of edition and revision management are made available to users. These features enable edition maintainers to push updates to their users, finalize a revision, and roll back to a previously defunct revision. Maintainers can change the configurations for the revisions of their editions and these changes are immediately committed to the file system. Auto-discovery of resources enables users to readily import resources such as OPACs, OpenURL Resolvers and generic resources such as Google, Amazon, etc. The web server hosts the downloads of the LibX versions for IE and Firefox for the end users of the institution.

The following techniques were used for realizing the goals outlined in the previous section –
Use of AJAX for Rich User Interaction. We have used an AJAX framework called ZK to deliver the functionality in the Edition Builder. The Edition Builder component described in Fig. 1.1 contains the ZK layer and the application layer. The application layer consists of the logic we have added to maneuver and adapt the components provided by ZK. Using AJAX, we have successfully represented complex functionalities such as a version management and instant update of user interface based on server-side processing.

Auto-discovery. The Edition Builder examines the IP address of the user. When she is on campus, the Edition Builder initiates probes to discover the library resources at that address. The user can also choose to explicitly initiate auto-discovery by entering the IP address or the URL of the OPAC server. Alternatively, the user is allowed to search for the profile of her institution in the OCLC WorldCat Registry using the name of the institution. The Edition Builder offers the results from the auto-discovery probe to the user. OPACs imported via auto-discovery can be customized further by users. Finally, the Edition Builder allows users to manually select and configure an OPAC.

The Edition Builder contacts the servers that house the OPAC of the institution to obtain the OPAC web page. We have developed a set of heuristics to examine the OPAC web page and classify the detected OPAC. These heuristics are based on inspecting HTTP responses, structures of URLs, and keyword occurrence.

Use of Sound Software Engineering Techniques. We have maximized maintainability with the help of two techniques, agile development and adoption of a model-view-controller architecture.

We have used principles of agile development effectively with emphasis on short iterations,
delivery of a working prototype at the end of each such iteration, frequent interactions among the developers, and incorporation of user feedback. Agile development was facilitated by tools such as Castor, an XML data binding framework. Any change in the plugin entails a change in the XML schema of the configuration files, which consequently results in a change in the user interface of the Edition Builder. Castor’s automatic binding mechanisms allow the instantaneous translation of changes in the XML schema into changes in the object model. Thus, changes to the Edition Builder can be quickly deployed.

We have applied the model-view-controller architecture to enable separation of the presentation, application, and data layers in a way that promotes deploying changes independently to each layer, without affecting others.

Efficient Storage Scheme. The storage scheme described in Fig. 1.1 uses both the database and the file system. The file system is the main site of storage. The edition identifier maps to a relative path where configuration files of an edition’s revisions are stored in the file system. This scheme allows easy relocation of files and consequent ease of hosting. Moreover, the directory structure in this storage scheme follows the edition-revision hierarchy, which facilitates revision management. It also allows easy access to the configuration files by the Edition Builder as well as external scripts.

Evaluation. We have carried out a study based on the logs in which user interactions with the Edition Builder have been recorded. From this analysis we can observe that over the period of December 2007 to May 2008, the number of public editions configured using the Edition Builder has increased to 350 from 150. This result proves that users are increasingly adopting the LibX Edition Builder to customize their LibX editions. 50 percent of the editions were created in 72 minutes or less and 80 percent editions were created in 190
minutes or less. This result proves that users were able to create complex editions in a very short period of time. Thus, the LibX Edition Builder has proved to be effective and efficient in the configuration of LibX editions.

1.5 Contributions

The core contributions of this work are:

- Developed and deployed the LibX Edition Builder. We have used the ZK framework [Pot06b] to create the user interface as well as the application logic of the Edition Builder.

- Developed heuristics to aid in the auto-discovery of OPACs and other library resources.

- Applied sound software engineering techniques such as adoption of Agile Development techniques and usage of design patterns like Model-View-Controller to maximize maintainability.

- Performed a usability evaluation of the Edition Builder based on log analysis.

1.6 Document Overview

The remaining document describes the background about the technologies that we have used during development, the motivation behind the goals, the design and implementation details, the questions we sought to answer with the log analysis and the conclusions that we drew from the same.
Chapter 2

Background

During the course of the development of the LibX Edition Builder, we have used several technologies. This chapter provides background information on these technologies.

2.1 AJAX

AJAX stands for Asynchronous JavaScript and XML. AJAX is not a new programming language, but it is a unification of the following technologies: HTML and cascading style sheets, exchange of data using XML, the HTTP and HTTPS protocols, asynchronous communication between client and server, and JavaScript \[Gar05\].

2.1.1 Need for AJAX Frameworks

AJAX has revolutionized the navigation model of web applications. It prevents page refreshes at each user interaction by providing incremental update of pages. AJAX avoids the page based navigation model of classical applications. Instead, AJAX applications present
a single user interface page consisting of various components that are updated independently as per user interaction. Hence, reloading the entire page after each user interaction is avoided. Substantial improvements are possible in this model such as better responsiveness than conventional web applications, and consequent user satisfaction [MvD07b].

An AJAX application interposes an intermediate layer between the user and the server called the AJAX engine. Each user interaction results in a call to the AJAX engine. The AJAX engine decides whether a call requires action from the server. If the AJAX engine itself can service a call, it does so without resorting to communication with the server. The AJAX engine updates only the part of the page that has changed. Thus, due to the interposition of the AJAX engine, a user interacts with the application without interruptions. Fig. 2.1 [Gar05] explains the working of AJAX. When a user interacts with the application, an event is generated for the AJAX engine. The AJAX engine may forward it to the server for processing. But some updates to the user interface are realized by the AJAX engine, until the server-side processing completes. Thus, AJAX follows an asynchronous pattern while communicating with the server.

Programmers face the following challenges in ad-hoc AJAX development [Pot06a]:

1. Unstructured and consequently unmaintainable JavaScript code: If the JavaScript language is used in an ad-hoc manner, particularly by inexperienced programmers, unmaintainable code may result. Additionally, there is no IDE support for such ad-hoc AJAX programming.

2. Synchronization between the Clients and the Servers: The server, when receiving a new client side request, cannot be sure if the client has received and processed the previous server response because of the asynchronous nature in which the client processes re-
Figure 2.1: Asynchronous Communication in AJAX

sponses. Therefore, the server cannot know what the client’s state was when the new request was issued. Avoiding this scenario requires the application to contain a queuing mechanism that prevents the client from sending new requests until outstanding requests have been answered.

3. Browser Incompatibility: Different browsers have different implementations to render JavaScript. The developer is responsible for taking care of such incompatibilities in the application so that the application is rendered in a deterministic way in all browsers.
AJAX frameworks provide utility functions or libraries that solve these common problems. Writing an AJAX application from scratch wastes substantial amount of effort in developing and testing functionality that is already offered by many frameworks. There is a plethora of AJAX frameworks from which to choose.

### 2.1.2 Classification of AJAX Frameworks

[Hor06] classified AJAX frameworks along 2 different axes, Declarative vs. Imperative and Server-centric vs. Client-centric, as shown in Fig. 2.2.

![Axes for Comparing AJAX Frameworks](image)

Figure 2.2: Axes for Comparing AJAX Frameworks

The following characterization was the main guidance in our choice of frameworks.

**Declarative.** Declarative frameworks allow representing the user interface in a declarative language similar to HTML and XHTML. These frameworks offer components to create
the user interface. These components can be nested within each other. The event handlers pertaining to the user interface components are linked from within each component. Code Snippet 1 contains the code to display a login dialogbox using the XAL AJAX framework [Apa06]. Declarative frameworks know how to handle events that pass between UI widgets from the declaration provided by the application developer without need of programmatic binding. For example, in Code Snippet 1 the login-url needs to be followed on a click of the login button is declaratively described by the developer.

