

# 國立交通大學

電機資訊學院 資訊學程

## 碩士論文

網際網路自動化平台設計及其應用



An Internet Automation Platform for Creating Web Agents

研究生：嚴文亨

指導教授：柯皓仁 教授

楊維邦 教授

# **An Internet Automation Platform for Creating Web Agents**

Student: Wen Heng Yen    Advisors: Dr. Hao-Ren Ke, Dr. Wei-Pang Yang

Degree Program of Electrical Engineering Computer Science

National Chiao Tung University

## **ABSTRACT**

With the growth of the World Wide Web (WWW), many people nowadays spend a lot of time performing various tasks with browsers, most of them repetitive and tedious. Some services and tools were created to reorganize and simplify the usage of Web resources. This thesis discusses some of them, some of our previous work related with it, and proposes a complete solution to create Web automation services, the WIS (Web Integration Solution) system. It is a Web browser integrating various tools and technologies necessities to provide an environment for developing Web automation services. We show the versatility of WIS by implementing three exemplary services. The first is a metasearcher system, which provides a single search interface for various indexing and abstract databases. The second is a cataloguing tool to simplify the task of retrieving bibliographic data from the Web; this tool is also integrated with a recommender system for libraries. The third is a Web site checking system, which periodically check if these critical Web sites are working correctly, no matter how complex the check procedure is.

**Keywords:** User Surrogates, Web Agent, Web Browser

# 網際網路自動化平台設計及其應用

研究生：嚴文亨

指導教授：柯皓仁博士，楊維邦博士

國立交通大學電機資訊學院 資訊學程(研究所) 碩士班

## 摘要

網際網路的迅速發展使得透過網路進行資訊交換日趨頻繁，瀏覽器也成為使用者完成各種資訊取得與溝通等作業的重要工具。其中許多作業是具有重複性的並且涉及數個網站，因此網際網路自動化的服務應運而生，將現有的網際網路資源重新組合運用，例如整合檢索、售價比較服務、與企業內部系統的整合等，未來這類的應用也勢必隨著網際網路的成長而日形重要。

本研究探討網際網路自動化並且提出一個可以用以建立各種不同網際網路自動化應用的系統架構 WIS (Web Integration Solution)。本論文最後以三個網際網路自動化應用說明 WIS 系統之可行性：1.整合檢索服務，對數種不同線上資料庫提供單一介面；2.搭配圖書館圖書薦購系統之編目資料取得工具；3.網站服務自動檢查工具。

**關鍵字：**網際網路代理人，瀏覽器，網際網路自動化

## 誌謝

首先要感謝柯皓仁老闆讓我有機會多方面學習以至於能夠跨進另一個領域再次進修、楊維邦老師的耐心指導與照顧、黃明居老師的指點與協助。

二謝曾經與我合作的黃夙賢學長、余明哲以及其他多位實驗室的同學、學長、姊、弟。

三謝計畫室同事崇瑋分擔了我的工作、超然不時的幫助、與佳欣、雪卿、莉池的切磋機會、玉菱、怡君、媛媛、慧貞、蔡姐、馬姐等幾位與我一起度過許多有說有笑的日子。

四謝許多素未謀面的前進讓我能夠站在巨人的肩膀上。

五謝父母多年的照顧。

六謝我忘了感謝但應該感謝的人。



# CONTENTS

Abstract (English).....	I
Abstract (Chinese).....	II
Acknowledgements.....	III
CONTENTS.....	IV
CHAPTER 1 INTRODUCTION .....	1
1.1 Motivation.....	1
1.2 Objectives .....	2
1.3 Thesis Organization .....	2
CHAPTER 2 RELATED WORK .....	3
2.1 Web Automation Related Work .....	3
2.1.1 Web Automation Applications .....	3
2.1.2 Web Automation Creation Tools .....	5
2.2 Previous Systems Developed.....	10
2.2.1 Unisearch .....	10
2.2.2 Virtual Union Catalog System .....	16
2.3 Related Technologies .....	17
2.3.1 Extensible Markup Language .....	17
2.3.2 Document Object Model.....	17
2.3.3 Regular Expressions.....	18
2.3.4 Dynamic HTML.....	18
2.3.5 Component Object Model.....	19
2.3.6 Script Technologies.....	20
2.3.7 Web Interface Definition Language .....	22
2.3.8 Web Services.....	31
CHAPTER 3 THE WIS PLATFORM .....	39
3.1 System Overview .....	39
3.2 Programming Basics .....	45
3.2.1 Programming Contexts .....	47
3.2.2 The WIS Surfing Process.....	48
3.2.3 Profile Management.....	49
3.2.4 The WIS API.....	50
3.3 Data Elements Extraction .....	52
3.4 Communicating With the Server.....	54
CHAPTER 4 WIS EXAMPLE APPLICATIONS .....	56
4.1 Unisearch 2 .....	56
4.1.1 Introduction.....	56

4.1.2 Application Overview .....	56
4.2 Book Recommendation System for Library .....	59
4.2.1 Introduction.....	59
4.2.2 Application Overview .....	60
4.3 Web Site Checking System.....	63
4.3.1 Introduction.....	63
4.3.2 Application Overview .....	63
CHAPTER 5 CONCLUSION and FUTURE WORK.....	66
5.1 Conclusion .....	66
5.2 Future Work .....	67
BIBLIOGRAPHY .....	70



## LIST OF FIGURES

Figure 2-1: The need for Web automation .....	5
Figure 2-2: Unisearch provides a single interface for multiple resources ...	11
Figure 2-3: Query template example .....	13
Figure 2-4: Unisearch Architecture.....	14
Figure 2-5: Selection of database sources for Unisearch.....	15
Figure 2-6: Search interface for Unisearch.....	15
Figure 2-7: Return of search results using Unisearch.....	16
Figure 2-8: Extraction of data elements with regions.....	25
Figure 2-9: Java Stub .....	29
Figure 2-10: The Web Service Solution.....	37
Figure 3-1: Multiple Processes in a Metasearch.....	41
Figure 3-2: Web Automation in a Three Tier Model.....	42
Figure 3-3: Three Tier Model with WIS .....	43
Figure 3-4: Structure of a WIS Surfing Process .....	48
Figure 4-1: Unisearch 2 search interface .....	57
Figure 4-2: Results returned by Unisearch 2 .....	58
Figure 4-3: Collected results using Unisearch 2.....	59
Figure 4-4: Conventional book recommendation process .....	60
Figure 4-5: Automated book recommendation process .....	60
Figure 4-6: Submitting the book URL.....	61
Figure 4-7: Extract metadata from the recommended URLs.....	62
Figure 4-8: Reader's recommendation list.....	62
Figure 4-9: The site checking system main page.....	64
Figure 4-10: After checking the sites .....	65
Figure 4-11: An error notification.....	65
Figure 4-12: Report of test results .....	65

## LIST OF TABLES

Table 2-1: Search fields supported by different resources .....	12
Table 2-2: Advanced search in the command-line format .....	13
Table 3-1: Summary of programming contexts .....	48
Table 3-2: Objects in WIS.....	51
Table 5-1: Design considerations and WIS support.....	67





---

# CHAPTER 1 INTRODUCTION

---

## 1.1 Motivation

The World Wide Web (WWW) provides a vast amount of information and plentiful services, and continues to grow at a staggering rate. Because of its explosive growth, people are spending more and more time on the Web, performing various tasks, many of these tasks repetitive and tedious. Here are some typical scenarios:

- I daily check several sources, including online news, Usenet newsgroups, and bulletin board systems (BBS) for some specific topics.
- Some WWW sites recommend several books to me, but I don't know where the best places to retrieve them are.
- I want to buy a second-hand notebook that fits some requirements. There are several places where I could find them and I don't want to miss any good buy.
- I want to find data about an interesting topic, but the related resources are scarce and I need to try every possible search engine.
- I daily track the stock market using some specific rules.
- I search a dozen online databases for news articles about some topics and spend much time organizing the data gathered.

From the above scenarios, it can be concluded that: (1) many tasks are repetitive; (2) user may not know where to start surfing in the Internet; (3) many tasks involve several resources in different sites. What we need in these scenarios are agents, or Web automation tools, which act as our surrogates to perform these laborious tasks. A good surrogate should have the following benefits: (1) let nontechnical users be able to exploit the information available on the Web without being overwhelmed by

technical detail; (2) free users from repetitive browsing tasks; (3) reformat and recombine the information from various Web sites to best fit a user's task.

## 1.2 Objectives

There are many applications that use data from the Web, but they are usually developed for specific purposes. Instead of creating from scratch every time we need a new type of Web automation application, it will be helpful to have a tool specialized for the creation of such kind of applications. The objective of this thesis is to (1) identify the common needs of Web automation applications; (2) create an architecture, which is called WIS in this thesis, that integrates these essentials together in a single tool suitable for the creation of a wide range of Web agent services; (3) provide technological solutions for the challenges imposed by Web automation tasks; (4) construct some applications to show the feasibility of this architecture.

## 1.3 Thesis Organization

This thesis is organized as follows:

- Chapter 1 introduces the motivations and objectives.
- Chapter 2 presents several web automation applications, web automation creation tools, our previous developed systems, and technologies used in the WIS platform.
- Chapter 3 depicts common needs of web automation applications and explains the new WIS platform proposed by this thesis.
- Chapter 4 gives three example applications created using WIS.
- Chapter 5 finalizes this thesis by a conclusion and provides some ideas about future work.

---

## CHAPTER 2 RELATED WORK

---

### 2.1 Web Automation Related Work

#### 2.1.1 Web Automation Applications

We define Web automation as having user surrogates operating on existing Web resources to simplify users' tasks. It can be classified as software agents, but not necessarily intelligent - many commercial products still don't have the capability to develop knowledge of the user's needs and of subject domains. Many applications were created to act as user surrogates, and we discuss some of them as examples to clarify the scope of what we mean by Web automation, what can be done.

Search is a very common task in the Web. A metasearcher is an application that helps users perform search on multiple search engines. It provides a single interface for the target search engines, and let the user search these targets simultaneously with the same query. Many types of metasearchers exist today, with different capabilities (search ability, results display, and so on) and purposes. The Metacrawler[6] is a metasearcher that searches general-purpose Web search services such as Lycos and Google. Our previous works, the Unisearch and VUCS, search different target collections: the first, abstract online databases and the second, online public access catalogs of multiple libraries. Metalib[22] is a library portal that lets institutions manage hybrid information resources under one umbrella. Such resources may be catalogs, reference databases, digital repositories, and subject-based Web gateways. In conjunction of resource management and personalization features, there is metasearcher capability. Metasearchers are not necessarily to work only in the server side; some were created as desktop applications. The Copernic Agent[7] is an example of such a desktop metasearcher application. As source Web sites tend to

change over time in many aspects, profiles of Web sites need to be updated to keep them accessible. Installed Copernic Agents keep up to date by downloading these profiles periodically from the Copernic Web site.

Price comparison sites collect data from various online stores and produce a report for the buyer about the interested good. BestWebBuys[19] covers several kinds of products such as books, music, video, electronics and bikes, and uses dozens of stores as sources. BestBookDeal[20] focuses only on books, searching and comparing prices among 61 online bookstores all over the world to make sure that the buyer gets the best price. It claims to get real time information of book, price, shipping cost, shipping time, sales tax and availability, saving the user from searching every online bookstore.

Readerware[21] is a useful tool to catalog and maintain a library and has some features that interact with the Web. The auto-catalog feature can search the Internet and automatically catalog the books you own. The cover art support automatically extracts cover art from Web sites and adds the images to your database. The integrated shopping cart provides online ordering, browsing and searching of all the major online retailers around the world. An integrated browser lets you catalog your collection as you surf the Web. Additional features not related with Internet interaction but important for the task are bar code reader support, database to store and track library holdings and many others, which in the same bundle, make sure the completeness of the application with regard to its purpose.

As we can see from the examples given, there are a plenty of user surrogates that work on existing Web resources, as what we call Web automation here. Many applications are still to be realized or discovered, and some tools were developed to facilitate the creation of such surrogates. In the following section we discuss some

tools related with Web automation and their designs.

## 2.1.2 Web Automation Creation Tools

### 2.1.2.1 Web Interface Definition Language

Many business systems are available that transform the Web browser from an occasionally informative accessory into an essential business tool. Business units that have previously been unable to agree on middleware and data interchange standards for direct communication are agreeing on communication through HTTP and HTML, which needs human intervention (Figure 2-1). The need of manual operation may become highly inefficient when a lot of transcription or copy and paste operations are part of the daily job. The goal of the Web Interface Definition Language (WIDL)[10][11] is to enable automation of interactions with HTML/XML documents and forms, accepting the Web to be utilized as a universal integration platform without efficiency problems.

