An Aggregator Tool for Extraction and Collection of Data from Web Pages

by

Alexander Malchik

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of Master of Engineering in Electrical Engineering and Computer Science at the MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2000

© Alexander Malchik, MM. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part.

Author

Department of Electrical Engineering and Computer Science

May 22, 2000

Certified by

Michael D. Siegel
Principal Research Scientist
Thesis Supervisor

Stuart E. Madnick
Professor
Thesis Supervisor

Accepted by

Arthur C. Smith
Chairman, Department Committee on Graduate Students
An Aggregator Tool for Extraction and Collection of Data from Web Pages

by
Alexander Malchik

Submitted to the Department of Electrical Engineering and Computer Science on May 22, 2000, in partial fulfillment of the requirements for the degree of Master of Engineering in Electrical Engineering and Computer Science

Abstract

In this thesis we describe the Aggregator Tool, an application that combines data extraction from the web sources with aggregation services to allow users to create flexible personalized information aggregations. This tool can be used for a variety of applications such as building customized “portals” for tracking market information and news, monitoring an aggregation of prices for a particular item of interest from various on-line retailers, or managing personal finance portfolio, such as bank and investment statements that are available online, conveniently from one set of web pages. The Aggregator Tool provides an easy-to-use graphical interface that requires no programming skills from the users. Once an aggregation application is built, it can be accessed from any browser. Users only need to remember their Aggregator Tool login information to view the aggregated data that is extracted instantaneously from multiple web sources.

The Aggregator Tool uses a client-server model. The client is implemented as a Java Applet. The modular design of the tool means individual components such as web wrapping heuristics or different post-aggregation templates can be easily extended or added in the future.

Thesis Supervisor: Michael D. Siegel
Title: Principal Research Scientist

Thesis Supervisor: Stuart E. Madnick
Title: Professor
Acknowledgments

First and foremost, I would like to thank Michael Siegel and Stuart Madnick for making this thesis possible in the first place. I am grateful for their support and guidance, and especially for their enthusiasm about the project and the Context Interchange group. I wish this group the best of luck in the coming years!

I would also like to thank Aykut for all his help, his abundance of ideas, and for the countless discussions of everything from this project to life that kept this office so much more fun to work in – Saurabh and Felicia, your contributions are also much appreciated.

Of course none of this would have been possible without my family and my friends, so thank you Mom, Dad, and grandma. Sasha, thank you for keeping me such a good company on AIM.

Finally, I would like to thank my girlfriend Sveta for her love, companionship, and help in so many things in our years here at MIT.
4.4.3 Pattern Generation

5 Conclusions

6 Future Work
List of Figures

1. The Architecture Diagram of the Aggregator Tool 10
2. Project Browser Interface 24
3. Interaction of Project Browser Elements 25
4. Registry Database Schema 30
5. Sample Registry Content 32
6. Web Wrapper Interface 36
7. Sample Spec File Listing 39
8. Parsed HTML Document Tree Display 44
9. Example JSP code 46
10. Aggregation Example 47
11. Web Sources for the Aggregation 47
1. Introduction

The World Wide Web is becoming the standard environment for information publishing. The number of information sources on the web is growing at a rapid pace. Likewise, more and more products are offered for sale online. What combines these two seemingly different Internet offerings is that they both can be aggregated. Aggregators provide a range of services, from being a mere user convenience when used to collect news or financial information from various sources, to making a large impact on manufacturers’ and resellers’ business models as shopping and price-comparison agents. Using the Internet and recent technologies, aggregators are emerging as an important new internet business sector. As noted by Pan et al [1], aggregators will soon emerge in the industries where they were not formerly present. In this thesis, we describe the Aggregator Tool, an application that combines web wrapping with aggregation services to allow users to create flexible personalized aggregations.

We call a collection of data extracted in real time from various web sources, personalized and displayed for a particular user an aggregation. Examples of such aggregations include a collection of pricing information on a particular product the user is interested in over several online retailers, or an aggregation of the user’s financial information from various online account access sites. An aggregator is an entity that transparently collects information from multiple sources, integrates it, and presents it to the user. Aggregators for many different markets are rapidly emerging on the Internet. The explosion of the aggregation market parallels the expansion of the Internet, because the wealth of information made available on the World Wide Web nourishes aggregators, and the apparent ease of obtaining that information encourages constant new aggregator development. Practically every bit of information made available on the web can be collected without the information provider’s knowledge. This clandestine power makes aggregators a force to be reckoned with in the future marketplace, and enables them to influence many online business strategies.
1.1 Motivation for the Aggregator Tool Project

Currently, new aggregators have to be made specifically for each kind of site being aggregated. This is a hard task that does not scale well with the rapidly growing and changing nature of the Internet. The Aggregator Tool makes the task of building custom-tailored aggregators easy and automatic.

The Aggregator Tool research is inspired by the observation of the growing importance of Internet aggregators, and the desire to do away with the limitations of the currently available ones. The Aggregator Tool attempts to solve the inflexibility problems of existing aggregators by providing users with the power to build any kind of aggregator they need. Users can create aggregator projects by defining through a simple point-and-click interface precisely which sites and what information from each site they would like to be included into their project. Once created, an aggregator project should be shareable with other users, and accessible from any web browser. The latest available data should be extracted without prompting the user for individual site user IDs and passwords.

The Aggregator Tool project emerged as part of the COntext INterchange (COIN) project at the Sloan School of Management. COIN [2] seeks to solve the problem of accessing multiple heterogeneous data sources by providing a unified view to them. All data sources in COIN are viewed as relational databases. The COIN model defines a mediation approach for integrating disparate data sources. For example, in the financial domain, data sources may use different currencies, scales, monetary or date formats, or exchange rates. By assigning contexts to each data source and receiver, COIN can automatically mediate between them to achieve a uniform, correct view of the data.

In the last decade, the number of data sources available to both corporations and end-users has increased dramatically. While in the past data sources included traditional databases and dedicated data feeds, currently the World Wide Web has taken a commanding lead as the data source both end-users and corporations increasingly rely on. However, the multitude of sources on the web and the variety of formats they employ to present data pose
significant challenges to the attempts of aggregators and mediators to collect the data and present it in a homogeneous way.

In order to represent semi-structured data from the web sources in a structured, SQL-accessible way, Web Wrappers are needed. The COIN group has developed a generic wrapper technology for web data sources that captures the hypertext structure of web documents and extracts data, yet remains flexible. Each web source is modeled by a Specification File (spec file), which describes the data of interest available at the web site, and how that data can be obtained. Caméléon [3], a web extraction engine, uses spec files to present the extracted data as a relational database table.

The Aggregator Tool, in turn, provides an intuitive way to create specification files and combine them into user-defined aggregation projects. Merged with context mediation technologies and used with multiple distributed data sources such as databases and web sites, the aggregators present a powerful solution to effectively managing and presenting a wide variety of data.

1.2 Mile-High View of the Aggregator Tool

The Aggregator Tool follows the client-server model. On the client side, it consists of the user interface implemented as a Java applet. The applet communicates with the server, implemented as a Java servlet, and a registry database which stores information about all the users of the aggregator tool and their custom aggregation projects. The servlet is responsible for reading and storing two kinds of user files: spec files and project files. Each spec file is a combination of the schema information and the extraction rules for a class of web pages. Each project file is a JavaServer Page file that contains queries to Caméléon web extraction engine, and calls to Java Bean components that perform post-aggregation data manipulations. The architecture diagram of the Aggregator Tool is presented in Figure 1.
Figure 1. The Architecture Diagram of the Aggregator Tool.
The applet allows users to browse the websites to be wrapped. With simple point-and-click operations, users can select the particular information to be wrapped, and the aggregator tool creates spec files that define extraction rules for the data. Users can then combine spec files into projects to display and manipulate extracted data as an aggregation. Once created, projects can be accessed directly from any browser. Users can also immediately view the aggregation results in the Aggregator Tool applet.

1.3 Organization of this Thesis

In the next chapter, we present an overview of the related research and technologies, followed by the design decisions and technology choices for the Aggregator Tool (Chapter 3), and the description of the implementation (Chapter 4). Finally, we discuss conclusions and future work.
2. Overview of the Related Work

2.1 A Look at the Existing Aggregators

The aggregators that emerged on the Internet so far are highly specialized. These include shopping bots that collect pricing information for specific products from different retailers on the web, and information management services that aggregate users’ financial transactions at various sites. Additionally, many informational portal sites offer personal customizations to attract the users, usually named by pre-pending “my” to the site name. While these portals do not strictly qualify as aggregators, they deserve some discussion here. Let’s take a closer look at the three representative examples.