Code Snippet 1 Example of a Declarative Framework

```xml
<dialog>
  <horizontalBoxPane>
    <label text="Username"/>
    <textField id="username"/>
    <label text="Password"/>
    <textField id="password"/>
    <button text="login"
            onCommand="login-url"/>
  </horizontalBoxPane>
</dialog>
```

Imperative. Imperative frameworks provide a programming language model similar to that of conventional programming languages like C and C++. Developers write instructions that control objects, respond to events, and change state. Code Snippet 2 is an example of an imperative framework called Dojo. It shows sample code to handle the onClick event for a button in JavaScript. Code Snippet 2 also shows a JavaScript function pointer onServerResponse, that will be used when data requested from the server arrives.

Server-centric. When using server-centric AJAX frameworks the application developer needs to develop only the server side of the application. The framework has representatives on the client side and the server side. The server side representative of the framework
Code Snippet 2 Example of a Procedural Framework

```javascript
function helloPressed()
{
    dojo.io.bind({
        url: 'response.txt',
        handler: onServerResponse
    });
}

...<button dojoType="Button2" widgetId="helloButton">
    Hello World!
</button>
```

generates HTML and JavaScript from the server side code developed by the programmer of the application. The client side representative manipulates the browser DOM using the generated code.

Since the application state is maintained at the server, the application code need not send any application state from the client to the server. Therefore, server-centric frameworks hide the distributed nature of the application from the user and any communication between the client and the server occurs transparently from users. In addition, developers can use any programming language of their choice and employ libraries of their choice to create AJAX applications. Hence, server-centric frameworks avoid the problem of unstructured JavaScript code. Other examples of server-centric frameworks are Echo2 [Nex05] and ZK.

Client-centric. A client-centric framework provides JavaScript libraries to handle common problems in AJAX development such as browser compatibility. Client-centric frameworks also contain libraries that provide an RPC [Nel81] abstraction for communication between client and server.
These frameworks entail significant amount of JavaScript programming, which can still give rise to unmaintainable JavaScript code. Moreover, with client-centric AJAX frameworks, the programmer needs to take the responsibility of sending client state to the server whenever required. If the client fails to send state information to the server then the server may not be able to retain consistency of the application. Examples of client-centric AJAX frameworks are the Yahoo user interface library [Yah05] and Direct Web Remoting [Dir05]. Fig. 2.3 illustrates the distinction between client and server centric frameworks.

2.1.3 Our Choice – ZK

ZK falls in the server-centric AJAX framework category. It is a comprehensive framework that offers a complete solution for AJAX development. It is an amalgamation of the declar-
ative and procedural patterns – it uses the declarative paradigm to define the user interface and the procedural paradigm to describe the business logic.

ZK offers many rich components. It also provides a markup language called ZUML, which allows developing a rich client user interface without programming in JavaScript. ZK provides a programming model similar to that of desktop programming using Java Swing. Using ZK, developers can represent the application as components and manipulate these components based on events triggered by user interaction.

Fig. 2.4 explains the architecture of ZK, which consists of the ZK Loader, ZK AU Engine, and ZK Client Engine. The ZK Loader is responsible for loading ZUML pages and presenting them as HTML pages in response to HTTP requests. The ZK Client Engine runs at the browser and the ZK AU Engine at the server. The Client engine monitors user activity such as mouse movement and button clicks. Once detected, it notifies the AU Engine about the occurrence of an event. Upon receiving the request from Client Engine, the AU Engine updates the content of corresponding component, if necessary. The AU Engine signals the application by triggering various event handlers. If the application logic changes the content of components, the AU Engine sends the new content of altered components to the Client Engine. Then, the Client Engine updates the DOM tree with the new components that it has received from the AU Engine.

2.2 Model View Controller Design Pattern

Need for Model-View-Controller. There are common problems that have to be addressed while developing user interfaces [MH06]. If the user interface changes quite fre-
Figure 2.4: Architecture of ZK

...quently and if the code that deals with presentation is tightly coupled with the business logic, then a small change in the user interface will trigger multiple changes to the business logic and entail re-testing. Another problem is that, data might be represented in different parts of the user interface and if the data is updated, then all the parts of the user interface that depend on the data need to be updated as well. Isolating functional units from each other as much as possible helps the application developer to modify or extend a single unit without resorting to knowing the insides of other modules. Therefore, we required an architecture that facilitates separation of these concerns. An example of such an architecture is the Model-View-Controller architecture [KP88].

**Model-View-Controller.** The Model-View-Controller paradigm modularizes software to separate the presentation of the user interface, business logic, and data storage.
- **Model**: The Model contains mechanisms to access application specific data and the data storage system. It responds to queries regarding its state and it follows instructions to change state. The Model does not require knowledge of the GUI and how the data is visualized from the user’s perspective. The Model knows about the constraints that are imposed on changing the data and mechanisms by which to change and update the data. When the state of the Model changes, all the corresponding Views and Controllers are notified of this change.

- **View**: The View is the visualization of the Model state. It uses the query methods exposed by the Model to obtain and display the data or a part of it. The View must remain consistent with any changes in the Model.

- **Controller**: The Controller provides an interface to change the Model state. The Controller translates each interaction of the user with the View into instructions to be followed by the Model.

The user performs an action in the View (a button click for example). The responsible controller notifies the Model to change its behavior, according to the user’s action. The Model follows the instructions from the Controller and updates its state. The Views and Controllers are registered as dependents of the model. Any change in the Model is broadcast to all dependents. Based on this change, Views can update their display and Controllers can change their behavior. This interaction is shown in Fig. 2.5.

**Advantages of MVC.** Modularity, parallel development, and consistency are advantages of MVC.

- **Modularity**: Decoupling of user interface, application logic, and data domain logic allows developers to change each layer independently without causing a cascade of
Figure 2.5: Interaction between MVC

changes to other layers. Thus, MVC enforces maintainability and extensibility on the software.

- **Parallel Development**: Development of the three layers can progress in parallel. This parallelization reduces software development time.

- **Consistency**: MVC enables each change in the model data to be instantly reflected in the corresponding views and controllers. Developers have to just register the views and controllers as dependents on the model and the model will notify them automatically. In this way, consistency of the application is ensured without substantial effort from the developer.
2.3 XML Processing

SAX (Simple API for XML) and DOM (Document Object Model) are the two traditional methods used for XML processing. A new approach to XML processing is XML Data Binding, which is the process of converting XML entities to an object model representation. The following discussion explains the differences between these three approaches.

XML Processing - SAX. The SAX API is an event-driven approach to XML processing. It is based on streaming data as input, i.e., it reads an XML input source incrementally and sends the data to the application as and when the data is encountered. To use the SAX API, the developer has to register callbacks for handling the encountered XML constructs. The SAX API invokes these callbacks when it notices the occurrence of events related to these constructs such as starting of an element, ending of an element, and occurrence of character data.

The advantage of this incremental approach of parsing is that it provides better performance than any other XML processing software. The SAX API makes data available to the application as soon as it is encountered and does not wait for the entire document to be read. But, since the data is read in a streaming manner, the application does not have access to the children, parents, or siblings of an element while processing it. Therefore, an application using SAX parsing has to use appropriate data structures to keep track of the XML structure. This programming complexity makes this model prone to developer mistakes such as omission of data.

XML Processing - DOM. The DOM method of XML processing originated from the World Wide Web. This method uses an in-memory representation of the XML source in the
Table 2.1: Differences between SAX and DOM Approaches

<table>
<thead>
<tr>
<th></th>
<th>The SAX API</th>
<th>The DOM API</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speed</td>
<td><em>Fast.</em> Makes data available to application as soon as it is encountered.</td>
<td><em>Slow.</em> Makes data available only after entire document is parsed.</td>
</tr>
<tr>
<td>Memory Requirements</td>
<td><em>Less.</em> Only a part of the document needs to be stored in memory at a time.</td>
<td><em>More.</em> The entire DOM tree needs to be stored in memory at a time.</td>
</tr>
<tr>
<td>Coding Complexity</td>
<td><em>High.</em> Use of data structures to keep track of encountered XML structure.</td>
<td><em>Low.</em> Provides easy navigation methods to walk the DOM tree.</td>
</tr>
</tbody>
</table>

form of a data structure called a DOM tree. The advantage of the DOM method is that the DOM tree provides easy methods of navigation through the XML structure. The DOM tree provides access to an element’s parents, children, and siblings while that element is being processed and no other data structure is required to keep track of the XML structure, unlike in SAX. The disadvantage of this method is that the application processing can begin only after the DOM parser has completely parsed the document. As a result, the DOM parser is slower than SAX [Jan02]. Table 2.1 shows the differences between the SAX and DOM methods at a glance.