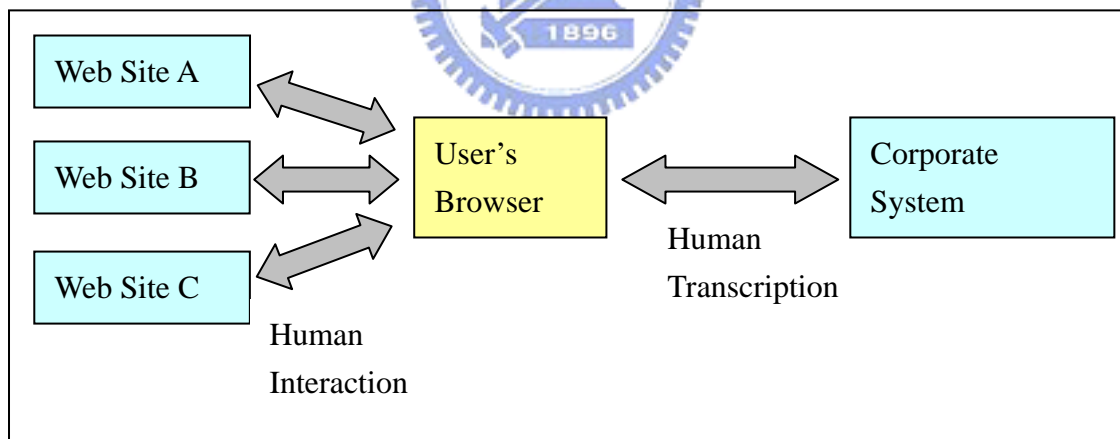


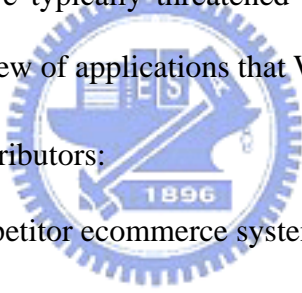
Figure 2-1: The need for Web automation

WIDL uses the XML standard to define interfaces and services, mapping existing Web content into program variables, allowing the resources of the Web to be made available for integration with business systems. It brings to the Web what is similar in IDL concepts that were implemented in standards such as CORBA for distributed computing. WIDL describes and automates interactions with services hosted by Web

servers on intranets, extranets and the Internet; it provides a standard integration platform and a universal API for all Web-enabled systems.

A service defined by WIDL is equivalent to a function call in standard programming languages. What WIDL defines is how to “Make a call” for a Web service. To make the call, it defines the location (URL) of the service, input parameters to be submitted, and output parameters to be returned. Note that like IDLs, a standard programming language is needed for further processing of the data, and a browser is not required any more.

The use of standard Web technologies empowers various IT departments to make independent technology selections. This has the effect of lowering both the technical and political barriers that have typically threatened cross-organizational integration projects. Here is a brief overview of applications that WIDL enables:

- 
- Manufacturers and distributors:
    - Access supplier and competitor ecommerce systems automatically to check pricing and availability
    - Load product data (specification sheets) from supplier Web sites
    - Place orders automatically (i.e. when inventory drops below predetermined levels)
    - Integrate package tracking functionality for enhanced customer service
  - Human resources:
    - Automated update of new employee information into multiple internal systems
    - Automated aggregation of benefits information from healthcare and insurance providers
  - Governments:
    - Kiosk systems that aggregate data and integrate services across departments or state and local offices

- Shipping and delivery services:
  - Multi-carrier package tracking and shipments ordering
  - Access to currency rates, Customs regulations, etc.

Shipping Companies were early leaders in bringing widely applicable functionality to the Web. Web-based package tracking services provide important logistics information to both large and small organizations. Many organizations employ people for the sole purpose of manually tracking packages to ensure customer satisfaction and to collect refunds for packages that are delivered late. Integrating package tracking functionality directly into warehouse management and customer service systems is a huge benefit, boosting productivity and enabling more efficient use of resources.

Using WIDL, the Web-based package tracking services of numerous shipping companies can be described as common application interfaces, to be integrated with various internal systems. In almost all cases, programmatic interfaces to different package tracking services are identical, which means that WIDL can impose consistency in the representation of functionality across systems.

#### **2.1.2.2 WebVCR[1]**

When the Web becomes more interactive, personalized and rich in content, increasing complexity of manipulation seems to be inevitable. The result is that users are forced to go through several steps and fill out a sequence of forms before reaching the desired results. For example, consider using a travel site. The steps are: (1) Go the travel site url; (2) Choose the Find/Book a Flight option; (3)Login; (4) Fill a form with the details of itinerary. However, it is likely that a single visit to the results page will be insufficient. It may take weeks or months to find an acceptable fare, and the process need to be repeated every time to reach the results page. What WebVCR do is

to record the browsing steps and replay it later, as many times the user needs. The saved sequence is called a smart bookmark, which differs from conventional bookmarks that can only save one-step reachable pages.

There are two different implementation architectures, client-based and server-based. In the client based version, the WebVCR is implemented as a Java applet that runs with the user's browser. The user starts the WebVCR by loading the WebVCR starting page into a browser window (main window), which immediately opens another browser window (applet window) to load the HTML page containing the WebVCR applet, making it persistent during record and play sessions. To record a smart bookmark, the user traverses the Web to the desired starting point and clicks on the Record button in the applet. Clicking on the Record button causes two actions to take place (which are transparent to the user): (1) the applet records the current URL as the starting location of the smart bookmark; (2) the applet inserts event handlers on all elements in the main window to capture whatever the user may do. From then on, as the user navigates via link traversals or form submissions, each action triggers the inserted event handler that causes the applet to record the corresponding action. When the user finally reaches the desired page, he clicks on the Stop button and WebVCR stops recording.

In the server-based version, the implementation is much more complex. Many issues that a browser could handle need to be implemented in the server. Javascript handlers are no longer a valid option for detecting browsing actions. Cookie handling and https connections claim additional work in the server part.

### **2.1.2.3 LiveAgent[1]**

LiveAgent allows developers to create Web automation agents by recording the browsing process. While a developer records an agent, LiveAgent's agent engine, also



called the AgentSoft proxy, intervenes between the browser and the Web by altering the Web pages being browsed so that user events can be monitored and recorded. The proxy monitors browsing sessions and inserts appropriate code into browsed Web pages. This involves adding and routing event handlers for whatever the user may change, such as input fields, links and buttons. To keep persistent data across pages, a hidden frame is needed, with the main frame used for browsing.

Whenever the user records an event of the browsing process, the proxy must try to understand the user's action for future replay. For every event, a window pops up asking the user to specify their intentions, which allows the proxy understand the user action. For example, for positioning a hyperlink, it could be understood as clicking on the fifth link on the page, or the link with word "profile" in the link anchor text. Additional flexibility is given to the recorded process by defining parameters for the whole process and conditional branching. For example, "If it is Sunday, follow the link to the cross-word puzzle and retrieve it." If it is not Sunday, then the link is simply not followed and another node is visited. There can also be loops with tasks "Download stories while there are still stories about the Internet" and interdependencies with conditions "If the Sunday puzzle was downloaded, then download the sports section too." The definitions given are kept by using AgentSoft's HTML Position Definition Language (HPD). Result extraction and report format definition is also part of the HPD language.

The MasteAgent tool is another part of the package. It combines several created agents with LiveAgent to collect information in parallel and then merges the information into a single report. Further processing on the results requires the developer to use java.

#### **2.1.2.4 Internet Scrapbook[3]**

Internet Scrapbook automates users' daily browsing tasks using a programming by demonstration technique. In Scrapbook, users can demonstrate in which portions of Web pages they are interested by selecting them on a Web browser. Once the personal page is created, the system automatically updates it by extracting the user-specified portions from the newest Web pages. Thus, the user can browse only the necessary information on a single page and avoid repetitive access to multiple Web pages.

Scrapbook generates a matching pattern when the user selects the desired data from the Web browser. Therefore, the pattern should contain information that is expected to remain constant even after the source page has been modified. It uses two kinds of descriptions to define matching patterns: heading pattern and tag pattern.

Heading pattern assumes that headings such as "Top News" and "Economy" are preserved in the news page while the articles following these headings keep changing. Scrapbook assumes that when the user selects data, the previous line, the first line and the first line after the selection area are permanent headings. The tag pattern represents the position of the selected data in the Web page by using the HTML elements.

## **2.2 Previous Systems Developed**

### **2.2.1 Unisearch**

The Consortium on Core Electronic Resources in Taiwan (CONCERT)[28] provides plentiful online databases, but a user may not be aware of which one he/she should use. Different search interfaces of different resources even make the search more difficult since the user must learn how to use every type of interface. To increase the simplicity to use CONCERT resources, Unisearch was created (Figure 2-2).

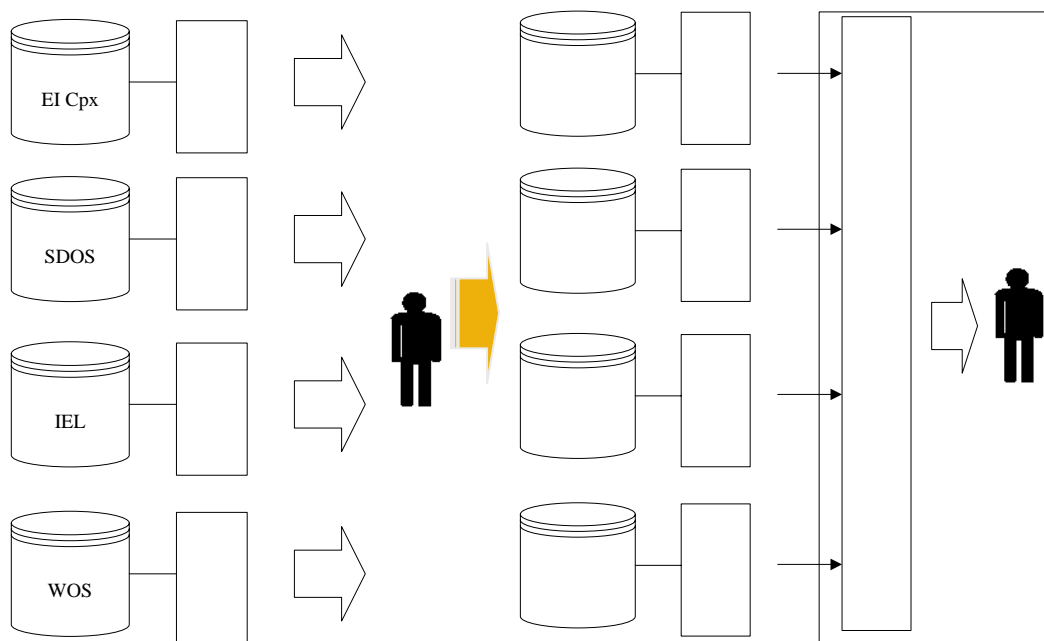


Figure 2-2: Unisearch provides a single interface for multiple resources

Unisearch is a metasearcher specially designed to take into account the constraints of CONCERT resources:

1. Although there are some protocol standards such as Z39.50 for distributed search, many resources do not provide such facility or need additional cost to purchase a separated module for compliance of the wanted protocol. Using a well-known protocol standard of the library community is not feasible.
2. It is aimed to satisfy CONCERT needs, and the scale was not sufficient to put resource providers working together to agree with following a common communication protocol.
3. All resources were not free, so access policies should be respected or some agreement should be done with every resource provider if special access privileges are needed.
4. Usage statistics is important and should not be misguided by the usage of Unisearch.
5. Interface of resources may change.

Issues 1 and 2 constrain the design to rely only on html and the http protocol. These protocols do not constrain much about performing search such as Z39.50 does, so the interface of every resource needs to be analyzed. A wrapper is created for every resource, which transforms the user's query to a form acceptable by the target

resource. Resources have different search capabilities (Table 2-1).

Search Field Resource	Abstract	Author	Journal Name	Title	ISSN/ISBN	Year of publication	Subject Heading	Keyword
Ovid	●	●	●	●	●	●	●	●
Ebsco	●	●	●	●	●		●	●
Proquest	●	●	●	●	●	●	●	●
SDOS	●	●	●	●	●		●	●
Ei CPX	●	●	●	●			●	●
IEL	●	●	●	●			●	●
Gale		●		●		●	●	●
CSA	●	●	●	●	●		●	●
First Search		●		●	●		●	●
IDEAL	●	●	●	●			●	●
SwetsNet	●	●	●	●			●	●
WOS	●	●	●	●			●	●

Table 2-1: Search fields supported by different resources

Most resources support advanced search which allow users create complex queries in a single command-line format. The command-line format makes the translation of queries simpler than mapping to html controls; some examples are given in Table 2-2. If the command-line format is not supported, then search terms need to be mapped to respective html controls. A query template mechanism is used to eliminate resource-specific definitions from the program code. An example of the template is given in Figure 2-3.