2.1.1 evenbetter.com

evenbetter.com, former dealpilot, earlier known as acses.com (http://www.evenbetter.com), is a price aggregation service for online book, music, and video sellers. Currently, evenbetter database includes about forty online bookstores, ten music, and ten video stores. According to the site, evenbetter “developed a unique Agent Technology, which gains useful information from rough data, called Internet Data Compiling (IDC).” Upon indicating his or her shipping location and entering a keyword search query, the user is presented with a comprehensive table listing the item price, shipping costs, sales tax if applicable, shipping time and service, and the total resulting price. The site converts all prices into the user’s local currency using current rates, updated daily.

evenbetter.com, a member of ComparisonShopping.Net, a network of Internet based comparison shopping services, is a well-known and established bookstore price comparison site. While its selection of bookstores is relatively large and includes many international retailers, the stores have agreed to participate in price aggregation through evenbetter.com’s partnership program. Users have no way of including additional, not featured stores in their search. Major specialized retailers, such as textbook oriented varsitybooks.com and
textbooks.com, are not presented on evenbetter.com. Selection of music stores is even scarcer, with such major sellers as buymusic.com not present.

2.1.2 Yodlee

Yodlee, an account customization and consolidation service, http://yodlee.com, offers users centralized access to hundreds of web sites in the finance, news, travel, shopping, and web-based email categories. Yodlee is specializing in providing customers a single point of access to many kinds of online statements, as well as some news and sports headlines. Yodlee stores user’s login information for each account, periodically extracts relevant information from the sites, and organizes it on a single page for the user. The order of categories displayed is customizable. Yodlee provides an additional convenient benefit of allowing users to immediately log on to any participating sites without resubmitting user ID and password.

Yodlee is a great tool for pooling together many logins and passwords, viewing all transactional information in one place, and getting quick one-glance updates. However, the online sources Yodlee can handle are once again limited to the predefined ones, and many account holders, for instance, at smaller banks—or even some new online banks—will find themselves out of luck. Moreover, the service extracts the entire, sometimes multiple pages long, transaction records for some accounts, and only balances for some others. There is no way for the user to specify exactly what kind of information he or she is interested in retrieving from the data sources.

2.1.3 My.Yahoo

my.yahoo is a typical personalizeable information portal. Once signed in, the user can add predefined informational “modules” to the left (narrow) or right (wide) side of the page, in any order he or she prefers. The contents of some modules—such as stock prices portfolio, weather conditions, sports scores and TV listings—can be modified to include only
the stickers or geographical places the user is interested in. Other modules cannot be modified, and the contents within any modules cannot be rearranged.

While my.yahoo is a very useful tool for collecting content from various parts of vast yahoo site in one convenient place for easy access, it does not allow information from any other sources to be incorporated into the same page. A tool that allows particular information from yahoo – or already customized my.yahoo pages – to be integrated into an aggregation of other information from various sources, such as user’s personal financial data, would be very handy and complimentary to my.yahoo service.

2.2 Web Wrappers: Overview of the Related Work

Let’s turn our attention to web wrapping, an integral part of the Aggregator Tool, and review the related work done in this area.

Currently, web sources present information in an unstructured or semi-structured way, since HTML is meant only to present the data, but not convey its semantic meaning. The data cannot be easily extracted, exchanged, or tracked unless web wrappers are built around the web sources. Web wrapping allows the user to define parts of the web site to be extracted and converted into structured forms, such as XML or relational database tables.

A prudent reader may point to structure-defining Extensible Markup Language (XML) as the future standard of the World Wide Web publishing. However, we observe that while W3C has been working on XML since 1996, and proposed a standard in 1998 [4], XML has not even began to replace HTML as the method of publishing information on the web, and the browser support is so far limited. In fact, Electronic Data Interchange (EDI) – according to some opinions, a format of the past and the primary candidate to be replaced by XML – continues to see soaring usage, while businesses take a wait-and-see approach to XML [5]. Thus, we may conclude it will take significant time until XML makes any significant inroads at the current HTML stronghold.
Technology developed in the COIN group includes spec files – as an input to the Caméléon web extraction engine – that use regular expressions [6] to define the rules for data extraction from the web page into the attributes following a relational table model. The analysis of publications on building web wrappers shows that several different approaches were taken by various commercial and research institutions. In one approach, wrappers utilize the HTML tag-based hierarchy of the web pages to build a path over the parsed HTML tree to each extracted element. Some wrapper generators that belong to this category, such as OnDisplay eContent [7] or W4F (described in more detail below) allow for regular expressions application to fine-tune the granularity of the text to be extracted. Another method, Record-Boundary Discovery proposed by Embley et al [8] employs a multi-heuristic approach to discovering record boundaries in web documents, leveraging on the HTML tags of the page. In another approach, inductive wrapper generation techniques used by WIEN [9], STALKER [10, 11], and SoftMealy [12] rely on supervised learning techniques to generate the extraction rules that are not dependent on HTML.

Caméléon, based on Grenouille [13], ignores the HTML tag-based hierarchy and applies regular expressions to the web page for data extraction. This allows for any level of extraction granularity, and significant expressive power of the extraction rules. However, regular expressions are non-trivial to write, and that prevents average user from creating personal web extraction specifications.

Automatic creation of the Caméléon-compliant spec files based on user data selection from a particular web site is a major functional component of the Aggregator Tool. In the next section we will take a look at another developed web wrapper, W4F, and discuss its potential merits and shortcomings as compared to the Aggregator Tool. Like practically every other existing web wrapper, W4F is not fully automated and requires users to have some programming skills. The Aggregator Tool, in conjunction with Caméléon web extraction engine, provides powerful capabilities to the users without requiring any special skills.
2.2.1 Wrapper generator example: W4F

W4F, the World Wide Web Wrapping Factory project at the University of Pennsylvania [14], takes the following approach to wrapping the web sites and extracting the data. The retrieval rules specify the web sources of interest (thus closely correspond to the header contents of the spec files used in Caméléon.) The retrieved HTML document is then parsed into a Domain Object Model (DOM) tree structure. The extraction rules are applied on the parse tree and the extracted information is stored in an internal format called Nested Strings List (NSL). Finally, NSL structures are exported to the upper-level application, according to mapping rules. Rules are compiled using the W4F compiler into Java classes that can be run as a stand-alone application using the W4F runtime. Thus, the wrapper for each page is a collection of Java objects. Compilation time and the required user involvement are some of W4F drawbacks.

The extraction rules are written in an HTML Extraction Language specified by W4F. (This is a very similar approach to that of WebMethods’ Web Interface Definition Language [15].) Use of an extraction language combined with HTML parsing allows for some powerful and easy to construct extraction rules; however, this approach also carries some disadvantages. The W4F’s Extraction Language uses the parsed HTML structure to pinpoint any element on the page between the two HTML tags. To capture finer details of the document, such as splitting text between HTML elements to extract only parts of it, the language uses regular expressions. Unfortunately, almost none of this process is automatic. There is an extraction wizard that displays the path to any information between any two HTML tags on the page; for example, html.head.title will lead to the tree node corresponding to the title tag of the document. However, the paths of interest and all the regular expressions have to be composed by the user manually, and it is not an easy task. Although not suited for a user inexperienced in programming, the Extraction Language has one important advantage: a condition on extracted text can be specified. For example, it is possible to extract information in a column of a table the first row of which contains or does not contain a specific word. This extra bit of flexibility is important in reducing reliance on the web page structure. If a table is later changed to contain an additional column, the wrapper will
Automatically "know" that and will extract the data from the correct column. This as of yet is difficult if not impossible in Caméléon.

Caméléon uses purely regular expressions-based extraction rules, and can extract any information on the web page. Caméléon extraction is also quicker, since parsing the page requires considerably more time than applying regular expression pattern matching, and W4F approach involves parsing the page each time the data is extracted. While the Aggregator Tool also parses HTML page into a tree structure, as we will see later, this is only done to aid in automatic building of the regular expression extraction rules when creating spec files— in other words, it is only done once. W4F reliance on the parsed HTML structure of the entire web page may create an unrecoverable problem for the generated wrapper even due to some "unimportant" face-lift changes. For example, adding a top-level <FONT> HTML tag to change the font of the entire web page will cause the wrapper to fail. Caméléon avoids this problem by applying regular expression pattern only to a region of the web page specified by the "begin" and "end" tags in the spec file.