**Need for XML Data Binding.** The SAX API does not provide an object model representation of the XML document. The DOM method converts XML into generic objects such as Document, Element, and Attribute. To use an object model that reflects the application’s business domain, developers must manually convert these objects into the object model corresponding to the application domain [McL02].
Table 2.2: Differences between DOM and Data Binding Methods

<table>
<thead>
<tr>
<th></th>
<th>DOM</th>
<th>Data Binding</th>
</tr>
</thead>
</table>

Data binding addresses this problem. It automatically creates a mapping between XML data and classes that reflects the structure of the application domain and uses this mapping to convert an XML document into the object model. This is a data centric approach to XML binding. Since the conversion of XML to the object model and vice versa is automated, development effort is significantly reduced. This mapping functionality is offered by XML data binding frameworks. These frameworks may use SAX or DOM for their underlying implementation, but hide these details from the developers. Table 2.2 summarizes the differences between DOM and XML data binding [RU05].

**Our Choice – Castor.** Castor [TSG01] is a free, open source tool that binds object models to XML. It takes mapping information in the form of a XSD (XML Schema Definition) or a DTD (Document Type Definition) [McL02] to create the mapping between XML and the Java Object Model. Castor consists of a marshaling framework that converts Java objects to XML and an unmarshaling framework that converts XML to Java objects. Castor can marshal arbitrary objects into XML, a feature not offered by other XML binding frameworks.

For binding an XML schema to an object model the XSD needs to be handed to Castor which will create a set of objects to bind to any XML instance document that conforms to the schema involved. Castor contains a source code generator, which creates a set of Java
classes corresponding to the object model for an XML schema, as well as the necessary class descriptors used by the marshaling framework to obtain information about the generated classes.
Chapter 3

Design and Implementation of the Edition Builder

3.1 Goals

3.1.1 Web Based Deployment

Deploying the Edition Builder as a web application offers the following advantages:

1. Web applications can be accessed by any user with a computer and an Internet connection.

2. Web applications can be updated and maintained without installation of software on the client side. Hence, deployment of changes and updates can be done transparently to the user and in an expedited fashion.
3.1.2 Rich User Interaction

To facilitate the creation of customized LibX editions, the Edition Builder needed to have features such as a configuration management scheme to manage editions and revisions. We also required the ability to update the user interface with the results of server-side processing, the ability to make advanced features available only after the user has authenticated herself, and the ability to display the status of operations instantly. Moreover, multiple fields have to be specified in a configuration to completely customize each resource in the edition, as seen in Code Snippet 3. We believed that the Edition Builder needs to provide increased levels of interactivity to users to enable them to accomplish the above tasks efficiently. We thought that the interactivity provided by the navigation model of classic web applications would have been insufficient to achieve these tasks, because in this model each user interaction entails a waiting period for the user, during which the user is unable to proceed with the application [MvD07a]. Thus, we believed that following this model for the Edition Builder would mean increased configuration time and effort for the users.

AJAX (Asynchronous JavaScript and XML) avoids page refreshes after each user interaction [Gar05]. Many companies are using AJAX for their products, for example, Google Maps, Yahoo Mail, and Microsoft Office Online are all AJAX based. Seeing the success of these products and based on our research we concluded that AJAX can offer significant benefits with respect to usability and responsiveness [MvD07b, Pau05, SA07, Smi06, ZC07]. Therefore, we decided to deliver the rich functionality in the Edition Builder using AJAX.
Code Snippet 3 A Sample XML Configuration File

```xml
<edition version="1.2.4" id="vt">
    <name short="LibX VT" description="Toolbar for Virginia Tech Library users"
        adaptedby="" edition="Virginia Tech Edition" long="LibX Virginia Tech">
        <links>
            <url href="HTTP://www.lib.vt.edu/" label="VT University Libraries"/>
            <url href="HTTP://addison.vt.edu/" label="Addison Catalog"/>
        </links>
        <catalogs>
            <millenium name="Addison" url="HTTP://addison.vt.edu" options="Y;t;jt">
                <xisbn cues="true"/>
            </millenium>
        </catalogs>
        <openurl>
            <resolver pmidsid="" url="su8bj7jh4j.search.serialssolutions.com/"
                name="VT Article Linker"/>
        </openurl>
        <proxy>
            <ezproxy name="EZProxy" url="ezproxy.lib.vt.edu:8080/login?url=%S"/>
        </proxy>
        <options>
            <option key="rewritescholarpage" value="true"/>
            <option key="supportcoins" value="true"/>
            <option key="sersolisbnfix" value="true"/>
            <option key="autolink" value="true"/>
            <option key="icon" value="chrome://libx/skin/virginiatech.ico"/>
            <option key="logo" value="chrome://libx/skin/shield50.gif"/>
        </options>
        <searchoptions>
            <searchoption label="DB Title" value="dbname"/>
        </searchoptions>
        <additionalfiles>
            <file directory="chrome/libx/skin/libx" name="virginiatech.ico"/>
            <file directory="chrome/libx/skin/libx" name="shield50.gif"/>
        </additionalfiles>
    </edition>
```
3.1.3 Auto-Discovery of Library Resources

Users have to specify a number of settings to configure a single resource in an edition. For example, to completely configure an Ex Libris Aleph OPAC a user must specify the name of the server that houses the OPAC, the URL of the OPAC, indexes to be used along with various search terms such as author, ISSN, ISBN, keyword, title, find functions, and icons to be used as cues. We believed that users may commit mistakes while specifying all this information and if the Edition Builder were to automatically discover library resources relevant to the user, then the configuration effort required from the users will be minimized.

The idea of automatic discovery of library resource settings is analogous to the idea of Zero Configuration Networking (Zeroconf) that automatically detects the settings of an IP network to avoid manual configuration by lay persons [CS02].

The main difficulty in the automatic detection of library resources lies in the absence of a complete reference database of all resources. The OCLC WorldCat Registry [OCL] provides information such as the electronic locations of OPACs, institution IDs, and IP addresses. But, the OCLC WorldCat Registry can provide information only about OPACs and not other library resources. Moreover, the meta information provided by the OCLC WorldCat Registry contains only limited description about the resources. This information cannot be readily used in a LibX configuration. The OCLC provides 2 registries, the OCLC OpenURL Registry and the OCLC WorldCat Registry. OpenURL [Her01] is a type of URL that contains meta-data about library resources and is recognized as a standard. Libraries provide an online service to direct the end users to a copy of the resources described in the OpenURL. The OpenURL Registry provides a mapping from the IP address of an institution to the OpenURL configuration as well as the OCLC Id of the institution. In this case, the OpenURL configuration is complete and can be readily incorporated in the LibX.
configuration. The OCLC WorldCat Registry provides a mapping between the Institution Id to the OCLC profile of the institution. The OCLC profile of the institution contains the OPAC hostname of the institution. The OCLC WorldCat Registry also provides a mapping between the name of the institution to the institution profile. In this case, the complete configuration of the OPAC is not obtained from the OCLC registry. The registry provides only the OPAC hostname, but not information such as the OPAC vendor and the type of OPAC. Therefore, the OCLC registries could provide a starting point for automatic discovery, but we needed to augment this information with our techniques to provide complete configurations of resources to the users. In addition, not all institutions have registered themselves with the OCLC Consortium. Therefore, the goals of the Edition Builder with respect to auto-discovery of resources were:

1. Automatically discovering all the library resources required in the LibX configuration.

2. Auto-discovery support for as many institution libraries as possible.

3. Enabling incorporation of any searchable resource in the LibX configurations.

### 3.1.4 Use of Sound Software Engineering Techniques

From Code Snippet 3 it is evident that the configuration XML file is complex. The users of LibX request support for new options in the configuration continually from the developers of LibX. Therefore, the DTD of the configuration file is constantly changing. Hence, maintainability was a primary concern during the course of our development to preserve the quality of software even in the face of multiple changes.