欄位名稱 資料庫名稱	Ovid	Ebsco	Proquest	Ei CPX
Abstract	ab	AB	ABS(customer delight)	AB
Author	au	AU	AUTHOR(Gertrude Enders Huntington) AU(Michael Kinsley)	AU
Journal Name	jn			ST

Source		SO	SO(chicago tribune) JO(computing)	
Title	ti	TI	TITLE(Future) TI(future AND career)	TI
ISSN	is	IS	ISSN(0011-4664) SN(00916358)	
Text				
Description				
Subject Heading	sh	SU	SUB(Music) SU(Health Care)	CV
Keyword				
Year	yr		DATE(Dec 1994) DA(December 14 1995) YR(oct AND 1996)	
検索例子	Root journal of brain.jn.	AU Jefferson	SUB(Music) and	solar cycle within AB

Table 2-2: Advanced search in the command-line format

---

```

<form name=formSDOS action="http://sdos.ejournal.ascc.net/cgi-bin/search.pl" method=get>
<input type=hidden name=collection value=journals>
<input type=hidden name="GetSearchResults" value="Search">
<input type=hidden name=search_field value="#Query_Phrase#">
<input type=hidden name=fields value="Any">

```

---

Figure 2-3: Query template example

Issues 3 and 4 infer that resource providers should be aware of who is using the resource even if the search is submitted by a metasearcher system. Most resources use the IP address of the user's browser to perform identification. In a traditional server solution, the server is responsible for connecting with the target resource, which makes it unaware of the user's IP address and misguides the access control and statistic mechanisms of the target resource. Unisearch instead creates the connection from the user's computer, not from the server, avoiding the problem.

Issue 5 requires that Unisearch uses some strategies to simplify adaptation to changes. When a resource changes, adaptation is achieved by modifying the template (Figure 2-3) and the profile of the resource, not the program itself, thus separating the volatile part from the code for convenient update. The information is kept in a server and users retrieve them every time it performs a search.

Figure 2-4 is the Unisearch Architecture. When a user uses the Unisearch System, the User Identifier module identifies the user and provides a list of databases that the user can use. The Database Selection Interface is then displayed as Figure 2-5, in which the user can select the target resources to search. The Search Interface lets the user input the query (Figure 2-6).

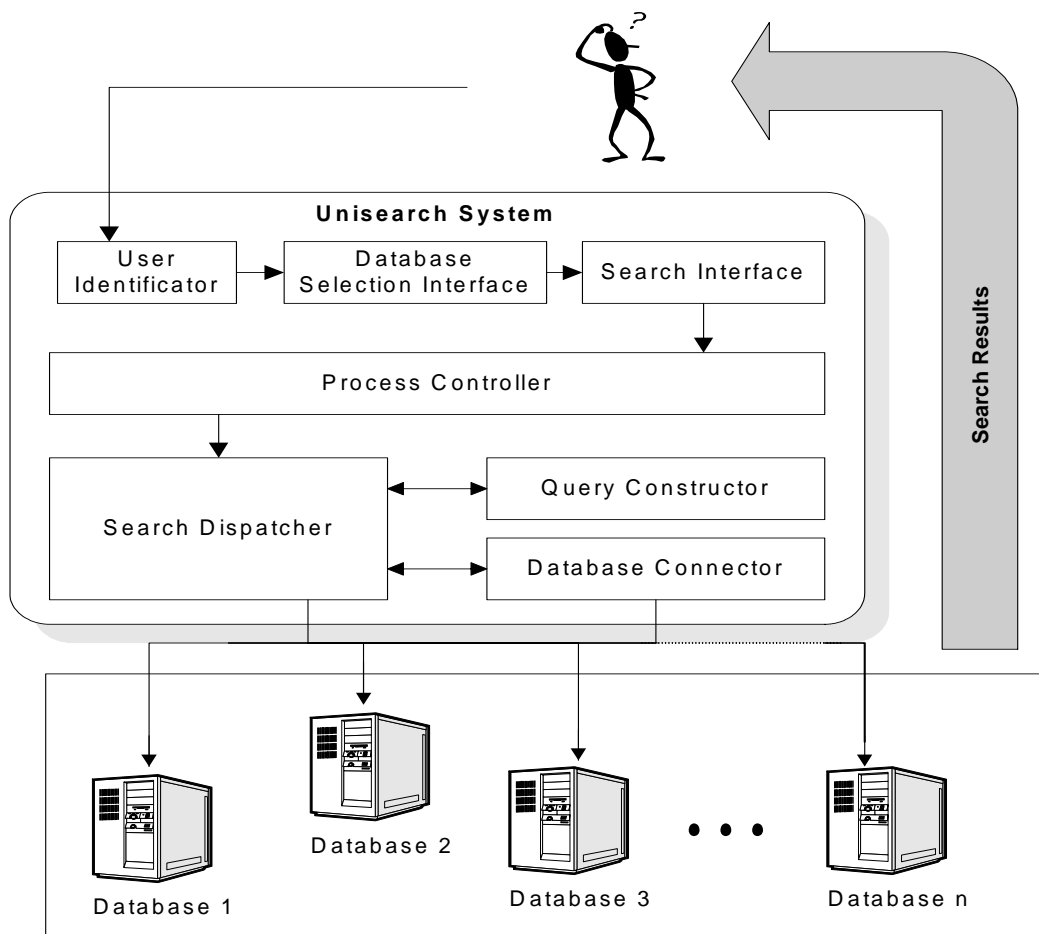


Figure 2-4: Unisearch Architecture



Figure 2-5: Selection of database sources for Unisearch

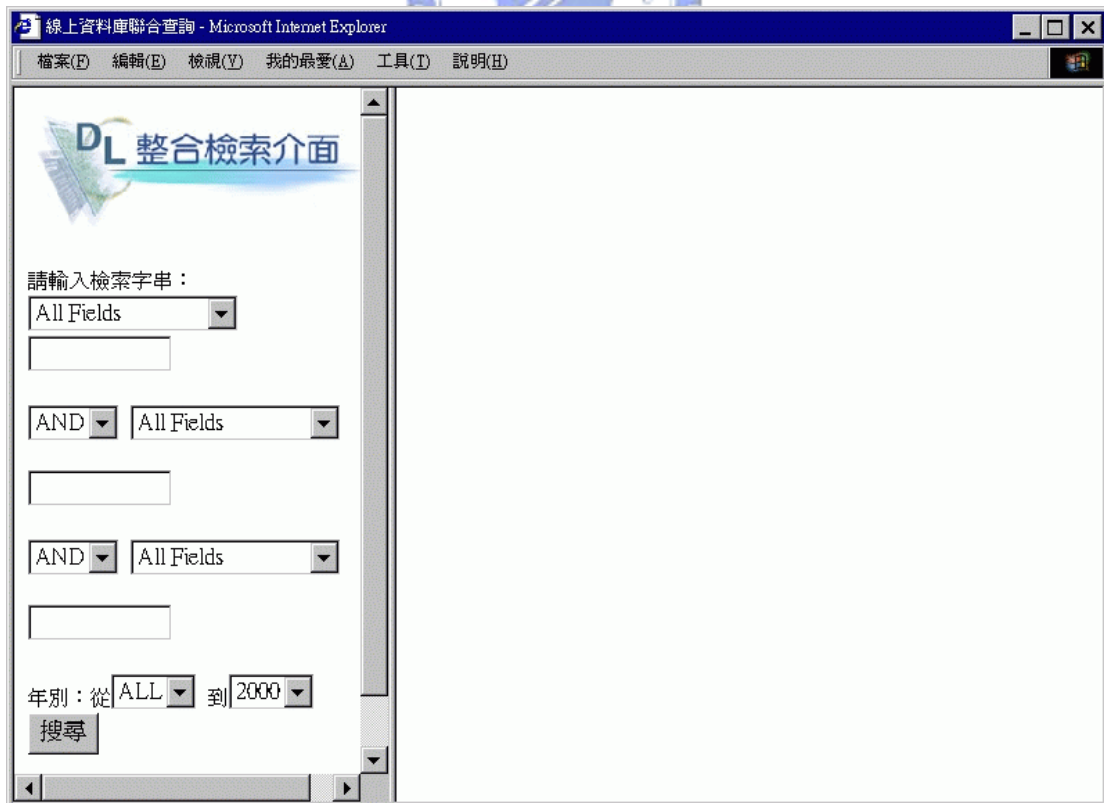


Figure 2-6: Search interface for Unisearch

The Process Controller module creates several instances of the Search Dispatcher object, according to the number of selected databases. The Search Dispatcher loads the target resource profile, and uses it to translate the user's query into the command line format acceptable by the target resource. The Database Connector module loads the query template and uses it to send the translated query to the respective target resource. Results are displayed as Figure 2-7.



Figure 2-7: Return of search results using Unisearch

## 2.2.2 Virtual Union Catalog System

Interlibrary loan help libraries compensate the deficiencies of their collections, but



without a unified catalog it is very inconvenient: users need to search the online public access catalog of every potential library. A unified catalog can be constructed by periodically harvesting information about the holding of the subject libraries, but the effort needed is great and users will eventually suffer some misses during periods. VUCS is a metasearcher designed to solve the latency problem and the periodical update effort of a unified catalog.

## **2.3 Related Technologies**

### **2.3.1 Extensible Markup Language**

XML is a markup language for documents containing structured information. Structured information contains both content and some indication of what role that content plays, so it identifies structures in a document. XML does not define the semantics or the tags; it is a metalanguage, i.e. a language for describing other languages, which lets you design your own customized markup languages for limitless different types of documents.

### **2.3.2 Document Object Model**

The W3C Document Object Model is a platform and language neutral interface that allows programs and scripts to dynamically access and updates the content, structure and style of documents.

The goal of DOM is to define a programmatic interface for markup languages such as XML and HTML. The DOM architecture is divided into modules. Each module addresses a particular domain. Domains covered by the current DOM API are XML, HTML, Cascading Style Sheets (CSS), and tree events. The Core DOM provides a low-level set of objects that can represent any structured document. While by itself this interface is capable of representing any HTML or XML document, the core interface is a compact and minimal design for manipulating a document's contents.

Depending upon the DOM's usage, the core DOM interface may not be convenient or appropriate for all users. The HTML and XML specifications provide additional, higher level interfaces that are used with the core specification to provide a more convenient view into a document. These specifications consist of objects and methods that provide easier and more direct access into the specific types of documents.

The Document Object Model provides a standard set of objects for representing HTML and XML documents, a standard model of how these objects can be combined, and a standard interface for accessing and manipulating them.

### **2.3.3 Regular Expressions**

Regular expressions are a very powerful text parsing language used widely in many applications. Their main use is to find a particular pattern within a given string that matches whatever rules expressed using this language. Regular expressions can be considered to be a generalized form of substrings. The alphanumeric characters retain their meaning, but some other characters become special and allow one to construct more general "substrings" to match. For instance, foo is a simple regular expression, only matching the pattern foo. But the power of regular expressions come from metacharacters such as the "." character. It matches any character, so "foo." will match "food", "fool" and "foot". If you wish to use a metacharacter as a real character, quote it with "\" and can still be used.

### **2.3.4 Dynamic HTML**

Dynamic HTML (DHTML) builds upon existing HTML standards to expand the possibilities of Web page design, presentation, and interaction. The basic notion behind DHTML is to allow any element of a page to be changeable at any time. Without it, any modification requires a post trip with the server, i.e. requires a request

to a server to perform the changes to the page, reconstructs the entire page in the server with the modifications and then sends everything back to the client. While workable, this process is quite slow, as it places a burden on both network traffic and server processing time. With long delays between a user's action and an on-screen response, building effective Web-based applications is quite constricting.

DHTML allows modifications occurring entirely on the client-side. This means that page modifications should appear immediately following a trigger, such as a user selection. For this to occur, it is more about scripting than HTML, the markup language. DHTML describes the abstract concept of breaking up a Web page into manipulable elements and expose those elements to a scripting language that can perform the manipulations.

DHTML itself is not a language. In practice, one programs DHTML by combining HTML, Cascading Style Sheets, and Javascript. To allow Javascript working with HTML/CSS, the Document Object Model describes each page element and the characteristics of which may be modified in an object-oriented fashion.

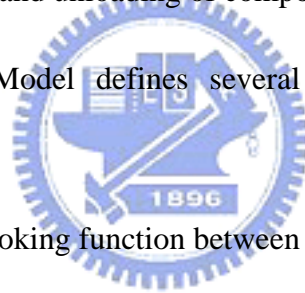
### **2.3.5 Component Object Model**

The Component Object Model (COM) is a Microsoft technology to software components. COM is the underlying architecture that forms the foundation for higher-level software services, like those provided by OLE (Object Linking and Embedding). OLE services span various aspects of commonly needed system functionality, including compound documents, custom controls, interapplication scripting, data transfer, and other software interactions. These services provide distinctly different functionality to the user. However they share a fundamental requirement for a mechanism that allows binary software components, derived from any combination of pre-existing customers' components and components from

different software vendors, to connect to and communicate with each other in a well defined manner. This mechanism is supplied by COM, which have the following characteristics:

- Defines a binary standard for component interoperability
- Is independent of any programming language
- Is extensible by developers in a consistent manner
- Uses a single programming model for components to communicate within the same process
- Provides rich error and status reporting
- Allows dynamic loading and unloading of components.