Because the ultimate result of the Aggregator Tool is a user-definable aggregator project, the tool combines the extracted data to create a presentation based on user's point-and-click input. W4F allows more flexibility in presenting extracted data, as mappings can be defined from the extracted information stored in a tree-like nested lists structure to any other desirable structure; however, these mappings have to be user-coded in Java. W4F provides no aid in agglomerating multiple-source extractions into a single 'project'. W4F defined yet another language to map the results into XML, and while this mapping is more powerful than Caméléon's XML output, the important distinction to remember is that Caméléon extraction follows a flat relational model and is intended for Structured Query Language (SQL) querying, while W4F uses hierarchical structures throughout. Caméléon's relational approach allows extracted data to be accessed by any complex SQL queries and thus presented in many ways or exported into any database.

Overall, W4F's lack of automatic tools to create the definition file for extracting information makes it hard to use for an inexperienced user. In addition, W4F cannot access
secure web sites (HTTPS protocol) or user authentication/login, both of which are handled by the Aggregator Tool.
3. Aggregator Tool Background

Choosing among various technologies for the Aggregator Tool implementation involved many design decisions about the nature of the tool itself. We present these decisions in the next section.

3.1 Signed Java Applets

The “thin client” software model is becoming increasingly widespread. More and more software applications are being implemented such that users are required to have nothing more than a web browser, which is most likely pre-installed on all the popular platforms by now. There are many ways to bring the necessary application functionality within the thin client paradigm: via dynamically generated web pages, application servers, or Java applets. Generally speaking, Java applets provide a better way to achieve any desired User Interface, faster performance, and greater cross-platform compatibility than a web-based application that is limited to browser-supported User Interface elements, and may have to rely on browser-specific technologies. Therefore, Java applets provide a unique platform for creating full-featured applications that are accessible via a web browser. Because vast majority of personal computers and workstations have a browser capable of executing applets deployed, applets also provide the easiest method of software distribution. Apart from some security restrictions imposed on them by the browser, applet creators can use all of the rich features of the Java programming language, including the platform-independent graphical user interface elements.

Unfortunately, the Java Virtual Machine (JVM) implementations in various browsers are inconsistent. There is also a noticeable delay between the time a new Java version is made available and its integration in the popular browsers. To get around the problem, care should be taken to use only such features of Java that are present in the older versions of JVM distributed with the browsers, or to create multiple versions of the applet so that its interface and functionality appears the same in different browsers. Sun Microsystems has attempted to address these issues by creating the Java Plug-in. The plug-in is free to download and is usually made available shortly after a new version of the JDK is released for
all the platforms. The Plug-in replaces the internal JVM used by the browser. Even though this approach works better in somewhat controlled environments than for Internet-wide deployment, we chose it for the current version of the Aggregator Tool. When a browser without the plug-in installed runs an applet intended to be used with the plug-in for the first time, the user will be prompted to download the plug-in. The download procedure is straightforward.

The decision to implement the Aggregator Tool as an applet, not an application, is largely dictated by the simplicity of use and desire to avoid placing significant computational requirements on the user. Rather than having to download the application to the local hard drive, the user should be able to use the tool from any web browser, as dictated by the thin client model. However, because malicious applets can cause damage to the unwitting user’s machine, the default browser security mechanisms ("sandboxing") impose restrictions on the applets. Applets are prevented from communicating with any outside sources apart from the server an applet was downloaded from. In our case, this means the Aggregator Tool applet would be prevented from accessing remote websites or the registry of spec files and user information on the server, or from reading and writing any files. However, like most useful applets, the Aggregator Tool requires connectivity to data sources and other applications to accomplish its tasks. One mechanism that allows applets to achieve that is through a back-end server running on the same host an applet was downloaded from. Java servlets, described below, are one kind of such servers. The applet then sends all its requests for data or intensive computations to the servlet, or sends the servlet the data it stores on the server file system.

Another mechanism invented to allow non-malicious applets to escape from the browser-induced security sandbox is code signing. Recent browsers have provisions to give "trusted" applets the ability to work outside the sandbox. To gain greater access to the system, applet code must be digitally signed by a known Certificate Authority with an unforgeable digital ID, or by the applet creator who obtained a certificate from the Certificate Authority. Digital IDs are in effect statements signed by a Certificate Authority stating, "This company (or a person) is who they say they are, and this CA vouches for them.” A simple dialog box is then presented to the user when the browser first runs such an applet,
and the user must state that he trusts applets signed with that ID. Therefore, users have a choice of declining the applets from untrusted sources, or permitting applets of known origins, since the origins are clearly specified by the certificate.

Obtaining digital certificates from Certificate Authorities automatically known to the browsers, such as VeriSign or Thwarte, is expensive. We used the equivalent free “test” signing procedure that allows me to generate my own key to encrypt the applet code and sign it by my own “certificate”. Because no browser recognizes this certificate by default, users must perform several steps to grant such applet permissions in excess of the usual restricted ones: a) download my certificate file from a web page and store it locally on a user machine; b) import the certificate file by running keytool command that comes with Sun’s Java Plug-in; c) download my policy file from a web page and store it locally. This relatively complicated procedure would not be necessary if the recognized certificate for the Aggregator Tool applet were purchased from the Certificate Authority.

3.2 Servlets

Servlets are the server side components written in Java. Servlets provide a framework for services built using the request-response paradigm. Servlets are widely used in making HTML web pages interactive, by building them dynamically based on user input. However, servlets’ capabilities extend beyond that, as they can be used to communicate with the client-side application and applets, and serve as a back-end for persistent data storage on the server’s file system or a database.

The Aggregator Tool uses a servlet to store and retrieve user spec files and project files on the server file system. To communicate to the servlet, a Java applet opens a URL connection to the servlet, and either passes it name-value parameter pairs (thus simulating HTTP GET request), or any serialized Java object (thus automatically invoking HTTP POST request). The servlet is designed to handle these two kinds of requests. In response, the applet can receive either an HTML page or a Java object from the servlet.
3.3 Relational Database and JDBC

The Aggregator Tool uses a relational database (henceforth referred to as registry) to store its user information. When a relational database is used for storing application data, the design of the application using it should be modular enough so that it could work with any relational database that supports the set of features needed. The Java Database Connection (JDBC) API is the ideal solution for this problem. The JDBC API allows an application to access virtually any database on any platform with a Java VM, as long as the driver for that database implements the JDBC API. JDBC drivers for practically any database have been written and are freely available.

Working with JDBC API usually involves 3 basic steps: connecting to the database, creating and executing a statement, and processing the result set. Aside from the connection, there is virtually no dependence on the database itself, and majority of the code involves working with JDBC Statement and ResultSet classes.

3.4 JavaServer Pages

JavaServer Pages (JSP) technology allows easy development of information-rich, dynamic web pages. JSP technology separates the user interface from content generation enabling designers to change the overall page layout without altering the underlying dynamic content. JavaServer Pages use XML-like tags and scriptlets written in Java to encapsulate the logic that generates the content for the page. Additionally, the application logic can reside in server-based resources (such as JavaBeans component architecture) that the JavaServer page accesses with these tags and scriptlets. The layout of the page itself is controlled by HTML or XML tags, and the product of the JSP is pure HTML (or XML) that can be displayed in a browser.

Using JavaServer Pages as a vehicle carrying users’ aggregations means the ultimate portability: whenever the user wants to re-run his or her “aggregation”, the user can simply type in a URL of the JSP file (for example, http://server.mit.edu/userid_paperclips.jsp) to instantly view the information extracted in real time. This makes the extracted data always
accessible from any browser, which in a near future will also mean accessible from most mobile communication devices.