Traditional software development methods seek to eliminate risks at the beginning of the
software development life cycle. This approach does not account for changed requirements. *Agile development* [HC01] stresses the following principles to be responsive to change: flexibility in software design so that deployment of changes is facilitated, frequent meetings between developers and customers, and emphasis on prototyping. To support agile development principles, we required a software architecture that is extensible and flexible to change. Therefore, we employed the *Model-View-Controller (MVC)* architecture to separate the presentation and application logic [KP88].

### 3.1.5 Storage Scheme

The data of the Edition Builder is in the form of XML files as shown in Code Snippet 3. Moreover, each edition contains image files uploaded by users. We required an efficient storage scheme to accomplish the following goals:

1. *Ease of integration with existing tools:* Configuration files and image files needed to be accessible not only by the edition builder, but also by the following tools:

   - *Edition list:* On the LibX website, we have displayed a list of editions currently maintained by users, as an indicator of the adoption of LibX and the Edition Builder. This list also provides a means for end users to download the installable packages of editions. In this list the logo of each edition, the description of the edition, and the date when the edition was made live are displayed. The existing PHP script that creates this web page needed access to the configuration files to generate the list.

   - *LibX build script:* A Perl script takes the configuration files and creates the downloadable install packages of LibX for Firefox and IE. This script also needs access to the configuration files.
2. *Ease of Revision Management*: A configuration management feature for revisions was required to enable users to manage revisions of an edition. A revision consists of changes and updates performed at different points to the edition configuration, on the way to completely configuring an edition. We needed to provide the following functionality in the revision management feature: finalizing a revision, rolling back to a previously defunct revision and offering the revision to the end users of libraries. Therefore, we needed a storage scheme to organize the configuration files, image files and LibX downloads to facilitate revision management.

3. *Ease of hosting*: We needed to store the downloadable installation packages of LibX for Firefox and IE (built from the configurations) in a way such that they can be offered using a standard web server configuration to the end users.

Conventionally, a relational database is used to persist data in web applications. To use a database approach for storing data in the Edition Builder would have required changes to the existing scripts enumerated above. Hence, we decided to avoid the database approach for storage, because we required compatibility with the existing scripts. Therefore, we used the file system to enable easy access by the Edition Builder as well as other scripts.

### 3.2 Edition Builder Functionality

This section illustrates the features of the Edition Builder and the expected user experience while interacting with these features. There are 2 types of users of our system – the edition maintainer and the end user. The edition maintainer is a librarian who creates the customized LibX version for their own institution using the LibX Edition Builder. The end user is the user of the institution’s library who downloads the LibX installations for Firefox and IE.
The end users need not interact with the Edition Builder.

### 3.2.1 Overview and General Features

The Edition Builder has a tabbed browsing interface to present the users with an overview of the sections that they have to complete to create a complete edition. Some of the tabs are made visible only after authentication to make advanced features available only to authorized users. An overview of the user interface is shown in Fig. 3.1.

![Figure 3.1: Overview of the Edition Builder Interface](image)

**Maintainer Management.** We allow maintainers to create editions without registering for an account. We believed that this feature will encourage first time users to explore the Edition Builder. The editions created without authentication cannot be retrieved. When a maintainer authenticates herself, she becomes the owner of any edition she creates (Fig. 3.2). The Edition Builder offers a feature to transfer ownership of editions to another user, which facilitates collaboration between multiple users to share the task of maintaining an edition.
Figure 3.2: User Login
Help Management. The maintainers must understand why they need to specify a setting and what purpose that setting will achieve in their finished edition. To that end, we provide help documentation for each setting. The help documentation is displayed as a tooltip that appears when the maintainer hovers over the question mark shown next to the item in question (Fig. 3.3). In this way, the maintainer refers to the help text only when she requires.

![Figure 3.3: Help Tooltips](image)

3.2.2 Edition Life Cycle Management

Edition Initiation. A maintainer is provided with 2 choices to initiate a new edition (Fig. 3.4 and Fig. 3.5). A maintainer can either begin her edition from scratch, or copy another edition’s settings, then proceed to make changes to it. We call these choices Start from scratch or Clone an edition. Cloning an edition is offered to enable institutions belonging to a consortium to reuse each other’s effort when creating editions that use similar resources.
Edition Management. The maintainer needs to specify information about different resources such as links, catalogs, proxies, and files to complete each revision of an edition as described in Section 3.2.3. The maintainer can declare the edition public, so that it can be included in the list of public editions and it can be cloned by other edition maintainers. She can also specify whether the edition is endorsed by the institution. She can also request help for the edition from the LibX team.

Revision Management. Each maintainer needs to maintain multiple revisions for each edition, to offer updates to users. There are three categories of revisions: Live, Test, and Archived. Their purpose is as follows:

1. Live - It is the most up-to-date revision of an edition, being offered to all the end users.

2. Test - The revision currently undergoing changes by the maintainer.
3. Archived - A revision which has been worked upon by the maintainer previously but which is currently defunct.

When an edition is initiated, a first revision is created in test mode. The maintainer configures this revision and later builds it. On building this revision, the LibX downloadable packages are created for Firefox and Internet Explorer. The user can test the configuration using a test page. This test page allows the user to test the functionality of the revision without actually installing the plug-in.

Once the maintainer is satisfied with the revision, the maintainer makes the revision live. Once the revision is live, no further changes to it are possible. The plug-ins of the live revision are offered for download to the entire community of users of the institution’s library.

If the maintainer decides to offer updates after making an edition live, she can copy the live revision forward to create a new test revision. The maintainer can make changes to this newest revision and declare it live, when she is satisfied with the changes. Subsequently, the previous live revision will be archived. Users can also copy an archived revision forward into the test mode to restore its settings. The user interface for the revision management scheme is visually depicted in Fig. 3.6.

**Committing Changes.** Users must be able to commit the changes they have made to their edition. We decided on automatically committing any change that a maintainer makes, a feature we call auto-save of editions. The benefit of this approach is that the maintainer does not have to worry about periodically saving her work as in conventional applications.
3.2.3 Configuring Resources

Edition Meta Information Management. In this section, the maintainer can provide a short name for the edition, description, and the name of the edition maintainer.

Shortcuts Management. In this section, the maintainer can provide shortcuts that are used in the completed toolbar. The user interface for configuring these links is shown in Fig. 3.7.
Catalogs And Databases Management. This section of the Edition Builder is used to incorporate catalogs and any other resources the library provides to its users, such as database lists, electronic journal lists, or any other searchable resource. These catalogs can be chosen manually or with auto-discovery. The user interface for configuring a sample catalog is shown in Fig. 3.8.

OpenURL Resolvers Management. In this section, the maintainers of the Edition Builder need to specify the OpenURL information for their library by using manual configuration or by importing auto-discovered settings. The user interface for configuring OpenURL Resolvers is shown in Fig. 3.9.
Figure 3.9: Configuring OpenURL Resolvers

Proxy Information Management. To help maintainers access their online resources of their library even when off campus, an edition needs to contain the settings for proxy servers. The interface for configuring these proxy servers is shown in Fig. 3.10.

Figure 3.10: Configuring Proxy

File Management. In this section, the maintainers can specify branding information by uploading icons, logos, and images of their institution. The interface for managing files is shown in Fig. 3.11.

Auto Discovery. The Edition Builder incorporates functionality to discover resources such as OPACs and OpenURL Resolvers. It displays the results of the auto-discovery process.
and enables the maintainers to select these auto-discovered settings. The maintainers can edit the list of current settings. The representation of auto-discovery features in the user interface is shown in Fig. 3.12.

### Community Sharing of Configured Catalogs.

We have created a shared catalog database that contains the configured catalogs of all live public editions. When a live edition is made public or vice versa, the Edition Builder adds the catalogs to the database. This database is utilized during auto-detection as explained in Section 3.4. This feature helps in sharing the effort used to configure OPACs.