The Component Object Model defines several fundamental concepts. These include:



- A binary standard for invoking function between components.
- A provision for strongly-typed groupings of functions into interfaces.
- A base interface providing: (1) A way for components to dynamically discover the interfaces implemented by other components. (2) Reference counting to allow components to track their own lifetime and delete themselves when appropriate.
- A mechanism to identify components and their interfaces uniquely, worldwide.
- A “component loader” to set up component interactions and to help manage component interactions.

### **2.3.6 Script Technologies**

Script languages are complete but simple programming languages that can be used

create applications within the context they are designed for. It is complete in that it provides you with a set of tools that you can use to accomplish any reasonable task in the target context, such as in the browser or in the operating system. It is simple enough so that the learning curve is not too steep. So script languages provide quick solutions.

Scripting languages are interpreted rather than compiled. A scripting environment provides a runtime engine (often called a parser) that processes instructions on the fly. In contrast, other programming languages (e.g. C++) must be compiled into a set of machine instructions to become executable. A compiled language requires a far more complex development environment but executes faster.

In Windows OS, script engines works tightly with COM, although a script programmer do not need to understand it unless if he/she wants to extend the script language itself with more objects. Microsoft calls it scripting technologies. Each software component that complies with COM's set of rules is a COM object. The functionality that each COM object makes available externally is organized into groups called interfaces. Automation objects are a special type of COM objects that allow their interfaces available to script engines through a basic interface called IDispatch. So, a scripting technology is a service that a component makes available to scripts through automation.

Scripting technologies provide access to a set of functions that are expressed in terms of objects, methods, and properties. The set of functions is often referred to as an object model. An object model is a hierarchy of logically related objects, each with a set of methods and properties. A script then can access those objects and invoke their methods and properties through the automation object's interface.

## 2.3.7 Web Interface Definition Language

An introduction about WIDL was given in section 2.1.2.1 .Here we introduce the nuts and bolts of the WIDL language.

The WIDL definition is stored in an ASCII file, which is utilized by client programs at runtime to determine both the location of the service (URL) and the structure of documents that contain the desired data. Client programs access WIDL definitions from local files, naming services such as LDAP, HTTP servers or other URL access schemes, allowing centralized management of WIDL files. Unlike the way CORBA and DCE IDL are normally used, WIDL is interpreted at runtime. As a result, Service, Condition, and Variable definitions within WIDL files can be administered without requiring modification of client code. This usage model supports application-to-application linkages that are more robust and maintainable than if they were coded by hand.



There are three models for WIDL management:

- Client side: where WIDL are collocated with a client program
- Naming service: where WIDL definitions are returned from directory services
- Server side: where WIDL are collocated or embedded within Web documents

Except for being expressed in XML, WIDL specifications closely correlate to existing IDLs. One significant difference is the notion of a WIDL record. A WIDL service may specify input or output variables within a particular interface.

The Web Interface Definition Language (WIDL) consists of six XML tags:

- <WIDL> defines an interface, which can contain multiple services and binding
- <SERVICE> defines a service, which consist of input and output bindings
- <BINDING> defines a binding, which specifies input and output variables, as

well as conditions for successful completion of a service

- <VARIABLE> defines input, output, and internal variables used by a service to submit HTTP requests, and to extract data from HTML/XML documents
- <CONDITION> defines success and failure condition for the binding of output variables; specifies error
- <REGION> defines a region within an HTML/XML document; useful for extracting regular result sets which vary in size, such as the output of a search engine, or news

One of the most important features of WIDL is the capability to reliably extract specific data elements from Web documents and map them to output parameters. Two candidate technologies for data extraction are pattern matching by regular expressions or pattern matching by tag patterns. Regular expressions are well suited to raw text files and poorly structured HTML documents. Tag patterns instead rely on the tag structure of the document and needs parse of the document. The parsed document structure exposes relationships between document objects, enabling elements of a document to be accessed with an object model, described in section 2.3.2 . Using an object model, an absolute reference to an element of an HTML document might be specified:

`doc.p[0].text`

This reference would retrieve the text of the first paragraph of a given document.

From both a development and an administrative point of view, pattern matching is more labor intensive to establish and maintain. Regular expressions are difficult to construct and prone to breakage as document structures change. For instance, the addition of formatting tags around data elements in HTML documents could easily

derail the search for a pattern. An object model, on the other hand, can see through many such changes.

The <VARIABLE> element is used to describe input and output binding parameters. Common attributes are:

- NAME: Required identifier for calling programs.
- TYPE: Required. Specifies both the data type and dimension (for arrays) of the variable.
- REFERENCE: Optional. Specifies an object reference to extract data from the HTML, XML, or text document returned as the result of a service invocation.
- MASK: Optional. Masks permit the use of pattern matching and token collection to easily strip away unwanted labels and other text surrounding target data items.

The <REGION> element is used in output bindings to define targeted subregions of document. This is useful in services that return variable arrays of information in structures that can be located between well known elements of a page.

Regions are critical for poorly designed documents where it is otherwise impossible to differentiate between desired data elements (for instance, story links on a news page) and elements that also match the search criteria.

- NAME: Required. Specifies the name for a region. This name can then be used as the root of an object reference. For instance, a region named foo can be used in object references such as:

foo.p[0].text

- START: Required. An object reference that determines the beginning of a region.
- END: Required. An object reference that determines the end of a region.



```

<WIDL NAME="News" VERSION="2.0">

<SERVICE NAME="TechWebOut" METHOD="GET" URL="http://www.techWeb.com"
      OUTPUT="techWebOut">

<SERVICE NAME="TechWeb" METHOD="GET"
      URL="http://www.techWeb.com/" OUTPUT="techWebOut">

<BINDING NAME="techWebOut" TYPE="OUTPUT">
  <REGION NAME="tops" START="doc.font['Last?Updated*']"
        END="doc.b['For?more*']" />
  <VARIABLE NAME="service" TYPE="String" VALUE="TECHWEB Top Stories" />
  <VARIABLE NAME="url" TYPE="String" REFERENCE="doc.url" />
  <VARIABLE NAME="stories" TYPE="String[]" REFERENCE="tops.a[].text" />
  <VARIABLE NAME="links" TYPE="String[]" REFERENCE="tops.a[].href" />
</BINDING>

</SERVICE>
</WIDL>

```

Figure 2-8: Extraction of data elements with regions

Figure 2-8 demonstrates the use of regions in a news service, where the number of news stories varies day to day. Regions permit the extraction of data elements relative to other features of a document. The tops region begins with a text object that matches the pattern ‘Last Updated’ and ends with an object that matches ‘For more\*’.

Variable references into the tops region collect arrays of anchors and anchor text, regardless of the fact that the sizes of the arrays change throughout the day. The object references within tops are vastly simplified by the processing already provided by the region definition:

```
tops.a[].text
```

tops.a[0].href

## Object References

The default object model used by WIDL provides object references for accessing elements and properties of HTML and XML documents. This model is based on the DOM object model in JavaScript, but without the JavaScript method definitions.

Using the default object model, all elements of HTML and XML documents can be addressed in the following ways:

- By name: if the target element has a non-empty name attribute, it can be used in the reference. For example, the value of an HTML element `<a name="foo">` can be referenced:

doc.foo.value

- By absolute indexing: each array of elements has a zero-based integer index, i.e.:

doc.headings[0].text

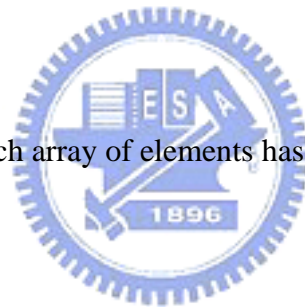
doc.p[1].text

- By relative indexing: directs the binding algorithm to search the VALUE attributes of each element in the array, until a match is found. The match must be complete, which requires the use of wildcard metacharacters for partial string matches. Note that the search will return the first matching element, if any:

doc.tr[\*pattern\*].td[1].text

- By region indexing: directs the binding algorithm to search an object's attributes until a match is found. Attribute matching is done with parenthesis instead of square brackets:

doc.a(name='foo').href



The following properties are available for all objects. To return the text of a container:

.text/.txt

To return the value of a container:

.value/.val

To return the source of a container:

.source/.src

To return the index of a container:

.index/.idx

To return the fully qualified object reference

.reference/.ref



### **Putting WIDL to Work**

WIDL files can be hand-coded or developed interactively with command line or graphical tools, which provide aid for determining object references used in <VARIABLE>, <CONDITION>, and <REGION> declarations.

Once a WIDL file has been created, its use depends upon the implementation of products that can process and understand WIDL services. A Web integration platform based on WIDL needs to provide:

- A mechanism for retrieving WIDL files, either from a local file system, a directory service such as LDAP, or a URL
- An HTML and XML parser, and text pattern matching capabilities, providing and object model for accessing elements of Web documents.

- HTTP and HTTPS support, to initiate requests and receive Web documents

Apart from these requirements, a WIDL processor could be delivered as a java class or a Windows DLL, for integration directly with client applications, or as a standalone server with middleware interfaces, allowing thin-client access to Web automation functionality.

### **Generating Code**

The primary purpose of WIDL is integration with corporate business applications. In much the same way that DCE or CORBA IDL is used to generate code fragments, or "stubs," to be included in development projects, WIDL provides the necessary ingredients for generating Java, JavaScript, C/C++, and even Visual Basic client code.

WebMethods has developed a suite of Web Automation products for the development and management of WIDL files, as well as the generation of client code from WIDL files. Client stubs, which we affectionately call "Weblets," present developers with local function calls, and encapsulate all the methods required to invoke a service that has been defined by a WIDL file.

```

import watt.api.*;

public class TrackPackage extends Object
{
    public String TrackingNum;
    public String disposition;
    public String deliveredOn;

    public String deliveredTo;

    public TrackPackage(String TrackingNum)

throws IOException, WattException, WattServiceException

    {
        String args[][] = {
            {"TrackingNum", TrackingNum},
            {"DestCountry", DestCountry},
            {"ShipDate", ShipDate}
        };

        Context c = new <I>Context</I>();

        c.loadDocument("Shipping.widl");
        Result r = c.invokeService("FedexShipping",
                                   "TrackPackage", args);

        disposition = r.<I>getVariable</I>("disposition");
        deliveredOn = r.<I>getVariable</I>("deliveredOn");
        deliveredTo = r.<I>getVariable</I>("deliveredTo");
    }
}

```

Figure 2-9: Java Stub

Figure 2-9 features a Java class generated from the package tracking WIDL presented earlier in Example 1. This class demonstrates the following methods that

are part of the API that WebMethods has developed for processing WIDL:

- Context
- loadDocument
- invokeService
- getVariable

After declaring the variables that will be used by the PackageTracking class, a handle `c` to a new Context of the WebMethods Web automation runtime is created. All API calls are then made against this handle.

`loadDocument` loads and parses the specified WIDL file, in this case `Shipping.widl`. Loading the WIDL defines the services of the Shipping interface to the runtime. `invokeService` actually submits the input parameters to the `TrackPackage` service, which makes the appropriate HTTP request and returns either a result set which contains the bound output variables or an error message specified by a `<CONDITION/>` statement within the `<SERVICE/>` definition. `getVariable` is then used to extract the values of the output variables and to assign them to class variables.

Within the Java application, the package tracking service looks like a simple instantiation of the `TrackPackage` class:

```
TrackPackage p = new TrackPackage("12345678");
```

In short, an application makes a call to a local function that has been generated by WIDL. The local function encapsulates the API calls to the WIDL processor. The WIDL processor:

- Loads the WIDL file from a local or remote file system
- Passes the function's input parameters as an HTTP request

- Parses the retrieved document to extract target data items
- Executes any conditional logic for error checking or service chaining
- Returns the extracted data into the output parameters of the calling function

Generated Java classes can be incorporated in standalone Java applications, Java Applets, JavaScript routines, or server-side Java "Servlets." Generated C/C++ encapsulating Web services can be deployed as DLLs, shared libraries, or standalone executables. WebMethods implementation, the Web Automation Platform, provides Java classes, a shared library, a Windows DLL and an Active/X control to support Visual Basic modules which can be embedded in spreadsheets and other Microsoft Office applications.