However, one important issue to keep in mind is user security and privacy. Consider the following question: if a user finds out about the JSP filename (i.e., a URL) that Citibank distributed only to a selected group of its customers, what prevents this unauthorized user from accessing it? This question actually ties in with another question: assume Citibank created an aggregator (a JSP) for its customers' bank statements. Each customer should then use his or her own Citibank user ID and passwords with the provided JSP. How can this be accomplished? The JSP aggregation files should be served by a secure server (for example, using Secure Socket Layer), and when accessed, pop up a login box for the users to enter their Aggregator Tool username and password; the JSP Bean classes should then access the Aggregator Tool registry to find all the appropriate (possibly multiple) passwords for web pages to retrieve data from. This implies that either Citibank needs to have the knowledge about all the users' IDs/passwords for the web pages involved in the distributed JSP aggregation as well as an "update" access to the registry database to populate it with these IDs/passwords, or after getting the JSP from Citibank users need to open it in the Aggregator Tool at least once to set up their web pages ID/passwords. Note that we assume users are likely to use the Aggregator Tool for multiple aggregations, but they only need to remember their single Aggregator Tool username/password for accessing the wealth of aggregated data after a one-time setup – a significant user convenience.
4. Implementation

4.1 Graphical User Interface

The Aggregator Tool Graphical User Interface consists of three main tabs: Project management and creation, Web Wrapping (spec file creation), and Results display.

Figure 2. Project Browser Interface.
Upon login, the user is presented with the Projects display (Figure 2). The project display area is split into three vertical parts, such that the interactions user would normally perform would flow from left to right (Figure 3).

![Diagram of Project Browser Elements]

**Figure 3.** Interaction of Project Browser Elements.

The rightmost part of the display contains the lists of projects, project-related buttons, and overall project settings underneath. The middle portion of this screen contains the lists of spec files, spec file-related buttons, and project settings that control the spec files. Finally, the rightmost column contains the spec file's attributes. Throughout the session, buttons are enabled or disabled depending on the user's selections, thus providing a "guide" to possible actions. The logical flow of user's actions could be as follows: Upon login, public and user's private spec files and projects list boxes are automatically filled in with the information pulled from the database. The user can select any project to see its description and the spec files it contains in the "Selected Project's Specfiles" list box. The user can delete the project, or shift attention to the right and select one of the spec files. If one of the selected project's spec files is selected, "Remove File From Project" button is enabled; otherwise, "Add File To Project" button is enabled. (An alternative implementation could change the caption and the
action of a single button depending on the spec file selection.) Moving on to the right of the screen, once the user selects a spec file, the attributes it defines and the User ID and Password needed to log in to the page it wraps are automatically filled in on the rightmost display portion. The user has a chance to update his or her user ID/password, change the selection of attributes for this project, or set any attribute-specific project settings, such as the attribute to sort the results by. The “Sort By Attribute” checkbox is only enabled when meaningful, that is, when user clicks on one of the Selected Attributes. Any change the user makes to the project enables the Save button. In addition to any changes to the existing projects, the user can create a new public or private project, and the applet will automatically place it as a selected project at the top of the appropriate projects list. None of the user’s changes or updates actually takes effect until the Save button is pressed; thus exiting the application without saving would simply discard the changes.

The class responsible for the User Interface is built such that it contains practically no functionality beyond controlling GUI. It queries the User, Project, and Specfile objects to obtain the lists of names for the list boxes, and provides the User object with user’s selection so that the latter could load and “cache” the appropriate project or spec file. If the Project or Specfile classes grow in the future to accommodate additional functionality, the existing User Interface code will continue working without modifications, and vice versa. The User Interface also implements a mechanism that allows only a single project or spec file selection among two or three list boxes, correspondingly. For example, selecting a private spec file will automatically unselect any previously selected project or public spec file.

The User Interface class makes a connection to the servlet to load the body of a spec file, and to write out user-modified projects and newly created spec files (An alternative implementation could easily place these functions in a separate class). To load a spec file, the UI issues a GET request to the servlet, which specifies the unique location of the spec file to be loaded. The UI then reads a serialized object that represents the spec file from the servlet’s input stream. Similarly, to permanently store modified projects or newly created spec files, the servlet needs to write them as files at the appropriate location on the server. To accomplish this, a vector containing the location and the body of the file to be written is created and passed to the servlet’s output stream. This operation automatically serializes the
vector object, and triggers the “doPost” servlet method call. The servlet reads the object from its input stream and writes the appropriate file. Currently, since the Camélleon engine can only access spec files in a particular directory, the location information passed to the servlet is stripped, and the spec file is written in that directory. The project files are written into a directory accessible by the JSP server.

4.2 Project Browser Design

The objects used in the Aggregator Tool are designed to be as GUI-independent as possible. The data model corresponds to the real objects the applet works with and needs to maintain the state of: a currently logged on user, projects, and spec files. The User class maintains the state particular to the current Aggregator Tool user: a list of user’s private projects and spec files, a list of new spec files user may have created that will have to be written out as files, as well as the lists of public spec files and projects. For better efficiency, the User object also maintains a cache of opened project and spec files. Since thousands of users could use the Aggregator Tool in the future, and the database could store the information about a great number of created projects and spec files, upon user login the User class queries the database only for the necessary minimum information: names of existing public projects and spec files, and the user’s private projects and spec files. The User Interface will query User object for this information to fill up its list boxes. Later, when the user selects a particular project name in the list box, the corresponding Project class object is created and filled with the project-related data (spec files it contains, project settings) by querying the registry database. Any subsequent selection of this project will not result in a database query, since the Project object is now stored in the User class’s list of opened public or private projects, as appropriate. A modification on the selected project, such as addition or deletion of spec files it contains, affects the Project object and not the registry database or project file itself. Thus the number of costly database, servlet, and file system operations is minimized. The User Interface calls the User class methods for adding, removing, updating project and spec file information.

The Project class contains the information to describe a user project: its name, location by which it can be uniquely identified, the list of spec files it contains, and the list of
attributes selected from each file for display in this project. In addition, as more project-specific user functionality is added to the Aggregator Tool, the Project class would maintain the necessary information. For example, currently it stores the spec files that constitute each project in a hashtable that defines a mapping from each Specfile class to a vector containing the selected attribute index to sort the results by as its first element, and the spec file’s selected attributes as its remaining elements. If the project functionality is significantly enriched in the future, a separate “project settings” object could be stored in the database in a serialized form as a Binary Large Object.

An important function of the Project class is its ability to write itself out. A project object contains all the information it needs to create the resulting JSP file: the spec files it contains, and the attributes selected for extraction from those defined by each spec file. Depending on the user settings, it would also know to generate the call to the appropriate JSP bean that implements the specific functionality a user requested, such as sorting the results by a particular attribute. Additionally, the Project object maintains a “dirty bit”. This boolean flag aids the User object in selecting which project files to write out when the user clicks on a Save button. Even though Project objects for all the project entries the user clicks on are created and loaded into User object’s “cache”, only the modified projects will be updated on the disk, and only when the user indicates he or she wants to save the changes made.

Similarly, the Specfile class stores all things spec file-related: the body of the spec file, its name and location, the list of attributes it defines, and finally the User ID and Password, if needed for the site this spec file is defined for. Whenever the user selects a spec file in the GUI, the User object will prompt the corresponding Specfile object to “load” itself. This involves obtaining the User ID and Password, if any, from the database, and, since currently the spec file attributes are not stored in the database, loading the spec file body and parsing it for attributes (this may change in the future.)

The database access functions, grouped by the dbAPI class, are called by the User class. Each function serves a specific role, for example, obtains the list of user’s private projects, spec files, or a user ID; deletes or updates database entries, etc. Each function does this by creating the appropriate JDBC SQL statement and returning the result in a vector or a
hashtable. For simplicity, this class is part of the applet in the current implementation. However, for a thinner client, better security and better database performance in multi-user environment, it could easily be called by a servlet instead without many modifications. The servlet would hold a pool of open database connections to choose from. The servlet would also be more effective in dealing with data validation, such as ensuring uniqueness of database primary keys when updated by multiple client applets. Using a simple GET request, the applet would then “call” the necessary method and pass it the arguments, which are exclusively strings, such as username or project location. In response, the servlet would pass back a serialized vector or hashtable object of results.

4.3 Registry

In the future, the Aggregator Tool will be one of many applications using the registry database. The registry design has to address the needs of all these applications. While this document does not address the full registry design and contents, the following sections review the requirements placed on the registry by the Aggregator Tool, and describe the registry fields it uses.