### 3.3 Architecture of the Edition Builder

Fig. 3.13 shows the architecture of the Edition Builder. The Edition Builder consists of two layers: the ZK layer and the application layer, which has been added by us. The *Database*
stores user authentication information, meta information of the editions and the mapping between users and the editions maintained by each user. The File System is the main site for storing configuration files and image files of each revision.

Fig. 3.13 shows how these components interact with each other during five different types of user interactions, namely, login, edition and revision management, changes that the maintainer performs to the revisions, auto-discovery, and download of the LibX toolbar.

**Login.** When the maintainer sends a request for login, the server requests authentication information from the database.
Edition and Revision Management. If the login is successful, a request to display the editions and revisions for the maintainer is sent to the server. The mapping between the ‘s identity and the editions she maintains is stored in the database, but the revision information is derived from the file system. The controller fetches the identifiers for the editions from the database, translates these identifiers to the appropriate path in the file system, and probes this path. Each revision has a directory under this path, which houses image files and configurations.

Changes to Configurations. To load a revision, the model is instructed to obtain the corresponding configuration file from the file system. The model also unmarshals this XML
to Java objects. Each change that the maintainer makes to a revision is automatically marshaled to the appropriate configuration file on disk.

**Auto-Discovery.** Auto-discovery can be initiated automatically or by explicit user interaction. The server initiates probes in the shared catalog database and also the remote servers that house the resources relevant to the user. The Edition Builder uses the OCLC registries to obtain information about the auto-discovery process.

**Download of the LibX plug-ins.** End users of institution libraries access the file system via the standard web server configuration of Apache to download the LibX plug-ins for Firefox and IE.

### 3.4 Auto-Discovery

The Edition Builder helps the users in customizing their editions by auto-discovering resources such as catalogs and OpenURL Resolvers. The Edition Builder contacts the OpenURL Resolver Registry with the campus IP Address and the WorldCat Registry using the OCLC Institution Id. In addition, it contacts the OPAC servers directly and tries to learn the catalog type based on heuristics that have been developed by us.

The flow of information during the auto-detection process is described in Fig. 3.14. The maintainer has 3 choices to trigger auto-detection. First, if the maintainer is present on the campus of an OCLC recognized institution, the Edition Builder obtains the IP address of the maintainer and initiates the auto-discovery process automatically. Second, the maintainer can choose to enter an IP address or the OPAC URL and the auto-discovery process
is initiated as before. If the maintainer enters an IP address, then the OCLC OpenURL Registry is queried for OpenURL settings. If successful, the Edition Builder retrieves the OpenURL information and offers the OpenURL configuration to the maintainers for import. If need be, maintainers can make additional changes to the OpenURL so imported. As a by-product of this process, the OCLC institution profile is retrieved by the Edition Builder since the OpenURL registry also contains a mapping of the campus IP address to the institution profile. This profile is used to obtain the URL of the library OPAC host. The Edition Builder then feeds this URL directly to the auto-detection mechanism.

Third, the maintainer also can enter the name of the institution whose resources need to be detected, if she does not know the IP address or the OPAC host. The Edition Builder queries the OCLC WorldCat Registry, obtains the profile of the institution, and offers it for import. If the maintainer decides to import the profile, then the host name of the OPAC and the IP address of the institution are gleaned from this profile and the process continues as if the maintainer had entered the IP address or the OPAC host. If the cannot locate the OCLC entry for her institution, then she can choose to manually configure the OPAC.

The auto-discovery mechanism retrieves the web page housed at the OPAC URL. It parses this page and attempts to learn information from this page or from the URL itself. We have developed a set of heuristics to classify these URLs and pages into categories corresponding to known OPAC types. Examples of these heuristics are:

**Inspecting HTTP Response Header Fields.** When the OPAC server is contacted, the Edition Builder inspects the *Server* field of the HTTP response from the OPAC host. If it has a specific value that we have previously observed, then the Edition Builder detects the appropriate catalog and offers it for import to the user. For example, for the III Server of
the III Millennium OPAC, the Server field contains the value *III*, which is highly likely to come from a III Millennium OPAC.

**Searching for key phrases.** If the Edition Builder sees a particular pattern occurring in the URL or anywhere else in the page, we detect the appropriate catalog and offer it for import. For example, for the Voyager OPAC we search for a sequence `/cgi-bin/Pwebrecon.cgi` occurring in the URL or the page itself.

**Inspecting Forms.** A *Bookmarklet* is a URL template that can be used to import generic resources in a LibX configuration. The Edition Builder can offer any searchable resource as a Bookmarklet. To create a Bookmarklet URL template, we need to create a URL identical to the one obtained on submitting a form based searchable web page.
We parse the contents of the page and collect the fields of the form. We create a form data set that is a sequence of form field name and value pairs. For example, Code Snippet 4 depicts the HTML source of the web page located at google.com. The fields taken into consideration while creating the form data set are hidden fields, text fields, submit buttons, check boxes, radio buttons, images and buttons which also serve as submit buttons. The form fields in Code Snippet 4 are input fields and buttons. The Edition Builder then creates a Bookmarklet URL template containing the form data set created and offers a Bookmarklet for import. The rules for the creation of the URL template are dictated by the W3C standard [W3C98]. The bookmarklet representation of the Google.com web page is shown in Code Snippet 5. The %Y will be replaced by the search term of the end user.
3.5 Maximizing Maintainability using Castor and MVC

3.5.1 Agile Development using Castor

The complexity of the configuration file for each LibX revision is evident from Fig. 3. The developers of LibX are constantly incorporating new options in the configuration of LibX. We decided to adopt the agile development paradigm to deploy these changes quickly.

We adhered to the agile development paradigm by using short development cycles. At the end of each development cycle, we evaluated our progress and assigned goals for ourselves for the next phase of development.

Agile Development was also facilitated by software tools such as the XML Data Binding Framework, Castor. We use the XML Document Type Definition (XML DTD) to validate the configuration files and check their completeness. Any change in the requirements results in a change in the DTD and consequently entails a change in the user interface. To deploy these changes in the user interface we have used Castor. The DTD is taken by Castor and converted into an XML Schema. Castor then generates an in-memory representation of the XML DTD in the form of a Java Object Model.

Castor includes *Type Descriptors* in the code it generates. These type descriptors are used to tell the data types of the elements and attributes occurring in the XML DTD to the application using the Castor generated Java Object Model. We inspect these type descriptors using Java Reflection. We implemented different user interface components to enable maintainers to access and change the values of these different types. For example, a boolean attribute with true or false values is represented as a checkbox, an enumeration is represented as a
drop down list, and an attribute having string values is represented as a textbox. Thus, any changes in the LibX requirements are converted automatically into changes in the Edition Builder as described in Fig. 3.15.

Figure 3.15: Use of Castor for Quick Deployment of Changes
3.5.2 Application of MVC

We have used the Model-View-Controller architecture, which entails separation of concerns between user interfaces, application, and data storage logic. This architecture allows change to occur independently in one concern without causing cascading changes to other modules. This architecture facilitated deploying changes in a maintainable way.

Model. The format of data that serves as an input to the Edition Builder is an XML file, which stores the configuration of each revision for an edition. Any changes that the maintainer performs on a revision, will be translated into changes to that file. The model is implemented using a class Model.java. Each instance of the model represents an edition. The responsibility of the model class is to unmarshal the XML file into Java objects, marshal the Java objects back into XML, and other operations that involve interaction with the underlying file system. Any user interaction that results in a change to the configuration, triggers a message being sent to the model. The marshaling method of Castor is invoked by the model and the current state of the Java object model (after the change made by the user) is marshaled into the XML format by Castor. The Model also contains methods to store this marshaled XML at the appropriate location in the file system. Any changes to the XML DTD will thus require making changes only to Model.java leaving other code untouched.

View. The view is represented using the markup language offered by ZK, ZUML. Code Snippet 6 is an extract from a file written using ZUML. It describes an example of a view in the Edition Builder. This view describes the user interface of the tab that allows users to make changes to the Catalogs and Databases in the configuration. Hbox is a user interface component offered by ZK that lays out inner components in a horizontal fashion. We have
applied styles and other UI parameters to this component.