### **2.3.8 Web Services**

Web services, in the general meaning of the term, are services offered via the Web. It is a consequence of a natural evolution. Over time, applications have become loosely coupled and split into multiple components. This has allowed the distribution of an application across many different machines. This way, multiple computers' resources can be used to provide the most resources possible to an application. To better understand why the need of Web services, and eventually where the WIS solution fits which is mentioned in later sections, the evolution of distributed computing and component technology is discussed here.

The architecture that grew out from this need became known as the client-server model. This model involves a central server that contains the database or other central data store that all clients access. The client handles the user-interface screens and some or all of the business logic before sending the data to the server. This frees the server resources to concentrate on data storage and access, while making full use of the resources of the client machines. But there are problems with this approach. The

major obstacle is the maintenance time required when new applications are shipped out to hundreds or thousands of desktops. The requirements associated with these fat-client applications are an obstacle to deploying them successfully. If all the correct DLLs and library files are not installed and registered correctly, the application will not function. Another problem is that it can become extremely costly in terms of resources required to scale up to a large number of users. They consumed database connections and other resources in a way that made it difficult to increase the number of client machines without dramatically increasing server power.

The solution to this architectural problem came with the broad acceptance of the World Wide Web. Web applications could now be built so that the only requirement for the client was an Internet browser. These applications became known as thin clients, because they used far fewer resources on the client machine. Web applications are built using dynamic Web pages accessing a database or middle-tier components. In this model, the client machine is used less, because the required resources for HTML pages are much less than for a standard application that runs locally.

The final result is a model that allows for the distribution of the application at the server level. A Web server is used to process all HTTP requests, while one or more application servers can be used to run and manage all middle-tier components. A database server contains the actual data store for the application. With so many tiers to the application, the total work is spread across many computers.

There are many benefits to this model, as we have seen from the huge growth in Internet companies and applications in the last few years. Clients are no longer forced to install and application locally; applications can be developed for large numbers of people and, when functionality has to be upgraded or replace, the central Web server can be upgraded without ever having to change the client.



While this architecture is often used in building enterprise-level Web applications, it is not tied exclusively to Web applications. This solution, the n-tier model, is designed to be flexible and able to support any type of user-interface from dynamic Web pages to pocket devices.

At the same time that the client-server revolution was taking place, interest in component technology was also growing. In the beginning, developers used simple code reuse. This involved sharing common functions and subroutines. This had many problems, one being that if code was not designed very carefully to be portable to another project, it was not very useful and had to be re-implemented.

When the world moved to the object-oriented programming model, the solution to this problem was going to be the class. Object-oriented programming hides the implementation of a class in private methods, allowing a client to access only the header, or definition file. For enterprise developers this still presented many problems as developers tried to bring classes from one project forward into new ones. They found that differences between compilers, access to the uncompiled source code, and dependency on a specific programming language made this solution very difficult to achieve.

The next solution lay in components. The concept behind this technology was interface-based programming. A component would publish a well-defined interface that could be used to interact with it. This interface was a contract that was guaranteed to remain in place. Other developers could, therefore, develop using these interfaces, confident that future changes to the component would not break their code.

The component interfaces were in a binary standard, giving developers the choice to use different programming languages for the component and the client. For Microsoft, there is COM and DCOM; for Java, there is JavaBeans. Both are very

successful solutions.

Web services are not necessarily going to replace component development; components make a lot of sense in an intranet solution where the environment is controlled, and it does not make sense to expose purely internal objects through a less efficient Web service interface. Web services make interoperating easy and effective, but they are not as fast as a binary proprietary protocol such as DCOM.

The problem with components lies in distributing them across the Internet. Before Web services, if a company created a great COM component for performing some functionality, they could sell and distribute that COM component. Your company might buy this component, install it on every server that needed the functionality, and use the component in your custom solution. With Web services this model changes. Now the third-party vendor exposes a Web service to provide the functionality previously offered by the component. Your company accesses this Web service in your custom application and you no longer need to install any component on your servers. As upgrades are made to the functionality of the Web service, you have access to them immediately, because the Web service is centrally located. Instead of having to redistribute new components to all servers when upgrades are required, nothing has to be done to take advantage of changes in the internal working of the Web service.

This new model allows for “functionality reuse”. This is a fundamentally new concept. It is not interface-based, though it uses many of the concepts related to interfaces. It is not object-oriented, although systems can be built using object-oriented and component-oriented concepts. What functionality-reuse programming allows is the ability to use other systems to perform specific functionality in your application. This allows you to concentrate on the real business

problems, while taking advantage of third party expertise and experience in those areas you choose to access via Web services. Instead of forcing developers to choose between certain technologies when looking for functionality, Web services allow them to choose the correct functionality, not the correct technology. This is because the interface is defined; the application performing the actual functionality can be written in any language. This frees architects and developers by allowing them to choose functionality based solely on the requirements of the system, not the technological constraints.

A Web service is accessible through standard Internet protocols such as HTTP. This means that any client can use the Internet to make RPC-like (Remote Procedure Call) calls to any server on the Internet and receive the response back as XML. The messages sent back and forth between the client and server are encoded in a special XML dialect called Simple Object Access Protocol, or SOAP for short. This protocol defines a standardized way of accessing functionality on remote machines.

The fundamental concept driving Web services is that clients and servers can use any technology, any language, and any device. These things are at the discretion of the developer and the medium of the device. It is only the interface that is explicitly defined for each service. The way they access a Web service is the same: SOAP over HTTP. With Internet access built into everything these days, all we need to access sophisticated server applications is a basic XML text processor to encode and decode the SOAP messages.

There are many distributed component technologies, such as Distributed COM (DCOM), Common Object Request Broker Architecture (CORBA), and Remote Method Invocation (RMI) for Java, work very well in an intranet environment. These technologies allow components to be invoked over network connections, this

facilitating distributed application development. Each of these works well in a pure environment, but none is very successful at interoperating with the other protocols. Java components cannot be called using DCOM, and COM objects cannot be invoked via RMI. Attempting to use these technologies over the Internet presents an even more difficult problem. Firewalls often block access to the required TCP/IP ports, and because they are proprietary formats both the client and server must be running compatible software.

The advantage of SOAP is that it is sent over HTTP. Most firewalls allow HTTP traffic to give end users the ability to browse the Internet. Web Services operate using the same ports in the firewall (port 80 for HTTP and port 443 for HTTPS), allowing server applications to be securely protected while still exposing business functionality via Web services. SOAP is not a proprietary protocol; instead, it is an XML standard that defines the messages set between the client and the Web service. Any Web service can therefore interact with, and be used from, any technology solution. This increases the ability to distribute systems without relying on a single technology like DCOM, CORBA, or RMI.

What makes Web services so revolutionary is the application of Internet standards. All messages are sent via the HTTP protocol. The messages passing between Web services and clients are encoded in XML. How a request for a Web service is encoded is specified in the Simple Object Access Protocol, or SOAP for short. This standard, which was developed by a consortium of companies including Microsoft, IBM, DeveloperMentor, and Userland Software, is now an official W3C standard under review. SOAP messages are specified in a well-defined XML format.

SOAP makes it possible to access any Web service using well-known calls and response. The actual system residing behind the Web service could be any proprietary

system. As long as they send and receive valid SOAP messages, any system can call them, and they can call any Web services on the Internet.

In addition to SOAP there are a handful of standards that are required to make Web services a viable solution:

- XML: provides a standard and unified way to represent data and messages across all Web services.
- WSDL (Web Service Description Language): This specifies the interface of the Web services: what each method is called and what parameters it accepts and returns. From this document, the valid SOAP messages that can be sent to a Web service can be established.

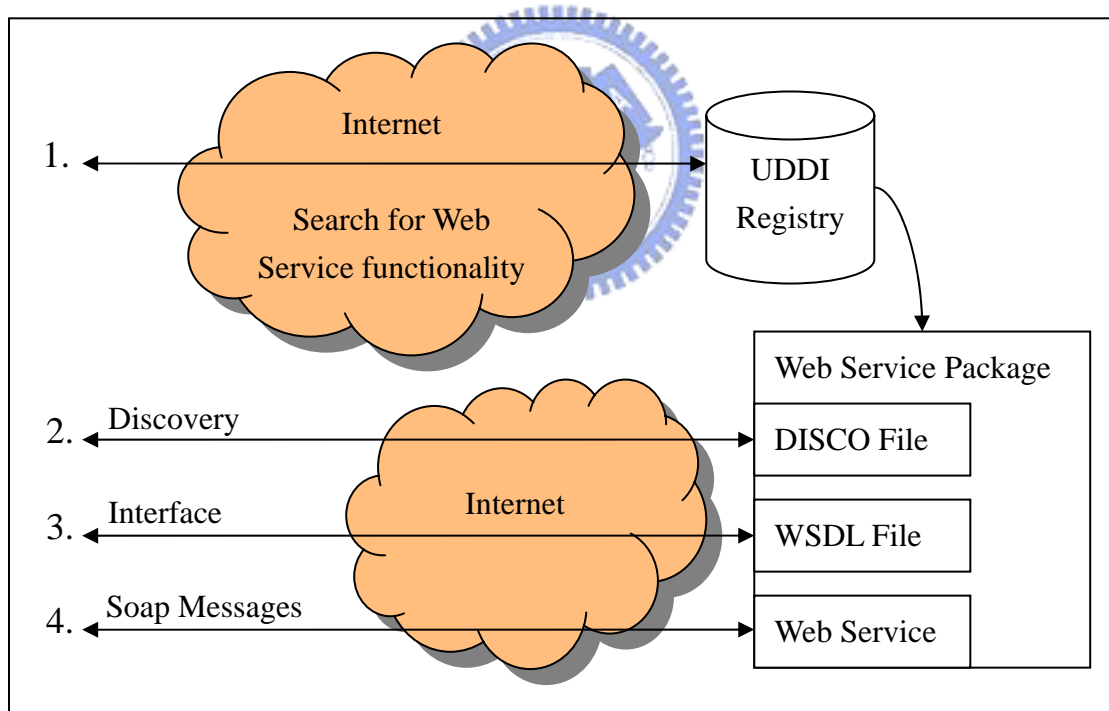


Figure 2-10: The Web Service Solution

- DISCO (Discovery Protocol): This acts as a pointer to all the Web services located on a particular Web site. It enables dynamic discovery of published Web services for an organization.

- UDDI (Universal Description, Discovery and Integration): This acts as a central repository of available Web services. Applications and developers can access the UDDI registry to learn what Web services are available on the Internet.



---

## CHAPTER 3 THE WIS PLATFORM

---

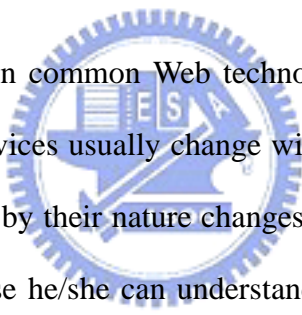
### 3.1 System Overview

The previous chapter presented the many possibilities of Web automation, and there are still much more to be discovered. In this chapter the common properties for Web automation applications are organized and discussed, and an ideal architecture is proposed. An implementation of this ideal architecture, the WIS system, is then documented in details.

The main concept of Web automation is to reuse existing services available in the Internet. It can be seen as the next pace in reusability. Instead of “functionality reuse”, mentioned in the discussion of Web services in section 2.3.8 , what is concerned in Web automation is the capability of “service reuse”, the reuse of entire services. For example, we can put online book store services, library online services and inter library loan services together to create a powerful solution for providing books of any kind; and empowered by Web automation, it will not have only descriptions and links such as traditional portal sites, but also really cooperating together. Continuing this example, which may be called a book agent, the user uses the metasearcher capability of the agent to search every potential provider of the book, no matter if the provider is a library or a bookstore. The agent may create two groups of providers having the wanted book, one of libraries and one of bookstores. If the user selects the group of libraries, the data is passed to an inter-library loan system to acquire the book. If the bookstores group is selected, the agent may propose the best buy by comparing the prices and conveniently send the request for the user. As we can see in the example, entire services are reused, creating an altogether new experience for acquiring a book.

To reuse services, the first issue that needs to be solved is interoperability. Many

protocols have been created for different situations in interoperation. The increase interest in the metasearcher area introduced various protocols, such as Z39.50, OAI and OpenURL. But there are always services that do not support these protocols, since their initial purpose is for interaction with human users, not machines. The result is, the only thing that we may be sure about Web services is that they use well known Web technologies that a competent Web browser will surely do its job without any problem. The most common is the HTTP protocol and the HTML presentation language. Additional mechanisms such as cookies, security and scripts make the picture of the common interoperation even more complicated. But to meet the goal that the ideal Web automation tool needs to be of common purpose, these are the only things that should be relied.