4.3.1 Overview

The Aggregator Tool applet is intended for use by multiple clients. Thus, the applet starts by displaying a user login box, requesting a user ID and password. The login process should cause the applet to look up user information in the registry database via a secure link for the purpose of use authentication. As mentioned earlier, all the communications between the Aggregator Tool and the registry are done through a registry access API (implemented in dbAPI class.) For each user, the registry should contain pointers to the user’s “private” spec files he or she created. In general, the spec files created by the Aggregator Tool reside on the same server, and if marked public, are visible to and usable by all the users of the Aggregator Tool. The spec file location entry in the registry for these files is the subdirectory where they are stored. Spec files could also reside on remote servers. In this case, the registry would contain a URL of the spec file as its location. In the future, the registry may also store a user ID associated with certain public spec files to enforce the notion of a spec file “owner”. For
example, if a user ID is present, it would indicate that this public spec file can only be modified or deleted by the user who owns it; otherwise, anyone can overwrite (modify) the spec file. The registry should also contain saved user and public aggregations, or projects. Each aggregation defined by the user is saved under a user-assigned project name. Each user may opt to create multiple “aggregations”, for example, one to track retail prices, and another to manage user’s finance portfolio. (In essence, a project is a collection of spec files, “bundled” with user’s personal information (passwords), and some basic settings for displaying the extracted results.) In addition, similarly to spec files, publicly provided projects may exist, possibly distributed by an organization – such as a bank – that will allow any user to customize them and provide his or her personal passwords where appropriate.

4.3.2 Schema

To achieve these objectives, the registry database uses the following relations:

<table>
<thead>
<tr>
<th>User</th>
<th>ProjectName</th>
<th>ProjectLocation</th>
<th>Settings*</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ProjectLocation</th>
<th>SpecFiles</th>
<th>SpecFileLocation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SpecFileLocation</th>
<th>User</th>
<th>ID*</th>
<th>Password*</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PublicProjectName</th>
<th>ProjectLocation</th>
<th>Settings*</th>
<th>Description*</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PublicSpecFile</th>
<th>SpecFileLocation</th>
<th>Description*</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*allowed NULL entries.

Figure 4. Registry Database Schema
4.3.3 Functionality

Users should be able to name their spec files and projects arbitrarily. For example, user B should be able to name his private spec file etrade even if user A has already created his own file named etrade; similarly for project names. Spec files can be located on multiple servers. However, if both user A’s and B’s spec files are located on one server, name clash should be avoided. (For the purpose of this discussion, the server refers to the central server that runs the Aggregator Tool servlets, which is the default server for storing users’ JSP pages and spec files. In fact, in the current implementation the database is located on a different physical machine from the server.) In addition, a collection of public spec files and projects should be maintained. Identically named multiple public spec files should not be permitted. Finally, assume a company such as Fleet or Citibank used the tool to create a project for its customers. The project information should be readily available in the registry for the Aggregator Tool users.

4.3.4 Example Scenario

A user, user1, creates a project and calls it myportfolio; creates a spec file firstUsa to be used in the project, as well as a publicly available spec file named etrade. A different user user2 uses a publicly available project, citibank, and his own project, myportfolio, which uses only etrade spec file. user3 also uses the project citibank, but user4’s citibank is an unrelated private project that happens to have the same name. It uses a different, private file that the user named etrade and stored on a remote server. The registry then looks as shown on Figure 5.
| user1  | myportfolio | \user1\myportfolio\myportfolio.jsp | user1 myportfolio settings |
| user2  | citibank    | \citibank\citibank.jsp           | user2 citibank settings   |
| user2  | myportfolio | \user2\myportfolio\myportfolio.jsp | user2 myportfolio settings |
| user3  | citibank    | \citibank\citibank.jsp           | default citibank settings |
| user4  | citibank    | http://remote.edu/citibank.jsp    | user4 citibank settings   |

| \user1\myportfolio\myportfolio.jsp | firstUsa | \user1\myportfolio\firstUSA.spec |
| \user1\myportfolio\myportfolio.jsp | etrade   | \etrade.spec                     |
| \citibank\citibank.jsp            | citi     | \citibank\citi.spec             |
| \user2\myportfolio\myportfolio.jsp | etrade   | \etrade.spec                     |
| http://remote.edu/Citibank.jsp      | etrade   | http://remote.edu/etrade.spec    |
| \fleet_tool\fleet.jsp              | fleet    | \fleet_tool\fleet.spec          |

| \user1\myportfolio\firstUSA.spec   | user1    | user1_ID1 | User1_pwd1 |
| \etrade.spec                       | user1    | user1_ID2 | User1_pwd2 |
| \Citibank\citi.spec                | user2    | user2_ID1 | User2_pwd1 |
| \etrade.spec                       | user2    | user2_ID2 | User2_pwd2 |
| \Citibank\citi.spec                | user3    | user3_ID  | User3_pwd  |
| http://remote.edu/etrade.spec       | user4    | user4_ID  | User4_pwd  |

| fleet_tool | \fleet_tool\fleet.jsp | project settings | project description or NULL |
| Citibank   | \citibank\citibank.jsp | project settings | project description or NULL |

| etrade     | \etrade.spec            | file description or NULL |
| unused_public_file | \unused_public_file.spec | file description or NULL |
| remote_file | http://moc.com/remote_file.spec | file description or NULL |

**Figure 5.** Sample Registry Contents.
Notice that when two users name their own private projects the same, the Aggregator Tool automatically places them in separate directories, as in the example above (first relation). Also, the first relation above assumes user3 decided not to change any project settings when using the public project citibank. Whenever a user starts using a public project, the system creates an entry for that user in the first relation, and will use default project settings automatically unless the user opts to change them, as user2 did for the same citibank project.

4.3.5 Why are these relations needed?

The first relation groups all users’ projects and their settings. If a million users use the project citibank, each may want to adjust project’s settings. Additionally, a user may have named his or her private project citibank, as user4 did. The system distinguishes between the two by automatically placing the project into username_citibank directory. The key of the first relation is (User + ProjectName).

The second relation states which spec files are used by all of the existing projects. Using ProjectLocation as a unique identifier for each project in existence, this table stores the physical location of each spec file these projects used. Note how this abstracts the user-assigned spec file name from the physical spec file. Just because user1 named his spec file firstUsa does not mean user5 should not be able to assign the same name to his own, unrelated spec file. The key of the relation 2 is (ProjectLocation + SpecFiles).

The third relation associates user ID/password with each spec file that requires them. Using SpecFileLocation as a unique identifier for each spec file in use, this table stores the user information Caméléon will “insert” into the spec files at the time of data extraction. The key of the relation 3 is (SpecFileLocation + User).

The fourth relation lists all the public project names, their physical location, and project settings. These default settings will be used if a user chooses not to modify them when using a public project. When a company uses the Aggregator Tool to create a project
and marks it as "public", this is the only relation that will store its information initially. The presence of Project Description, which may be quite useful for a public project, in the registry can be viewed as "cache", since it could easily be stored in and read from the beginning of the project (.jsp) file, delimited as a comment by some marks, although this is not implemented in the current version. However, that would require a read access to all the project files, some of which may be located on remote servers, every time a user logs onto the Aggregator Tool and lists all the public projects. Storing this optional description in the registry makes its access faster.

Finally, the fifth relation lists all the public spec files. It is similar to the previous relation.

4.3.6 Browsing

When a user logs in, the Aggregator Tool displays a list of all the user's own private spec files, public spec files, user’s private projects, and public projects.

To display:
- Private spec files: Select User’s records from the first relation; using ProjectLocations retrieve all the SpecFiles from the second relation. Filter out the duplicates using SpecFileLocation; filter out the public spec files using the fifth relation, to display them separately.

### Corresponding SQL statement:

```sql
SELECT DISTINCT specfiles, plocation.specfilelocation FROM users, plocation WHERE user = ? AND users.projectlocation = plocation.projectlocation AND plocation.specfilelocation NOT IN (select pubspec.specfilelocation from pubspec)
```

- Public spec files: These files are grouped by the fifth relation. We envision public spec files as a library of spec files provided for anyone’s use in any project.
Corresponding SQL statement:

```
SELECT publicspecfile, specfilelocation FROM pubspec
```

- **Private projects:** Look up all the User's projects in the second relation; using ProjectLocatoins filter out the public ones using the fourth relation.

- **Public projects:** The fourth relation contains all the public projects. These projects may have been defined by a company and distributed to the users. Should the Aggregator Tool also list the Public Projects which the user looked at or modified settings of, this could be done by using the first and the fourth relations.