**Code Snippet 6** MVC Example

```html
<hbox id="catalogsTabController"
    use="org.libx.editionbuilder.CatalogsTabController"
    width="100%" widths="40%,60%" style="table-layout:fixed">
</hbox>
</box>
```

**Controller.** Controllers are Java classes that contain primary logic and act as the link between the model and the view. The view in Code Snippet 6 links to the controller for the user interface component. The controller is a Java class org.libx.editionbuilder.CatalogsTabController, which is specified by the use attribute. This controller class is responsible for logic such as addition, deletion of catalogs, and catalog configuration. We have decoupled the user interface layout from the controller class as much as possible. Even major layout changes do not require changes to the controller.

**Example of the interaction between the Model-View-Controller.** We illustrate the interaction between the model, view and controller using the example of user login. The user enters the authentication information and clicks the button *Log On*. This user interaction triggers the operation of the controller, UserInfo.java. The controller instructs the model to fetch the authentication information from the database, namely, the email and the password. The model uses JDBC methods to interact with the MySql database and returns the results to the controller. The controller then validates this information
and instructs the view to update itself. For example, if the password entered by the user is incorrect, the controller updates the View by adding user interface components such as labels that display the password incorrect message and the password reminder message. If the login is successful, the controller sends a message to the model to retrieve information about the editions and revisions from the file system. The controller then presents this information in the view.

3.6 Storage Scheme

3.6.1 Configuration Repository

A revision consists of a bundle of files such as configuration files and the image files uploaded by the user. We use a directory as a container to represent a revision. Since each edition may comprise multiple revisions, a parent directory is used to represent an edition and all the directories representing revisions are sub-directories of this larger directory.

The Edition Identifier is an 8 digit system-generated hexadecimal number. The file system path where the revisions are stored can be inferred from the Edition Identifier. For example, if the edition identifier is ABCD1234, all the revisions are housed in directories that have relative paths such as AB/CD/ABCD1234.1, AB/CD/ABCD1234.2, and so on. This scheme provides for easy relocation of the repository, because the edition id maps to a path in the file system relative to the root of the repository.

The organization of files in this edition revision hierarchy facilitated revision management. We have used file system operations to implement features of revision management. For example, restoring an archived revision is accomplished by copying the entire directory per-
taining to the old revision under a different name. Moreover, many users may access the Edition Builder simultaneously and attempt to create editions. This behavior would have created a race condition with respect to assigning unique identifiers to each edition. We have resolved this problem by utilizing the atomic create of files and directories provided by the operating system.

The downloadable install packages of LibX for Firefox and IE are also stored in the revision directory. Storing these downloads in the file system allows them to be offered to end users by a standard web server configuration of Apache. The live revision of an edition has a symbolic link pointing to it. Hence, the presence of a live revision is determined by the Edition Builder by inspecting whether the edition directory contains a symbolic link to it.

### 3.6.2 Database Design

In the Edition Builder, the revision management functionality needed a mapping of the users to the editions, so that users know for how many editions they are responsible immediately on authentication. This user-to-edition mapping information is not present in the XML configuration files. Therefore, we decided to maintain this mapping in the database. When a new edition is created by a maintainer, the new mapping is recorded in the database. The Edition Builder makes use of this mapping to provide the list of editions on authentication. The database is also responsible for maintaining information about the editions themselves that is not maintained in the configuration files, such as information that describes whether the edition has been made public and whether it has been endorsed by the respective institution. The shared catalog database contains the configured catalogs of all live and public editions to aid in the auto-detection of catalogs. The database also contains the login information of users.
The database contains the following four tables (Fig. 3.16):

1. **editionInfo** - Describes information about each edition. It has the following fields:
   
   (a) `editionId` – contains the system generated alpha numeric identifier for the edition.
   
   (b) `shortDesc` – contains a short description of the edition.
   
   (c) `isBlessed` – a flag that describes whether the edition is endorsed by the institution.
   
   (d) `isPublic` – a flag that describes whether the edition has been made public.

2. **userInfo** - Contains the authentication information for users.

   (a) `email` – email id of the user.

   (b) `password` – contains the password entered by the user.

3. **editionMaintainer** - Describes a mapping between editions and the users who maintain these editions. It contains the following fields –

   (a) `editionId` – Corresponds to the `editionId` field of the `editionInfo` table.

   (b) `email` – Corresponds to the `email` field of the `userInfo` table.

4. **catalogInfo** - Contains the catalogs of all live and public editions.

   (a) `url` – the URL at which the particular OPAC is found.

   (b) `name` – the name of the catalog.

   (c) `type` – the type of the catalog (example: Millennium, Sirsi).

   (d) `xml` – the XML representation of the catalog in the configuration files.

   (e) `editionId` – the identifier of the edition to which the catalog belongs.
3.7 Implementation Experiences

Castor. We used Castor to automatically convert from the XML schema to the Java object model and vice versa. We adapted the code generated by Castor to suit the needs of the Edition Builder. In the course of development, we discovered shortcomings in the code generated by Castor. We used the property change support provided by Castor generated code to manipulate the Model state based on client side events. But these property change events cannot be propagated to the parents, children, or the siblings of the XML element to which the property change listener is attached. We required this feature to implement the auto-save feature in the Edition Builder; whenever the maintainer effects a change in any object, the object model must be marshaled in a configuration XML file, which is persisted on the disk. Since property changes cannot be propagated up and down the object hierarchy, we had to attach listeners to each object in the object model manually, using reflection recursively, which was cumbersome.
In the Edition Builder, in case of defunct revisions or live revisions we do not allow the maintainer to persist changes, since the settings of those revisions are frozen. We implement this functionality with the help of a read-only model. The Edition Builder needed to reject property change events to prevent maintainers making changes to any read-only Model. But, this functionality to veto a property change was not provided by the code generated by Castor.

In addition, any enhancements that we implemented manually to address the above problems would have been lost when new code was generated by Castor in consequence to the changes in the configured DTD, resulting in wastage of programming effort. To automate the enhancement of generated code, Tilevich et al. [TB08] describe design patterns that can be used to express common program enhancements and can be automatically implemented using Aspect Oriented Programming. We have incorporated some of these enhancement patterns in the Edition Builder.

ZK. ZK is a server centric AJAX framework that allows programmers to develop server side code, which ZK then converts to JavaScript and DHTML. Thus, ZK shifts the responsibility of browser compatibility from the developer to itself as explained in Section 2.1.3. We found that ZK worked well for our purposes but we discovered that ZK’s claim of browser compatibility is not correct with respect to the rendering of user interface components. For example, ZK has a different definition of the width attribute from that of the HTML elements. ZK passes the width attribute to the HTML element while rendering it. Thus, ZK puts the responsibility of rendering components on the browser. Consequently, the results are different for different browsers. These bugs are difficult to solve by the application developer, since developers need to understand how ZK converts application code to JavaScript and HTML, an aspect normally hidden from the developer.
Chapter 4

Usability Evaluation

After developing and deploying the Edition Builder, we determined the usage characteristics of the Edition Builder to validate the claim that the Edition Builder has increased the efficiency and eased the process of customizing a LibX Edition. With this end in mind, we investigated the following questions:

1. How effective and efficient has the Edition Builder been in building LibX editions?

2. How intuitive and helpful has the user interface been?

3. How effective has auto-detection been in correctly configuring editions?

4. Has the adoption of the Edition Builder increased over time?

5. Has the community sharing of configured catalogs been beneficial to users?

Methodology of Analysis. During the course of our study, we logged the interactions of each user with the Edition Builder. These interactions included initiating an edition, making an edition live, importing a resource using auto-detection, and using the OCLC WorldCat
Registry search feature. We used the Tomcat web server logs to record these interactions.

Each statement in the log contains the date of the interaction, a session identifier, and the type of interaction of the user. To aid our analysis, we created an in-memory data structure to represent the entire log. We queried this data structure to obtain the required metrics. We also queried the user information, shared catalogs, and edition information databases to obtain data about the evolution of the adoption of the Edition Builder. We also inspected the configuration files of revisions to determine the complexity of editions created using the Edition Builder.