By choosing to only rely on common Web technologies, another problem arises. The user interface of Web services usually change with the time. For example, many sites have advertisements that by their nature changes frequently. For the human user it is not a big problem because he/she can understand their meaning and skip them. But it is difficult for a computer to really “understand” the interface of a Web service. Some solutions were mentioned in previous sections to put away these noises. Intelligent solutions are hard to be widely applicable, in which heuristics are usually suitable for only specific cases. WIDL instead provides a language to describe the position of the required data with more chance to skip unwanted changes. The WIS platform proposed in this thesis uses the WIDL solution, with some modifications.

Parallel processing is a common need of Web automation applications. In a typical metasearcher for example, when the user submits a query, the query is dispatched to various sources at the same time, so that every source can do its job in parallel with others. When a source terminates its job, the results are returned and the agent can do



further processing while there may still have sources working with the request (Figure 3-1).

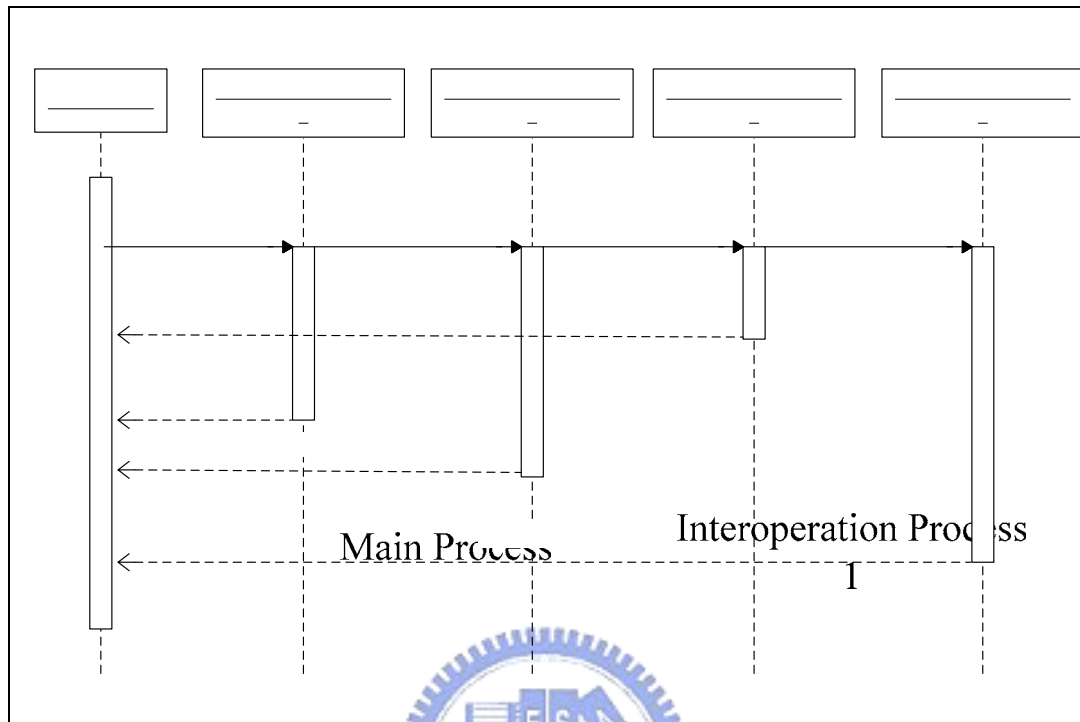


Figure 3-1: Multiple Processes in a Metasearch Dispatch Query

From the metasearcher example, it seems that only the main process need to have the privilege of creating other process. But there may be cases in which a process may need to create sub-processes, and messaging between any one of the processes is needed. The WIS system supports any arbitrary arrangement of processes due to its various scopes. Scopes are to be discussed in next sections.

In a Web automation application, the need to interoperate with various sources at the same time may consume the resources of a server very fast. In a three tier architecture, the business logic tier accomplishes the Web automation tasks (Figure 3-2). For example, in a previous work, the VUCS system, the automation component is implemented as a DCOM object. When a user performs a search, the interface program requests the Web automation DCOM object, which interoperates with the target resources. To scale up to a large number of users, computing power can be

increased by adding more servers in the business logic tier; but the network may eventually become the bottleneck of the system when multiple users are performing Web automation tasks in the server.

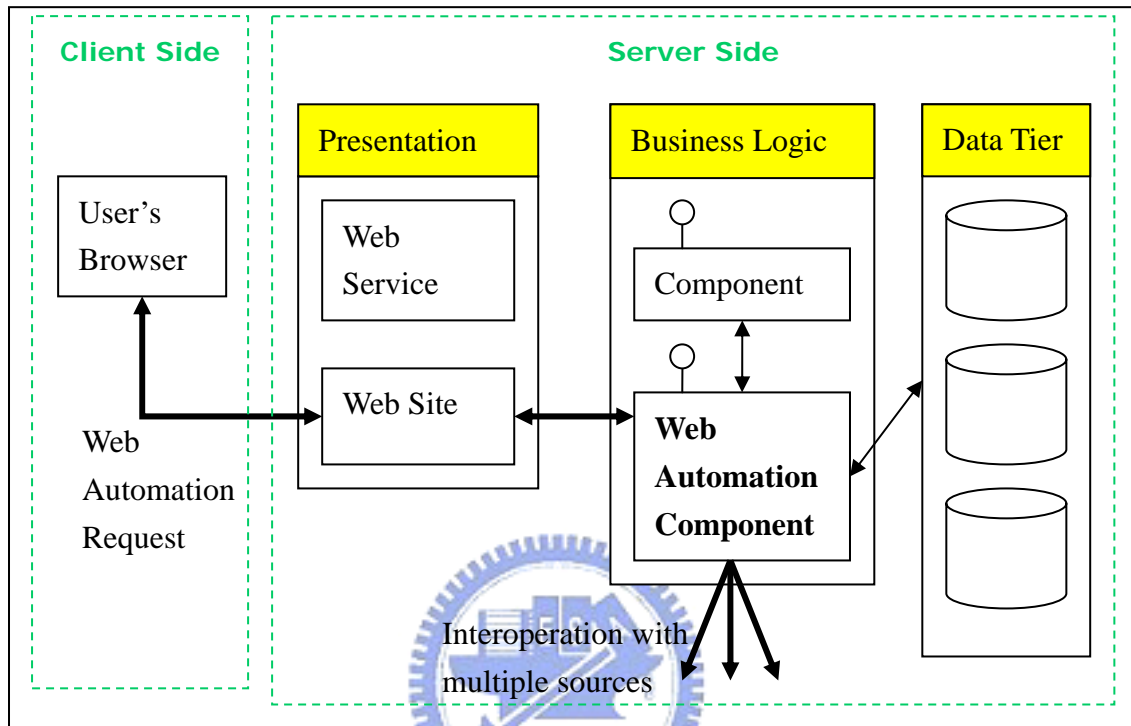


Figure 3-2: Web Automation in a Three Tier Model

The solution provided by WIS for this problem is that it permits Web automation tasks be performed at the client side, which reduces the load in the business logic tier. From Figure 3-3 we can see that WIS replaces the position of the browser. It is especially efficient when users are widely spread in the network because the Web automation task will mainly spend local network bandwidth, saving the bandwidth of the server. But for this architecture to work, some issues need to be solved.

By moving the Web automation task from the business logic tier, a new problem arises. When in the business logic tier, the Web automation component could easily work with other components and interact with the data tier. For example, in our book recommendation system, the Web automation's task involves reading hyperlinks

stored in the database, extracting the metadata from the online bookstores, and writing the new information back to the database. In the discussion of WIDL in section 2.1.2.1, a lot of applications were mentioned, and many of them refer to integration with the enterprise system. WIDL is mainly designed to run at the server, and any protocol could be used since it needs the supplement of a traditional programming language. But with the Web automation task moved to the client side, there need to be a way to keep the interaction with the enterprise system. Web service technology plays this role by exporting functionality from the enterprise system to WIS. Whenever the Web automation needs support from the enterprise system, it acquires interface information by the WSDL document and then with the definition, functions at the server side can be called by using SOAP messages. Thus the Web automation task running in WIS can access whatever function it needs from the server, from anywhere in the network.

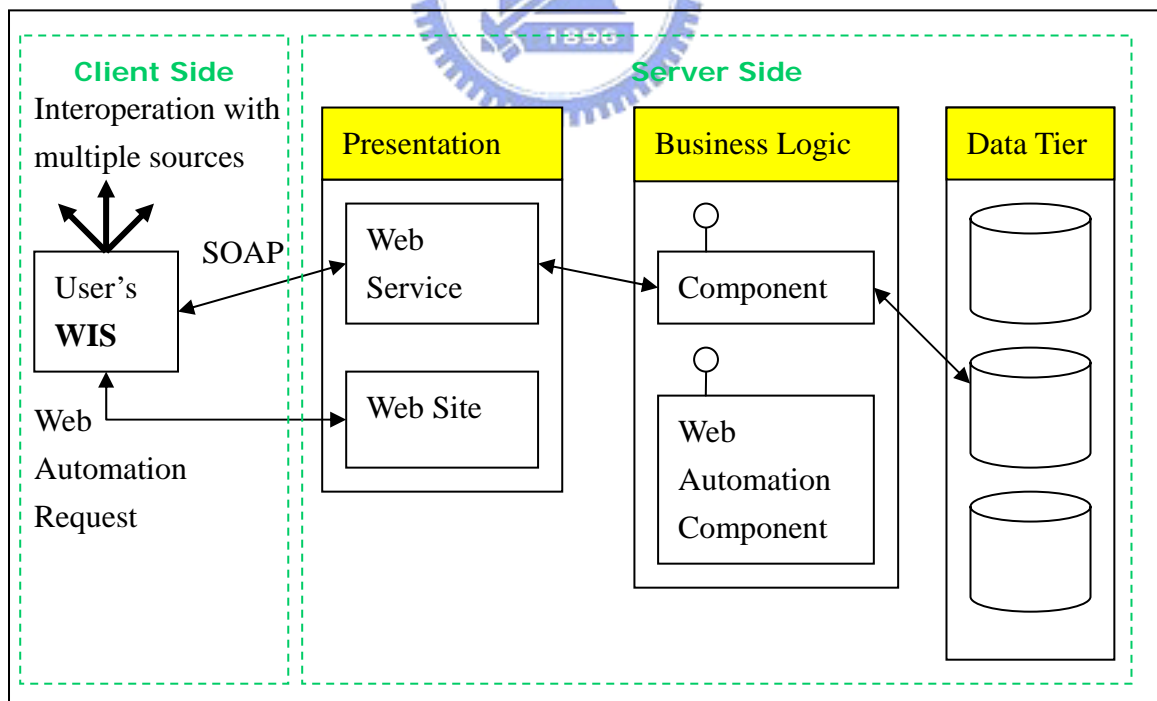


Figure 3-3: Three Tier Model with WIS

The business logic issue in the WIS architecture is solved, but there is still problem

with the interface. In the traditional three tier model, modifications to interfaces can be easily done at the presentation tier. When the user requests a Web automation task from the server, an entire new page is passed back to the browser whenever the server finishes it. But when entire Web automation tasks are running at the client side, how can WIS create whole new pages to show the progress or changes, which varies a lot from application to application? A solution would be to go back to the two-tier model, where clients are designed to specific applications, making the business logic work tightly with the presentation in the desktop. With this approach, WIS will come with different tailored interfaces for different applications, and the maintenance nightmare of updating hundreds or thousands of desktops comes back. Another solution is to use HTML interfaces instead of hard-coded interfaces, which are downloaded from the server. To refresh the interface, the Web automation process can request the server to construct a new page for it according to the parameters given and send it back. By this way WIS can keep its generality and compliance with the three-tier model. Better performance can be achieved by using DHTML, preventing the need of the round-trip every time the interface needs a refresh. The Web automation process can notify the user about any change without having to reload entire pages from the server. Any kind of report can be created by this way.

Many Web automation tasks are repetitive and need to be executed periodically. In the Website checking application, the check task is performed by intervals determined by the site manager. The solution provided by WIS comes from DHTML, which provides timed function call.

Data returned from sources need further processing before presenting them to the user. Users' requests to the Web automation application need processing before sending to sources. In a metasearcher for example, the query given by the user is

translated to a format that the target resource accepts, which may be different for every resource. Results from every resource may have many differences, such as different date formats and lack or availability of some fields. Reranking will need even further computing. The WIDL, which plays only the role of an interface definition language, let this work to the complementary programming language. Because it is difficult to have a single tool that can process so many variations in data processing (if any exists), WIS adopts to use a common programming language. The developer can use JavaScript or use Java applets to perform the transformation task.

With this architecture, WIS moves the Web automation task from the server to the client, distributing even more the work load without affecting the convenience of a counterpart browser. Although WIS substitutes the browser, it still keeps the advantage of being a common client for the various applications. There is no need to have a special version of WIS for every application since it is designed with the concern of supporting any application, keeping the original idea of a common client that simplifies the deployment of new applications which usually happens in the maintenance phase.