### 4.4 Web Wrapping

The Web Wrapping engine is at the heart of the Aggregator Tool. The goal of the automatic web wrapping is to create robust spec files while requiring the minimum input from the user. Spec files define the data to be retrieved from the web sites and the rules for its retrieval. In essence, spec files structure certain web site data in such a way as to be accessible by a database query language. Spec files assign attribute name to each data item to be extracted. Generally, there is one spec file per web page or set of web pages of interest. A spec file contains a source section consisting of a URL, http method (Post or Get), and optional parameters for Post method, followed by the attributes section that defines regular expression patterns for extracting each attribute from the web site data. However, there are some cases in which user needs to go through multiple web pages while entering some data into the forms to get to the final page where the data of interest resides, and in the process a unique URLs are built by the web server, based on the user-entered parameters, a unique session ID, etc. A spec file for such web site would contain multiple source sections, and will define a rule to capture a URL for the next source section as an attribute in the previous section. Caméléon has recently been modified to work properly with such spec files.
4.4.1 User Interface

The Web Wrapper pane (Figure 6) of the Aggregator Tool allows users to easily wrap web sites of interest.

Figure 6. Web Wrapper Interface
After switching to the pane, the user can enter the URL of the site he or she wants to wrap. The URL is automatically inserted into the input table. In addition, the URL is parsed so that the parameters and their values, such as “s” and “IBM” in “http://finance.yahoo.com/q?s=IBM” are placed in the input table as well. If the user replaces a value with some arbitrary name, the Web Wrapper will use it as an attribute name to parameterize the source (URL) section of the spec file*. The input table also contains the relation name whose value is expected from the user. The web page is rendered in the upper portion of the pane. (The applet is allowed to access remote web sites since it has been digitally signed.) The user’s main task is to provide values for the output table. This can be easily done by selecting a particular row of the output table; selecting (highlighting) an item of interest on the page; and clicking on the appropriate Add button. The corresponding cell of the output table’s selected row will be filled by the user selection. Each row of the output table corresponds to the definition of a single attribute. Once the row is filled, the user can automatically add a new row by adding a new attribute while no output table row is selected. The user can also remove the unwanted rows very easily. In addition to automatically placing the highlighted information into a selected cell, users can edit the cells manually. For example, the user may assign an attribute name to each attribute by typing it in a Name cell of each row. In the future, Web Wrapper may present the user with a drop box of possible types for each attribute he or she defines, such as “number”, “date”, “string”, etc., or attempt to guess the correct type by looking at the kind of data the user has selected for extraction. Currently Caméléon does not distinguish between types, and all the datatypes are strings by default. Also, since the Aggregator Tool has no way of knowing whether the user intends to extract the hypertext link if the selected extraction item is linked, a checkbox option “Extract Link” could be added.

The Aggregator Tool applet must feature two main characteristics: the ability to render and browse arbitrary HTML pages, and to acquire user selections on the rendered HTML page. Java conveniently features these abilities via JEditorPane. Java has built-in support for both HTTP and URL-based connections. As part of the Swing library, JEditorPane automatically renders HTML 3.2 when it is passed a URL of a web page. It is

* In this example, the resultant spec file will contain
#Source=http://finance.yahoo.com/q?s=#arbitrary_name#
then trivial to obtain any user text selection by a simple Java call. An implementation of mouse listener also allows for navigating the hypertext links within JEditorPane the same way it is done in the browser. Finally, Java HTTPClient [16] replaces Java URL handling seamlessly and is able to deal with authentication, redirection, and cookie issues when connecting to web pages. This is the current implementation. Unfortunately, JEditorPane is buggy and its capabilities are limited. It does not properly render HTML 4.0. Neither does it interpret any JavaScript included in the page. While JavaScript is mostly used for unimportant interactive or cosmetic enhancements on most web pages, such as changing icons on mouse pointer movements, it is increasingly being used for creating important cookies, or sometimes even for generating contents of a page.

A full-featured alternative, Microsoft Internet Explorer browser can be used as an object control accessed from other Windows or Java applications. The browser control provides full web browsing and rendering capabilities, including JavaScript interpretation and the latest HTML compliance, embeddable into applications without Explorer’s customary menus and buttons. While this would make an ideal choice for the purposes of web wrapping, unfortunately it greatly limits the platforms on which the Aggregator Tool would be available to those with Microsoft Explorer. Additionally, the Aggregator Tool applet would have to be written using Microsoft’s own Java SDK, non-standard extensions of which are necessary to access and manipulate Internet Explorer and an object. Thus, the Aggregator Tool would effectively become a Windows-only application.

There are several alternative third-party methods of displaying HTML within a Java application, such as HTMLWindow (http://home.earthlink.net/~hheister/), or ICE browser (http://www.icesoft.no/ICEBrowser/), and others, but since they face very small demand and formidable task of competing with fast-moving commercial browsers, none are very robust, nor do they provide any Java calls to obtain user selection in their APIs.

The ideal candidate for the job is Mozilla browser. The latest generation of Netscape standalone rendering engine, Gecko, is fully embeddable in any other application. As part of the Mozilla effort, Project Blackwood [17] works on the integration of the Java Platform with Mozilla. Embedding the Gecko layout engine in an application hosted on a Java virtual
machine requires a Java Wrapper API for Gecko. Unfortunately, at the time work on the Aggregator Tool was being done, this wrapper, which consists of Java interfaces that present the functionality of Gecko's layout engine to the developer as a JavaBeans component, was in its infancy. The project has matured since, so the future implementations of the Aggregator Tool's Web Wrapper should leverage Mozilla-based browser for HTML rendering.

### 4.4.2 HTML Parser

Camélleon, like its predecessor Grenouille, applies Perl5-compatible regular expressions to the HTML pages to extract information. While this approach allows for matching with any level of granularity, regular expressions are hard to form and understand by regular users. It is possible for experienced programmers to create regular expressions that will effectively extract practically any data from a set of web pages. However, it is quite challenging to create these expressions programmatically based on the limited input of a novice user.

The ultimate product of web wrapping, spec files consist of a source portion followed by an attributes portion. For instance, the spec file reproduced in Figure 7 creates a relation moneyrates by extracting some information from http://www.money-rates.com/mmarket.htm into an attribute named bankname:

```plaintext
#Relation=moneyrates
#Source=http://www.money-rates.com/mmarket.htm

#Attribute=bankname#String
#Begin=INFO
#ObjectBegin=<TR\s*bgcolor=
#Pattern=(?:<TD[\O-\377]*?</TD>\s*){2}<TD[\O-\377]*?<FONT[\O-
\377]*?\s*(?:.\s)*\s*</FONT>[\O-\377]*?(?:<TD[\O-
\377]*?</TD>\s*){2}
#ObjectEnd=</TR>
#End=highlights
```

**Figure 7.** Sample Spec File Listing.
Begin and End elements, when matched on the web page, delimit the region of the HTML that contains the attribute data we want to extract. Starting from the beginning of the page, the first Begin match will be used. Thus, care should be taken to provide a unique value for Begin. The first End match on the page following the Begin terminates the region. Each attribute may have its own Begin and End pair. In addition, attributes will be matched only between ObjectBegin and ObjectEnd, if present, which delimit attribute groupings into tuples (see [3]).

To aid in creation of regular expressions, Web Wrapper resorts to the HTML tag-based hierarchy. To do so, HTML must be parsed into a tree structure. There are several ways to obtain the tree hierarchy of a page. The Document Object Model (DOM) is a platform- and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure and style of documents, including HTML documents. DOM tree representation of a web page allows every object (element, link, etc.) on the page to be manipulated; it also could be used to construct a path by traversing a tree to the particular element user selected on the page to be extracted. This is the approach employed by XWRAP [18], W4F, and OnDisplay. However, the goal of Web Wrapper is to be able to wrap a page based on simple user text selections, instead of presenting a page disassembled into its elements and asking user to pinpoint his or her selections.

In an ideal world, all HTML pages would be DOM-compliant. However, the world is not ideal, and a great majority of HTML pages do not conform to the HTML standards, taking advantage of forgiving nature of browser HTML rendering. HTML Tidy [19] is a tool that recognizes mistakes in HTML code and makes pages compliant. However, in the process of doing so the original page is modified; thus building regular expressions based on a parsed tree of a page that has been “tidied up” is meaningless, as they would most likely not properly work on the original web page.