**Data Overview.** We examined the log files for a period of 5 months from December 7, 2007 to May 7, 2008, which is our *study period*. 562 distinct users actively used the Edition Builder in the study period. 330 public editions were built using the Edition Builder, 320 of these editions were made live. Public editions are completely configured editions marked public by their maintainers, so that they can be viewed universally. Live editions are finalized revisions that are being offered to the end-users of libraries for use. 1150 editions are being currently maintained by users and 2282 editions have been initiated using the Edition Builder. Of the initiated editions, only a few editions have been created by users who had not logged in. We believe that such users were merely experimenting with the Edition Builder. Therefore, we did not take into account their behavior for the purposes of the analysis, except while analyzing user behavior of initiating editions. The next sections describe the metrics that we evaluated.
4.1 Effectiveness and Efficiency

4.1.1 Cumulative Distribution of Time Taken to Make Editions Live

We hoped that the Edition Builder would greatly speed up the process of creating customized LibX editions. We calculated the cumulative time taken by users to create their first live edition (Fig. 4.1). The amount of time it took to make an edition live also includes time taken to test it. The results show that 50% of editions were built in 72 minutes or less and 80% were created in 190 minutes or less. These results prove that the Edition Builder is efficient in enabling users to create editions in a short time.

4.1.2 Complexity of Configurations

We gathered summary statistics about the configurations users created in live editions. These statistics revealed the complexity of each revision and consequently the Edition Builder’s effectiveness in allowing users to create complex configurations. The results of this analysis are shown in Fig. 4.2, Fig. 4.3, Fig. 4.4, and Fig. 4.5.

In the Edition Builder, we provide by default only one shortcut in the shortcuts tab (Fig. 3.7). From Fig. 4.2 we can see that, for 60% of the public editions, maintainers changed the default configuration of links, that is, maintainers added links pertaining to their own institution. Hence, we can conclude that the user interface provided by the Edition Builder to configure links was intuitive to the users.

From Fig. 4.3 we can see that 45% of editions contained only one catalog. This result is expected, since most libraries have a single OPAC. Only about 3% failed to add catalogs to
their editions. 52% of editions include more than 2 catalogs. Libraries with more than a single catalog in their configurations are often consortial libraries, including multiple branch libraries, and also libraries looking to expand access to information by including nearby libraries in their edition. These results prove that the Edition Builder enables users to create functional editions and even complex ones that include multiple catalogs.
Even though the Edition Builder does not support auto-detection of proxies, Fig. 4.4 shows that 45% of editions contain proxy configurations. This relatively low number is not surprising, because many libraries do not need proxy access as they are public libraries. In addition, other libraries support access by different methods such as Virtual Private Networks, which do not need to be configured. This result proves that the user interface of the Edition Builder has been successful in allowing users to incorporate proxies in their edition.

Every edition contains 4 files by default. 2 of these have been added by us (configuration file and default preferences file) and the remaining 2 are image files to be used as icons. We provide default images in each edition, but users can upload images of their choice in the File Management tab in the user interface. Fig. 4.5 shows that 53% of editions contain 4 files. Of these 53%, about 51% replaced the default image files with their own logos. About 28% of editions contain 5 files. Of these 28%, 65% replaced the default image files with their own images. The fifth file is the icon of the OpenURL resolver or OPAC added during auto-detection. These results prove that the Edition Builder has enabled users to conveniently upload their own logos and icons to the edition and thus add branding information to the configuration easily.

4.2 Intuitiveness of the User Interface

4.2.1 Number of Live Editions Maintained Per User

The number of live editions maintained by a single user can be used to derive insights about the user’s understanding of the revision management scheme. More live editions per user may mean that the maintainer failed to understand the functionalities of revision management, which would allow customization and adaptation within one edition, rather than creating
multiple, different editions. In turn, a larger number of live editions per user may also mean that the maintainer created different editions to support different communities of end users. Fig. 4.6 shows the results of this analysis. The results show that 46% of users have one or more live editions. Further analysis of the configuration files showed that many of these users configured multiple editions to support different institutions belonging to the same
4.2.2 Usage of the Help Tooltips

Help tooltips are provided beside each setting to assist maintainers in the configuration of that setting. The display of these tooltips is recorded when the user hovers over them. A
consistent usage of the help tooltips demonstrates that users are turning to the help tooltips in order to configure their editions correctly. But excessive usage of these help blurbs could also point to features that appear to be confusing and in need of improvement.

The results of this analysis show that help tooltips for some features are being used excessively. Some aspects of the user interface are confusing to users, for example the *Build button*
Figure 4.5: Number of Configured Files

(Build button is used to create the downloads of LibX for Firefox and IE) help tooltip was read over 300 times over the study period. The functionality of the Build Button may need to be presented in some other manner that would be more intuitive to the users.

We discovered that not only the tooltips of complex features have high usage counts but also the help tooltips for some features that offer simple functionality like *Deleting an Edition* and
Transferring Ownership to Another User. Many users asked for help about these features in the emails that we received. Thus, it is required that these simple features be provided in a manner that is clearer than the current one. We did expect that the help tooltips related to Revision Management and Configuring Bookmarklet URLs would be read frequently since these tooltips illustrate concepts new to many maintainers. In these cases, we believe that
reading the help tooltips is beneficial, even necessary to them.

4.2.3 Choice of Edition Initiation

We offer 2 choices to the user to initiate editions: one option is to create editions from scratch and another is to clone existing editions. We observed that 70% of users prefer to initiate their edition from scratch, rather than clone the settings of an existing edition. We offer the clone edition feature so that libraries belonging to the same consortium can reuse each others’ settings to reduce configuration time. We believe that maintainers may not be able to discern which institutions are similar to their own, and thus were unable to make use of this feature. For libraries not belonging to a consortium, usage of this feature is usually not necessary.

4.3 Usage of Auto-Detection Features

4.3.1 Usage of the Auto-discovery Feature Per User

We expected that the auto-discovery of resources will assist users in configuring their editions correctly and in a short span of time. Fig. 4.7 shows the number of times each user used the auto-detection features successfully to import a catalog. These results show that 50% of users used the auto-detection features 1 or more times. Only about 1% of users have used the auto-detection features exceedingly, which may indicate that these users found it difficult to understand the user interface for auto-detecting or configuring catalogs. Fig. 4.8 shows the number of times each user used the auto-detection features to import an OpenURL Resolver. About 28% of users used this feature 1 or more times in the duration of the study.
The Edition Builder also offers a feature for searching institutions’ profiles in the OCLC WorldCat Registry, and then using this profile in the auto-detection of resources. The usage characteristics of this feature helped in understanding its awareness among users. Fig. 4.9 shows the per user usage of this feature. 31% of users used this feature 1 or more times. We have to further investigate why more users did not choose to use the OpenURL detection and WorldCat Registry Search features more often.

Figure 4.7: Autodetection of Catalogs - Per User Usage
4.3.2 Categorization of Auto-detected Resources by Type

As explained in Section 3.4, we used heuristics for the automatic discovery of resources. We used different sets of heuristics to detect catalogs from different vendors. The Edition Builder can automatically detect OPAC systems such as III Millennium, Dynix Horizon/IPAC, Ex Libris’s Aleph, Sirsi, Sirsi Web2, Endeca, and Voyager. OPACs that are not supported
directly and generic web sites such as Google and Amazon can be added as Bookmarklets. To know the performance of our heuristics in general and with respect to specific catalogs, we analyzed the type-wise division of the total number of resources detected. This categorization is shown in Fig. 4.10.

Fig. 4.10 shows that 48% of catalogs detected are Bookmarklets. The Edition Builder can
offer any webpage containing an HTML form as a Bookmarklet, hence the majority of catalogs imported are Bookmarklets. 23% of detected catalogs are of the type III Millennium. The heuristics for detecting this type of catalog are robust in nature. This large number may be explained by a concentration of advertisement of LibX in the III user community, since III Millennium is the OPAC of the Virginia Tech library. 16% of detected catalogs are Voyagers. The heuristics we developed for detecting Voyager catalogs are prone to false positives, which may be a reason for the preponderance of detected Voyager catalogs.