## **3.2 Programming Basics**

WIS provides all basic needs to create a Web automation application. So there are many things to understand before being able to create applications with WIS.

A WIS application is composed of several pieces working together to accomplish the Web automation needs, which are called WIS components. WIS components are located at the server and downloaded to the WIS client whenever a user starts a Web automation application. Every WIS component is acquired by a URL, so WIS components does not necessarily need to be in a single server; it can be located in different locations, adding more management possibilities.

WIS components can be divided in two distinctly different categories: WIS pages and WIS profiles.

WIS pages are similar to Web pages, with the exception that it contains code proprietary to WIS and cannot be displayed in ordinary Web browsers. Its main function is to provide interface to the user. The first WIS component that a WIS client gets from the server is the WIS application main page. It provides a starting point for the Web automation application, which can do initializations such as downloading WIS profiles. WIS pages can contain HTML, DHTML, JavaScript or Java applets.

WIS profiles are documents encoded with XML and can contain WIS proprietary profiles or profiles specifically designed for the Web automation application. Profiles specified by the developer may contain any information such as setting parameters for the automation application. WIS proprietary profiles instead are for WIS and must follow its definition and may have several different types. At the moment there are only two types: WIS surfing processes and WIS extraction definitions. The WIS surfing process defines the process of interoperation with a resource. The WIS extraction definition tells WIS how to get desired data from pages returned by sources.

In a typical scenario, a Web automation application works as follows. The user first selects a Web automation application by giving the URL of the WIS application main page. Once the application is downloaded, the main page performs initializations, usually downloading the profiles needed. After initialization is finished, the user can interact with the Web automation application interface. The Web automation application will then create WIS sessions as many as needed to accomplish the required task. Every WIS session performs interoperation with a single resource according to the description in the WIS surfing process. A WIS session can also be

used to load other WIS pages from the Web automation application server and display to the user.

### 3.2.1 Programming Contexts

The programming language used by WIS is JavaScript. But the stateless nature of Web applications and the various WIS components creates different regions of code that have different concerns, which are called programming contexts by this thesis. The Web automation application developer should be clear about which context he/she is using. There are two programming contexts that originate from the type of the WIS component: WIS pages context and WIS surfing process context. Like JavaScript in Web pages, the developer can expect to have anything that an ordinary browser would provide for scripting, and the lifetime of entities such as functions, variables and objects declared here are only within the page. These two contexts are essentially stateless, since every time the page is reloaded or substituted, user defined entities disappear. Then what can be done to make these entities more persistent when a longer lifetime is needed? WIS provides two more contexts to solve this need. The WIS session context has the lifetime of a WIS session, which supplements the WIS surfing process. When an entity should persist between pages of a surfing process, it can be placed at the WIS session context. Code in the WIS surfing process context can access entities defined in the WIS session context. Another context available is the WIS global context, which exists until the Web automation application terminates. The WIS global context is accessible by all other contexts.

<b>Context type</b>	Page	Surfing process	Session	Global
<b>Purpose</b>	Automation application	Resource interaction	Session persistence of entities	Global persistence of entities
<b>Lifetime of</b>	Within page	Within page	Within session	Within

entities				application
----------	--	--	--	-------------

Table 3-1: Summary of programming contexts

### 3.2.2 The WIS Surfing Process

Resources have different interfaces, so when the user uses the browser he/she must “understand” the interface and make the correct reactions to proceed. The WIS surfing process tells WIS how to manipulate resource page interfaces.

```

<AutoProcesses StartURL=“starting url”>

  <ScriptCode>Session context code</ScriptCode>

  <ScriptCodeGlobal>Global context code</ScriptCodeGlobal>

  <AP>Page context code
    <SP>Frame context code</SP>
    <SP>Frame context code</SP>
    ...
  </AP>

  <AP>Page scope code</AP>
  ...
</AutoProcesses>

```

Figure 3-4: Structure of a WIS Surfing Process

The <AutoProcesses> is the parent element for the definition of a WIS surfing process. The process consists of a series of by page operations. The following are the attributes for the <AutoProcesses> element:

- StartURL: The location of the first page for the entire process.
- Name: Identifier for the WIS surfing process.
- ProxyType: The type of the proxy that will permit connection access to the target resource. The value can be: Default, Direct and Auto.



- ProxyURL: Address of the proxy if type of the proxy is Auto.

The <ScriptCode> element defines code to be executed in the session context. It can be used to initialize variables persistent to the session context, or declare functions.

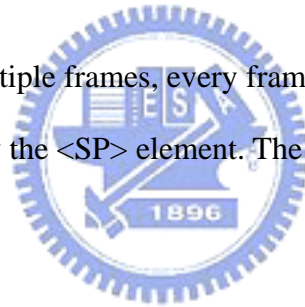
The <ScriptCodeGlobal> element defines code to be executed in the global context. Variables and functions persistent to the entire Web application can be declared here.

The <AP> element defines code to be executed for a page of the resource during the surfing process, i.e. the code interoperates with the page. The attributes are:

- Name: Identifier for the step in the surfing process.
- NumberOfFrames: Declares the number of frames if the interface uses frames.

If the page consists of multiple frames, every frame can have its own manipulation code, which can be defined by the <SP> element. The attributes are:

- Name: Identifier.
- FrameIndex: Name of the frame that the code is for. WIS uses it to know which frame will use the given code.



### 3.2.3 Profile Management

WIS profiles are documents encoded with XML. A single WIS application may consist of many WIS profiles. For convenience, these WIS profiles can be divided in small pieces for development and management convenience. WIS profiles can be placed anywhere. Putting them in the client-side can save download time, especially if there are hundreds of profiles. But frequent updates may be difficult if there are many clients. It is more convenient to have it stored at the server-side and be accessed every time a Web automation application needs.

Multiple profiles can be unified to a single document for easy access from the Web automation application. The UniDoc object does this job. It identifies the <UniDocInclude> element which is substituted by the demanded profile defined in the Src attribute. By this way, profiles are embedded into parent profiles, forming a single profile for the Web automation application. Parts of the profile then can be obtained by using DOM or XPath.

### 3.2.4 The WIS API

WIS exposes various functionalities through an API that is called by JavaScript. All extensions provided by WIS are obtained through an object named external. The external object can be obtained from the window object with the following JavaScript code:

```
var ext = window.external;
```



What can be accessed directly by the external object is the WIS session context. For example, if we have a variable called “CommandLine” in the WIS session context, we can obtain its value with the following code:

```
var val = window.external.CommandLine;
```

For a function in the WIS session context, the call procedure is similar:

```
var result = window.external.MyFunction(x);
```

The WIS global context can be accessed by any other context. We obtain access to it by the GlobalCO object, and its usage is similar to the WIS session context:

```
var val = window.external.GlobalCO.MyGlobalVariable;
```

WIS gives access to the rest of the API through the GlobalCO object. Table 3-2 gives a complete list of provided objects at the moment.

Property	Description
GlobalCO	Global context
UniDoc	Provide access to WIS profiles
WMultiWB	Provide management facilities for a collection of WIS sessions
WAutoWB	WIS session
WMessage	Log messages from the Web automation application
WBExt	Miscellaneous tools are provided by this object
Soap	Support for Web services

Table 3-2: Objects in WIS

UniDoc provides a centralized manner of using WIS profiles. We prepare it by using the Load method, giving it the location of the main profile, which may include other profiles:

```
window.external.GlobalCO.UniDoc.Load(MainProfileURL);
```

WMultiWB is used to manage WIS sessions. To create a new instance of a WIS session, we use the AddSession method:

```
var oWAutoWB = oWMultiWB.AddSession(SessionName);
```

The parameter SessionName is a string identifier for the session. AddSession returns an WAutoWB object, which is a WIS session instance. Other methods provided by WMultiWB are:

- Reset(): Method. Destroy all existing WIS sessions, except the main session.
- Scr: Property. This is another way to declare entities in the WIS global context.

The WAutoWB gives us control to a single WIS session. To start a WIS surfing process we first load it into WAutoWB:

```
oWAutoWB.Load(SurfingProfile);
```

The parameter SurfingProfile is the XML node of the element <AutoProcesses>. Then we can execute the given surfing process:

oWAutoWB.Run0();

Other methods and properties provided by WAutoWB are:

- ID: Property. Used to obtain the identifier of the WIS session.
- Navigate(TargetURL): Method. Use it to navigate to the given TargetURL instead of running a WIS surfing process.
- SOnAllDocsCompleted: Event. Write an SOnAllDocsCompleted event handler to be notified whenever a step is accomplished in the WIS surfing process.
- InsertedProcessName: Property. Gives the name of the WIS surfing process used in the WIS session.

WExt provides miscellaneous facilities:

wHTMLGetElementsC ( CollectionOfElements, AttributeConditions): Method. Search for an element in CollectionOfElements which matches the given AttributeConditions. Find elements in a page returned by a resource.

NewSoap(): Method. Creates a new Soap object.

### 3.3 Data Elements Extraction

WIS uses a language similar to WIDL for data elements extraction from HTML pages, but with some enhancements.

The <REGION> element has a new attribute, the SINGLE attribute, with the value being an object reference. It defines a reference point, i.e. a reference element for locating other elements instead of a targeted subregion of a document. When a user wants to find something in a HTML page that has changed, he/she first identifies unchanged parts, such as titles, which were associated with the wanted data in past versions. Whenever the wanted title is found, it is very likely that the desired part of

the page is near. The <REGION> element with the SINGLE attribute tells WIS what probably will not change in the HTML page, and thus can serve as a reference point.

Object references in WIDL uses an object model to provide access to elements and properties of HTML. To access a child property or element of the parent, the dot operator is used. WIS introduces another operator, the parent operator (^), which returns the parent of the element. It is useful with the reference element. For example, in a online bookstore we may find that the element with the text “ISBN” is the most likely to not change, and the rest of demanded data surrounds it. To define it as a reference element:

```
<REGION NAME="RefEle" SINGLE="li['ISBN']" />
```

With the traditional dot operator, we can get the ISBN number:

```
<VARIABLE NAME="ISBN" REFERENCE="RefEle.text"/>
```

The parent operator returns the element <UL> containing the <LI> element with the ISBN, which is also the parent of other variables. To get other variables such as title and author, we can use the parent operator with the reference element:

```
<VARIABLE NAME="Title" REFERENCE="RefEle^li ['Title'].text" />
```

```
<VARIABLE NAME="Author" REFERENCE="RefEle^li ['Author'].text" />
```

The parent operator can be used consecutively, since every element have only one parent. For example, if the title is in a higher level of the document object hierarchy, the variable definition can be:

```
<VARIABLE NAME="Title" REFERENCE="RefEle^^text" />
```

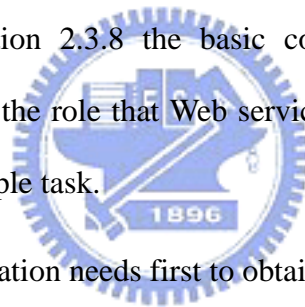
It tells WIS to extract the text from the element which is three elements above the reference element.

WIDL provides several types of indexing, which were discussed in section 2.3.7. But only one type can be used at the same time. There are occasions that a single indexing method is not sufficient to match wanted elements and multiple conditions must be given to filter out undesired elements. WIS solves this by providing multiple indexing, with every condition separated with a comma. For example, to define a variable that has the class attribute with value “small” and contains the text “Price”:

```
<VARIABLE NAME="Price" REFERENCE="td[class='small','Price'].text" />
```

### 3.4 Communicating With the Server

WIS communicates with a server on demand of the Web automation application by using Web services. In section 2.3.8 the basic concepts of Web services were explained. Section 3.1 shows the role that Web services play in WIS. To use WIS to connect Web services is a simple task.



The Web automation application needs first to obtain a new SOAP client object:

```
var oSoap = window.external.GlobalCO.WBExt.NewSoap();
```

To initialize it:

```
oSoap.mssoapinit(WSDL_URL, ServiceName, Port, WSML_Language);
```

The meaning of the parameters are:

- **WSDL\_URL:** The URL of the WSDL file that describes the services offered by the server or a string containing the WSDL document. In the value specified for this parameter, if the first character is '<', the value is assumed to be a WSDL string. Otherwise, the value is assumed to be a WSDL file name.
- **ServiceName:** Optional. The service in the WSDL file that contains the operation

specified in the Simple Object Access Protocol (SOAP) request message.

- Port: Optional. The name of the port in the WSDL file that contains the operation specified in the SOAP request message.
- WSML\_Language: Optional. The URL of the Web Services Meta Language (WSML) file. This is a required parameter only when using custom type mappers.