A working solution we found was JavaCC HTML Parser [20]. Java Compiler Compiler (JavaCC) [21] is a popular parser generator for use with Java applications. A parser generator is a tool that reads a grammar specification and converts it to a Java program that can recognize matches to the grammar. This parser uses JavaCC grammar to parse HTML
documents into a simple, flat parse tree. It does not enforce or require a proper HTML structure, but instead attempts to accept any document accepted by most web browsers. According to the Parser’s web page, “The goal was to produce a parse tree which threw away very little information contained in the source file, so that by dumping the parse tree, an almost identical copy of the input document would result.” This fit perfectly with my goals. (In fact, the “almost” in the statement above can be safely ignored since the discarded by the parser information – a white space in tag attributes – will always be replaced by the “match-whitespace” regular expression.)

4.4.3 Pattern Generation

Once the HTML page is parsed, classes employing commonly used Visitor design pattern [22] can be made to walk in any direction over the resulting parse tree while performing a specific task, such as creating regular expression pattern to match the element user selected. Because we do not require users to point to a particular HTML element, but instead accept text highlighted on the web page, it is now the Web Wrapper’s challenge to find the selected elements in the parsed tree. To do so, we wrote visitor class TextCounter that collects all the text elements of the page into one large string, and keeps track each element’s index in that string. Given user selected text, we can then find it in the large string and use indices mapping to obtain the HTML parse tree element. Since any HTML tag is interpreted by the browser and not displayed to the user, it is a fact that any text the user selected is in fact a text element as parsed by the HTML parser. There are several caveats in this tactic: extra whitespaces (spaces, tabs, or new lines) in the HTML code are not actually displayed by the browser, while special escape characters, such as &lt; or &nbsp; would be rendered as “<” and “ ” by the browser – and therefore in the user’s selection the program works with. We deal with the first problem by creating whitespace-tokenized versions of the large text string as well as the user’s selection string, and attempting to locate the selection words as adjacent words in the large text string; we deal with the second situation by substituting appropriate HTML escape sequences for the special characters in the user’s selection string.
Once Begin and End elements are found, care must be taken when generating the actual regular expressions for them. Because we determine the exact indices of the beginning and ending of the selected text, we can retrace the elements in the parsed tree and substitute appropriate regular expressions if needed. For example, if the user’s Begin selection is “Important Notes”, but in reality it corresponds to “<I> Important</I> Notes” HTML code, the generated matching expression would be “Important[^>]*\s*Notes”, i.e., match the word Important, followed by any tag, followed by any number of spaces, followed by the word Notes (for a detailed treatment of regular expressions, refer to [6]). Element indices of both the start and end of the selection are stored for Begin and End values. (These indices are equal if the selection spans a single text element.)

To create a regular expression pattern for the element extraction, we invoke PatternBuilder visitor class, passing it the indices calculated earlier. PatternBuilder creates a regular expression pattern by “visiting” the web page elements starting with the extraction element going backwards to the Begin element, then from the extraction element forward to the End element. If the parsed text element that contains Extract selection holds any additional text, that text is be added to the pattern. In addition, elements immediately surrounding the extraction element are added to the pattern. The matching pattern itself, \([\0-\377]*\)? is added to the pattern string. As HTML elements are visited, each of them adds appropriate information to the pattern. Text elements contribute a “skip all” pattern, \([\0-\377]*\)? (care is taken not to use this pattern more than once in a row); newline elements contribute “match space characters” pattern, \s* ; finally, “significant” tag and end-tag elements contribute themselves without the attributes, i.e. <\[Tt\] [Aa] [Gg] [^>] > pattern. We observe that the following tags:

\td \tr \a \b \t \r \l \i \u \l \h \r

are “significant”, that is, they are more likely to contribute to the long-lasting layout of the web page, instead of serving as a more volatile “cosmetic” text property modifiers.

\(^1\) This pattern states, “Match any value non-greedily, and store the result.”
\(^2\) This pattern states, “Match <TAG regardless of case, followed by any number of any character until the first > is matched.”
A wonderful application of the visitor design pattern is the ability to implement multiple visitors without affecting the rest of the code in the system. For example, if the extraction element is in the HTML table, PatternBuilder invokes a specialized TableVisitor. This class ignores all but <TD> and </TD> tags, and terminates pattern creation when it reaches <TR> tag, therefore creating a pattern that extracts a particular column of an HTML table. Other specialized visitor pattern creators that use more sophisticated heuristics can be implemented in the future.

Finally, in case a particular wrapping requires ObjectBegin and ObjectEnd, which are almost always some HTML tags, we present the user with the generated tree of parsed HTML elements. Selecting a tree element and clicking on “Add Begin” or “Add End” button while in the Tree pane will insert the selected tree item into the ObjectBegin or ObjectEnd cell of the output table, respectively. To simplify the task as much as possible, the tree is built such that only the elements between Begin and End values of the selected output table row are present, and the extraction element is selected and scrolled into view for user convenience, while the rest of the tree is collapsed (Figure 8). This is accomplished by parsing the document when the Tree tab is selected, and inserting Begin, End, and Extract annotation elements into the parsed document structure. The tree builder visitor class uses them to build the minimum necessary tree. In the future, the Web Wrapper will “suggest” the correct elements by selecting them automatically, which the user will simply be able to accept. This is possible because ObjectBegin/ObjectEnd elements are most often the table row (<tr>, </tr>) tags, etc.
For example, consider extracting rates and bank names from the following HTML code fragment of Savings and Checking Accounts page on Money-rates.com (http://www.money-rates.com/savings.htm):

**Figure 8.** Parsed HTML Document Tree Display.
<table>
<thead>
<tr>
<th>Rate</th>
<th>Yield*</th>
<th>Bank</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.06%</td>
<td>6.25%</td>
<td>American Bank</td>
<td>6.25% APY</td>
</tr>
<tr>
<td>5.84%</td>
<td>6.01%</td>
<td>everbank</td>
<td>Offers</td>
</tr>
</tbody>
</table>

The user chooses "Notes" as the Begin element. It is a unique text element positioned right before the table that carries our information. The user then selects "Savings Accounts" as the End element.
Accounts" as the End element, <TR and </TR> as ObjectBegin and ObjectEnd elements, and designates “6.06” as the attribute named “Rate” to be extracted. TableVisitor will create the following pattern by browsing through the parsed elements backwards from 6.06 to the beginning of the table row, followed by browsing forward from the percent sign (%) following 6.06, to the end of the table row:

`<TD[^>]*>[\0-\377]*?<FONT[^>]*>([\0-\377]*%)?\[0-\377]*?\<TD[^>]*>[\0-\377]*?<TD[^>]*>[\0-\377]*?\s*\[0-\377]*?\<TD[^>]*>[\0-\377]*?</TD>`

This is equivalent to the manually generated spec file pattern like the one shown in Figure 7.

Figure 10 shows an example of aggregation produced by the Aggregator Tool by wrapping two web sites shown on Figure 11. Figure 9, below, shows the generated JSP file that produces this aggregation.

```html
<html>
<body bgcolor="#FFFFCC">
<%! String[] urls =
{"http://rombutan.mit.edu:8081/servlet/camserv?query=Select+BankName%2C+Rate+from+stumoneyrates&format=tsv",
"http://rombutan.mit.edu:8081/servlet/camserv?query=Select+BankName%2C+Rate+from+banxQuote&format=tsv"}; %>
<jsp:useBean id="foo" scope="page" class="aggregator.Aggregator" />
<% foo.setUrl(urls); %>
<br>
<H2>The results are:<H2><br>
<table>
<%! String[] lines; %>
<%
lines = foo.getContents();
for (int i=0; i<lines.length; i++) {
    out.println(lines[i]);
}
%>
</table></body>
</html>
```