To examine if the Edition Builder is used equally by libraries subscribing to different vendors, we compared their market share to the frequency with which they were included in our auto detection. III Millennium catalogs constitute about 13% of the market share of installed OPAC systems, Voyager catalogs constitute about 12%, Ex Libris Aleph catalog constitute 20% and Sirsi Dynix’s Horizon catalogs constitute 16% [Bre07]. These market share statistics do not currently match the distribution of the detected catalogs, since in the distribution of auto-detected catalogs there is a clear majority of catalogs such as Millennium and Voyager, but in terms of market share Aleph catalogs are more dominant.

The Edition Builder maintains a shared catalogs database that contains the configured catalogs of all public and live editions to assist in the auto-discovery of catalogs. In the shared catalogs database maintained by us there were 38% Bookmarklets, 24% Millenium, 11% Voyager, 7% Aleph, 3% Horizon, and 5% Sirsi at the end of the study period. These findings closely correspond to the distribution of heuristically detected catalogs. A possible explanation of the distribution of the heuristically detected catalogs is that the users of the Edition Builder are early adopters and in due course of time, this distribution may resemble the market-share of OPACs.
4.4 Adoption of the Edition Builder

A librarian offers a LibX edition for her entire community after making it live. The development of the total number of live, publicly visible editions over time gives us an insight into the evolution of the Edition Builder’s user base over time.

We hypothesized that this number will only increase over time and such a result will demon-
strate that librarians are increasingly adopting the Edition Builder to build LibX editions for their libraries. The development of the number of all editions live and in testing would also give additional support to above hypothesis. The results of our analysis are shown in Fig. 4.11, Fig. 4.12, and Fig. 4.13.

Fig. 4.11 shows that the number of public editions has increased by 40%. Fig. 4.12 shows that the number of total editions has increased by 92%. Fig. 4.13 shows an increase of 99% in the number of registered users. These results prove that the adoption of the Edition Builder is steadily increasing with time at a rate of about 20 editions per month.

4.5 Community Sharing of Configured Catalogs

As users make their editions public, the Edition Builder adds all the catalogs incorporated in each such edition into a database. This database also aids in the process of auto-discovery and allows the re-use of effort. During auto-detection of catalogs the Edition Builder consults this database to detect catalogs and offers results from this probe as well as from the heuristics based detection to the user. The user can choose to import catalogs detected by either method. We have determined that 29% of imported catalogs came from the database.

The development of the number of configured catalogs added to the shared catalogs database over time is shown in Fig. 4.14. We can see that the number of catalogs in the shared catalogs database has increased by 40%, with 756 configured catalogs present in the database at the end of the study period. Hence, the likelihood of a user finding a catalog resembling the catalogs of her institution library has increased.
4.6 Conclusion

The results of our study validate our hypothesis that the Edition Builder has been effective in assisting users in creating customized LibX editions efficiently. The features in which we invested significant development effort, such as auto-detection, are being used frequently and effectively by users to configure their editions. The data also shows that the user interface
Figure 4.12: Number of All editions Over Time

of the Edition Builder is intuitive and enables users to create complex editions.

The following aspects have to be investigated further. Since a small number of users have used the auto-detection features repeatedly, we plan to determine if these users had difficulty in comprehending the mechanism of auto-discovery or in configuring resources. From the log-analysis and the emails that we received from users, we discovered that some aspects of
the user interface are not clear and ought to be changed in order to be more intuitive to the users. Moreover, a majority of users has not used the OpenURL Resolver Auto-detection and WorldCat Registry Search features. We need to determine if these features need to be placed more prominently or whether the help documentation accompanying these features is insufficient.
Figure 4.14: Development of Shared Catalogs Over Time

Number of shared catalogs since Dec 07, 2007

Days from Dec 07, 2007 to May 7, 2008

Total Catalogs
756
Chapter 5

Related Work

5.1 Toolbar Configuration Web Interface - Conduit

Conduit\cite{Con08} is a web interface that enables the creation of community toolbars. The purpose of Conduit is to create toolbars that will create publicity and fetch more traffic for web sites. The Conduit interface is created using the classical navigational model of web applications, hence to change any information previously specified users have to move back and forth between pages. This problem is not a substantial hindrance to users since the functionality provided by Conduit toolbars is simplistic and thus does not require sophisticated configuration. Conduit offers limited support for customization. It allows users to select from amongst a limited number of icons and enables searches only from Google. Thus Conduit differs widely from the Edition Builder since the functionality encompassed by the LibX Edition Builder is involved and geared towards the library domain.
5.2 Usability Evaluation Methods

5.2.1 Log Analysis Methods

Analysis of web server logs has been employed in various studies to identify usage patterns of web applications, but different methods of logging and analysis have been used in each case. Data mining methods have been used by Spiliopoulou [Spi00] to analyze logs. These methods were applied to evaluate online shopping sites. As future work, we consider using data mining approaches to analyze the logs that we have maintained.

An approach described by Paganelli et al [PP02] creates a task model for a web site and uses a tool (WebRemUSINE) to examine the log data in conjunction with the task model. A task model is a classification of user interactions into high-level tasks that can be performed using the web site. If the user behavior does not comply with the task model, then an error is said to have occurred. The advantage of this approach is that it enables partial automation of usability evaluation techniques. We regard integration of automated tools and task models into our analysis as future work.

A client-side logging approach uses an HTTP proxy to modify HTML pages by introducing JavaScript before serving the page on the client-side. This JavaScript then tracks user interaction and logs it. This approach is useful for evaluating AJAX applications that contact the server only to perform data operations [AWS06, AS07]. Since this solution is deployed on the client-side, lower level data such as user scrolling and typing behavior can be recorded, which can provide deeper insights into user behavior.

The analysis based on logs that we have carried out is different from the above described
approaches. The intention of the log analysis was to answer high-level questions that relate to the usability of the Edition Builder. We answered questions pertaining to the efficiency of the Edition Builder in customizing LibX editions, the usage of features like auto-discovery, and the adoption of the Edition Builder over time. We did not focus on identifying specific features that hinder users. We also did not require recording low-level user interactions like mouse movements and scrolling, since we did not use client-side JavaScript to develop the Edition Builder. We used a server-centric AJAX framework in which each client request is serviced by the server.

5.2.2 Questionnaire Methods

Zaharias used a questionnaire-based method to evaluate the e-learning software [Zah06]. He described the stages in the formulation of the survey and also the parameters measured using this method. Lewis evaluated the use of After Scenario Questionnaires (ASQs) [Lew91] for usability evaluation. Ssemugabi et al describe a method called Heuristic Evaluation (HE), which is defined as evaluation by domain experts [SdV07]. They also compared HE to survey based usability evaluation techniques and made the observation that results from both methods closely correspond to each other, but in their study they found HE to be effective in identifying problems. Piller et al argue that even if objective usability evaluations such as log analysis have given positive results with respect to the usability of the software, subjective evaluations may still turn out to be negative [PM01]. We regard a survey based evaluation of the Edition Builder to be future work.
Chapter 6

Conclusion and Future Work

The LibX Edition Builder is being used currently by around 750 users to create customized LibX editions for their institutions. Our work makes the following contributions:

1. We provided rich user interaction by developing the Edition Builder using AJAX. As a result, we were able to represent complex functionalities such as configuration management and auto-detection in a manner intuitive to users. Users are able to make use of the Edition Builder to create complex customized editions.

2. The Edition Builder automatically discovers library resources such as OPACs and OpenURL Resolvers using a set of heuristics. The Edition Builder is able to offer any web based search interface as a Bookmarklet. This automatic configuration of resources reduces the time and effort required by a user to configure an edition.

3. The Edition Builder was developed using agile methods. Hence, changed user requirements can be quickly incorporated in the Edition Builder. We have used the Model-View-Controller design pattern to create an extensible framework to facilitate the deployment of changes.
4. We recorded key interactions of the user with the Edition Builder in the web server logs. We processed these logs to evaluate the usability of the Edition Builder. The results obtained from this evaluation indicate that the Edition Builder has been effective and efficient in assisting the creation of complex LibX Editions. The adoption of the Edition Builder has increased at a high rate.
Bibliography