Figure 9. Example JSP code.
<table>
<thead>
<tr>
<th>BANKNAME</th>
<th>RATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>American Bank</td>
<td>6.06</td>
</tr>
<tr>
<td>everbank</td>
<td>5.84</td>
</tr>
<tr>
<td>Security First Network Bank</td>
<td>5.83</td>
</tr>
<tr>
<td>Bank CarolLine</td>
<td>5.83</td>
</tr>
<tr>
<td>Presidential Savings Bank</td>
<td>5.13</td>
</tr>
<tr>
<td>Wingspan Bank</td>
<td>4.50</td>
</tr>
<tr>
<td>BankDirect</td>
<td>4.64</td>
</tr>
<tr>
<td>E*Trade Bank (formerly Telebank)</td>
<td>4.35</td>
</tr>
<tr>
<td>USAccess Bank</td>
<td>3.95</td>
</tr>
<tr>
<td>Net.B@nk</td>
<td>3.93</td>
</tr>
<tr>
<td>Legacy Bank</td>
<td>3.71</td>
</tr>
<tr>
<td>Bank of Wausau</td>
<td>6.13</td>
</tr>
<tr>
<td>Resource Bank</td>
<td>6.13</td>
</tr>
<tr>
<td>Nexity Bank</td>
<td>6.08</td>
</tr>
<tr>
<td>Corus Bank</td>
<td>6.02</td>
</tr>
<tr>
<td>Chase Manhattan USA</td>
<td>5.83</td>
</tr>
<tr>
<td>Abbotsford Bank</td>
<td>5.73</td>
</tr>
<tr>
<td>American Express</td>
<td>5.50</td>
</tr>
</tbody>
</table>

**Figure 10.** Aggregation Example.

**Figure 11.** Web Sources for the Aggregation.
5. Conclusions

In this project we created the Aggregator Tool that can be used for a wide variety of applications, from building customized “portals” for tracking market information, news, and stock prices, to monitoring an aggregation of prices for a particular item of interest from various on-line retailers, to managing personal finance portfolio, such as credit cards, bank, and investment statements that are available online, conveniently from one set of web pages. Once an aggregation application is built, it can be accessed from any browser. Users only need to use their Aggregator Tool login information to view the data extracted instantaneously from multiple web sources.

The Aggregator Tool follows the client-server paradigm and combines user project management and web wrapping facilities. The modular design of the tool means individual components such as web wrapping heuristics or different post-aggregation templates can be easily improved, extended, or added in the future.

The Aggregator Tool uses Caméléon web extraction engine that is designed to work as relational front-end to web sources and accepts regular expressions-based spec files defining the schema information and the data extraction rules. While it may seem that web wrapping for the purpose of data extraction would require prohibitive amount of spec file maintenance to keep them functioning properly due to fast-changing nature of the web, our experience has shown the opposite. In fact, over the course of more than a year we spent working on Caméléon, we observed that a great majority of spec files we created have continued working to this day without any modifications. This applied to a relatively wide variety of personal and business financial-related web sites, such as credit card, bank, and brokerage accounts information, stock quotes, etc. We believe this is due to the fact that while the façade of the web sites – their front page, graphics and GUI elements – changes fairly frequently, the information-carrying pages themselves remain mostly unchanged throughout the years.
6. Future Work

In the course of this project we implemented the initial version of the Aggregator Tool. In this chapter we discuss the ways in which future development can enhance this application.

As mentioned earlier, the registry database access layer should be moved to the servlet, and enhanced in the following ways. Currently, the user logon does not perform any authentication. Although the provided at logon username is used in the registry database in conjunction with projects and spec files information, user logon data is not validated. In the future, this task should be performed in combination with the database over a secure JDBC link from the servlet. In addition, the “business logic” of the registry data validation, responsible for avoiding name conflicts (i.e. user attempts to assign a duplicate name to a project or a spec file) should be added. This can be easily done by checking for JDBC SQL exception of a database constraint (primary key) violation. The appropriate error – for example, “duplicate project name entered” – should then be reported to the user.

In order to fully utilize the architecture of the system, Caméléon must be modified to take advantage of the registry database. In the long run, a single registry database will unite the registry implemented for the Aggregator Tool with many other types of information in the COIN system, such as versions, context definitions, etc [23]. In the very least, Caméléon should be able to fetch spec files at their locations as specified in the registry. More importantly, Caméléon should integrate user ID/password information for each spec file that uses them. There are currently two methods of providing user ID/password: storing them clear text in the spec file, or, if the spec file has been appropriately written, passing them as parameters in a query, such as:

SELECT rate FROM fleet WHERE userID = sasha AND password = pwd

which, for the Aggregator Tool projects, is equivalent to storing them clear text in the project JSP file. Clearly, neither method is terribly secure. Instead, Caméléon could securely obtain user ID/password from the registry and use them at the time of web extraction. To let Caméléon know which Aggregator Tool user is running a project, private project files could store username and pass it to Caméléon in a query; public project files, on the other hand,
should be constructed to query user for the information when invoked via URL. An alternative implementation could make a JavaBean the JSP project file calls query the database and dynamically append the queries with user ID/password by parameter-passing method illustrated above, if needed.

Another registry related enhancement should introduce the notion of public spec file / project owner. Only owners of the public files would be able to delete them. By default, creator of the public spec file or project would be its owner.

Current functionality of the projects is limited to the most basic aggregation. In the future, the component nature of the tool allows for many exciting and inventive types of projects to be added. In fact, there could be a library of pre-defined project templates offered to the user. Depending on project type selection, different elements would be loaded into the Project Display Settings area of the Aggregator Tool GUI. Programmatically, project’s JSP file would call a JavaBean class that implements the needed functionality. For example, a “template” for Universal Financial Application [24]-like project would allow users to select a particular attribute as the base for recommendations automatically selected among other attributes; for instance, “show me bank names with rates higher than the rate I am currently receiving.”

In fact, much of this functionality could come with the forthcoming OLE DB provider for the Caméléon Web Wrapper Engine [25]. The provider will expose the data extracted by Caméléon in the way that could be manipulated by the powerful commercial database engine such as SQL Server. User projects would then make use of the rich SQL capabilities to present extracted data in variety of customizable ways. XML offers additional – and similar – exciting possibilities. Since Caméléon is capable of XML output, the JavaServer pages could perform post-aggregation data transformations and display formatting on XML using XSLT [26] or other currently emerging technologies, and present it in a format appropriate for a variety of browsers, including those on wireless devices. An example of an application using these technologies is Jetspeed [27], which takes existing XML streams of data and formats them into a portal.
As mentioned earlier, the other component of the Aggregator Tool, the Web Wrapper, would see a significant improvement with Mozilla browser as both HTML rendering component and DOM parser. However, it would come at the expense of significant new development, so we will concentrate on several potential improvements to the existing implementation.

A disadvantage of the current pattern generating algorithms is their significant reliance on the underlying HTML of the web page. In order to alleviate this rigidity, which would cause the generated spec file to break easily when HTML changes, a somewhat arbitrary, perhaps empirical set of HTML tags that are skipped in particular kinds of pattern creations has been chosen. More experimentation in wrapping various web sites would be desired to come up with better choices, and possibly different heuristics – implemented as different visitors patterns – for particular kinds of web pages.

When the user fills out HTML form(s) or clicks on some buttons to get to the page which contains the extraction attributes, the form’s name-value pairs as well as the HTTP method (GET or POST) must be stored in the “source” portion of the spec file. This should be accomplished by obtaining the data from the JDK classes javax.swing.text.html.FormView and HTMLDocument.

Caméléon also supports multiple page traversal. For example, in some cases getting to the web page of interest requires following a dynamically generated link on the previous page. The URL of that link would be different every time if it contains, for example, a SessionID. To implement this, the user will have to supply the same extraction information (begin, end, and the link), and follow the link to the next page. If the user proceeds to define additional items to be extracted, the Web Wrapper should automatically know to create a slightly different, multiple-page traversal spec file.

An advanced Caméléon feature is ability to match multiple extraction patterns for the same attribute. This is very handy in cases when a single pattern would not successfully match the data of interest in every variation of some web page. Consider this example that extracts LastTrade stock price such as “106” or “106 ¼”: 
The Web Wrapper could automatically create such patterns when the user presents it with multiple extraction cases for a single (based on its name) attribute. Unfortunately, this requires user knowledge or observation that a spec file does not properly extract an attribute for some variations of the data on the web page. The Aggregator Tool has no automatic learning abilities. Finally, the last Caméléon feature to mention is its ability to substitute a parameter or extracted value into a regular expression in any part of the spec file. This would be very difficult to achieve automatically and as needed in the Web Wrapper; however, it is possible in special cases such as the one this spec file fragment illustrates:

Here, a hypertext link is extracted based on the country name parameter the user specifies in a query. Web Wrapper would be able to create a pattern like this if the user provides a "parameterize by link text" directive via a checkbox for each attribute.
Bibliography


[19] HTML Tidy Utility, W3C. http://www.w3.org/People/Raggett/tidy/


[22] E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, Massachusetts, 1995.