After the initialization, the SOAP client object has acquired the WSDL file, which is used to construct a COM dispatch interface. With the dispatch interface, calls to the Web service can be made:

```
oSoap.DoSomething();
```



---

## CHAPTER 4 WIS EXAMPLE APPLICATIONS

---

To show the feasibility of WIS in creating Web automation systems, this Chapter describes three applications developed by WIS. The first one is Unisearch 2, successor of Unisearch mentioned in Section 2.2.1 which is an improved metasearcher to online databases in CONCERT. The second is a book recommendation system for libraries which automates the process of book recommendation by readers. The third is a Web site checking system aimed to periodically check critical Web sites, ensuring uninterrupted service.

### 4.1 Unisearch 2

#### 4.1.1 Introduction

The Unisearch metasearcher system created in our previous work (See Section 2.2.1 ) performs the translation of queries and dispatches queries to various sources, but does not combine the returned results. Exploiting WIS's data extraction capability, Unisearch 2 improves Unisearch by organizing the results and then returning the organized results to the user.

Using WIS to create Unisearch 2 still accomplishes the requirements of Unisearch plus some improvements. In WIS, the unisearch application makes connections from the client side, thus respecting the access policies and statistic mechanisms of sources. Depending only on HTTP, its creation does not need the cooperation of source providers.

#### 4.1.2 Application Overview

When the user connects to Unisearch 2 using WIS, he/she gets the search interface of the metasearcher and inputs the query (Figure 4-1). The main program is written in Java, which is more suitable than JavaScript when the Web automation application is



complex and has a lot of code, because the byte code format of Java is smaller and faster to execute than scripts. More databases are supported by Unisearch 2 than Unisearch because WIS can surf through several pages to get the results; on the other hand Unisearch is limited to a single step, making some databases impossible to be searched.

After submitting the query, the results are returned (Figure 4-2). The original pages of the sources are visible to the user, giving a chance to take a look and even continue the interaction with the original result pages returned from the target resources.

Results are collect and displayed in the main page (Figure 4-3). The display process uses DHTML, so there is no round-trip with the server. Using the XML capability in WIS, further processing of results are possible, such as ranking or clustering.

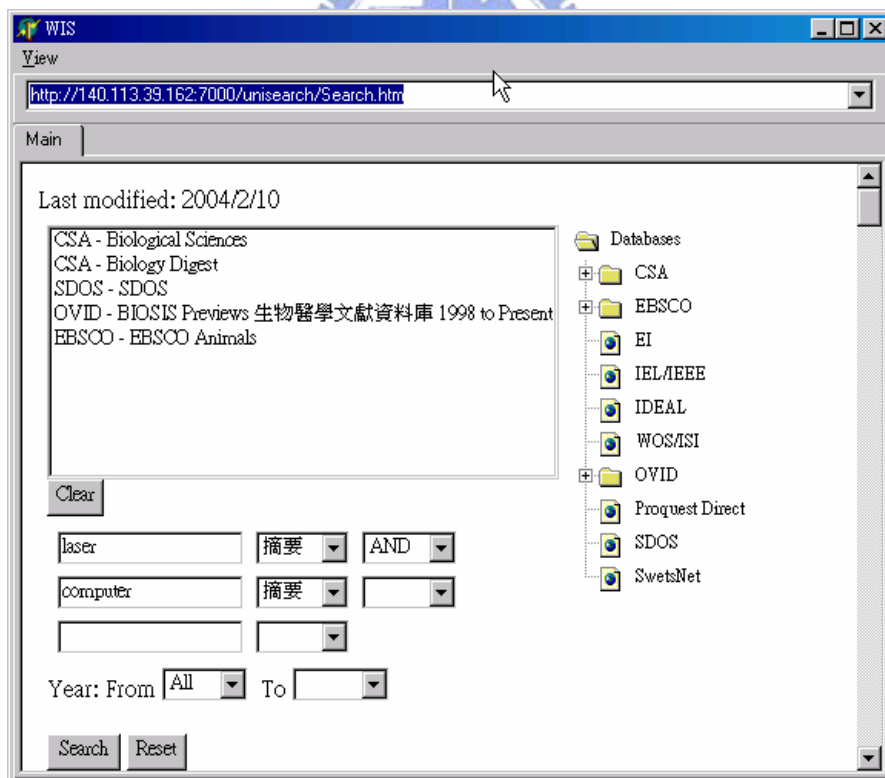


Figure 4-1: Unisearch 2 search interface

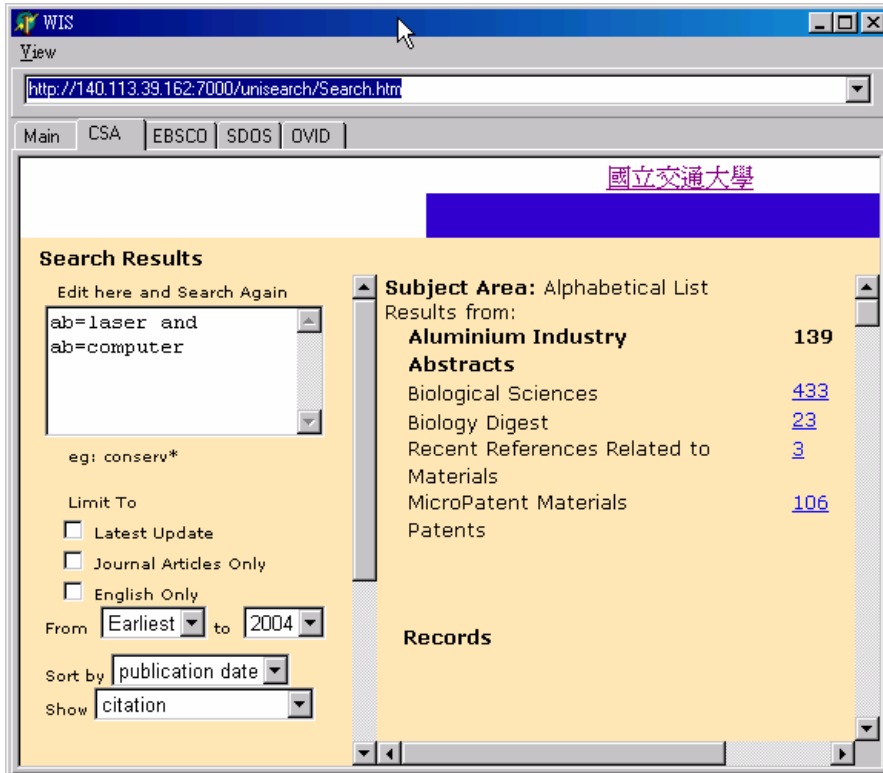


Figure 4-2: Results returned by Unisearch 2



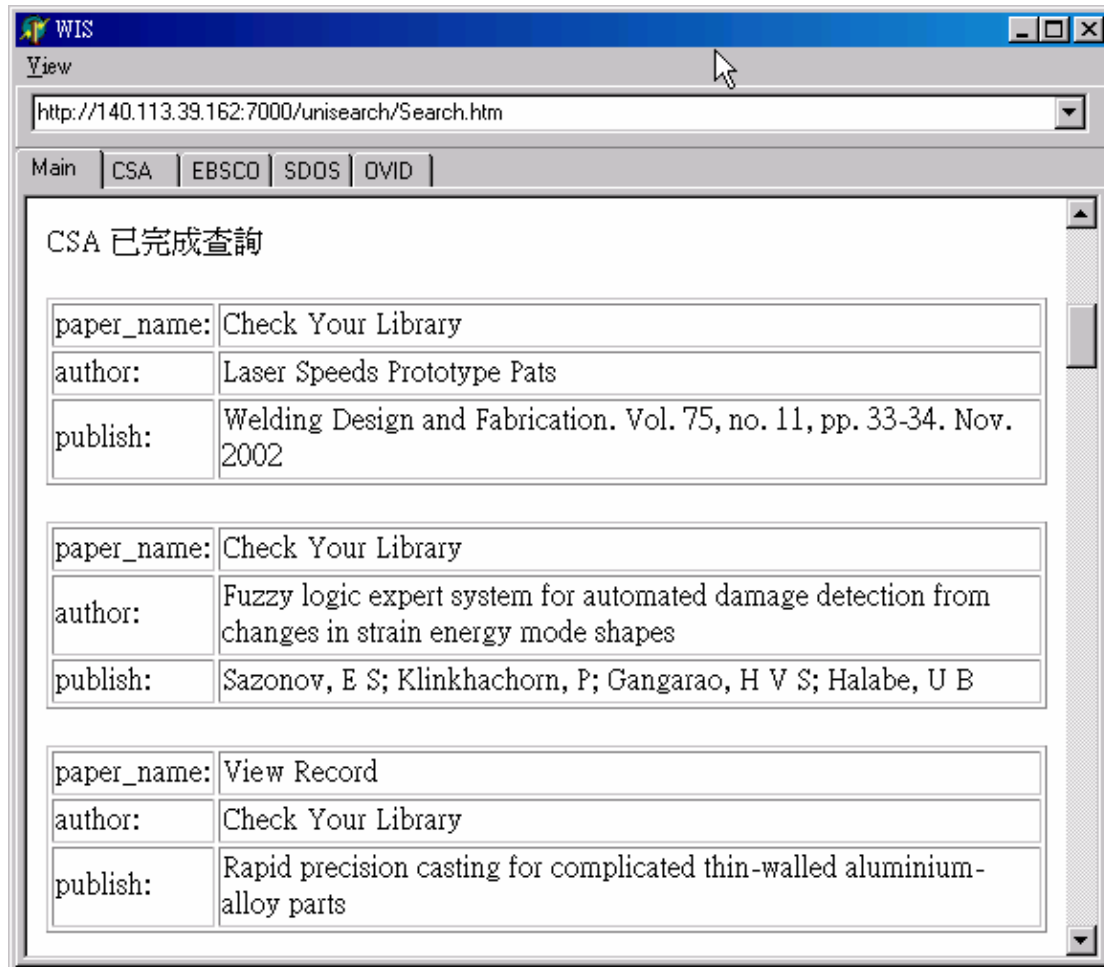


Figure 4-3: Collected results using Unisearch 2

## 4.2 Book Recommendation System for Library

### 4.2.1 Introduction

A library has to know which books are really needed by readers before acquiring them into the library's holdings. Traditionally, a reader obtains metadata about a desired book from a source, such as online bookstores. To recommend the book to the library, the reader may provide its metadata through many different ways and formats, which may be sending emails, passing slip of papers or filling forms. No matter which way, errors and losses in the metadata provided seems to be inevitable, forcing a librarian to check the data. After the check, the librarian needs to type the metadata into the library automation system (Figure 4-4).

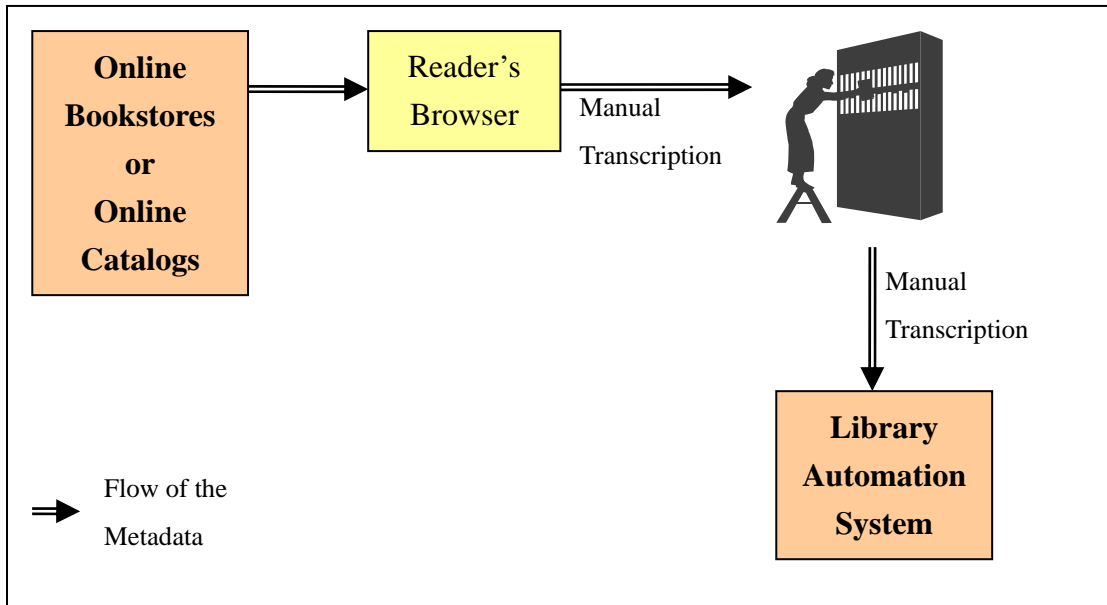


Figure 4-4: Conventional book recommendation process

With the recommendation system described in this Section, the metadata is passed directly from machine to machine, increasing efficiency and convenience (Figure 4-5). No more manual transcriptions are needed.

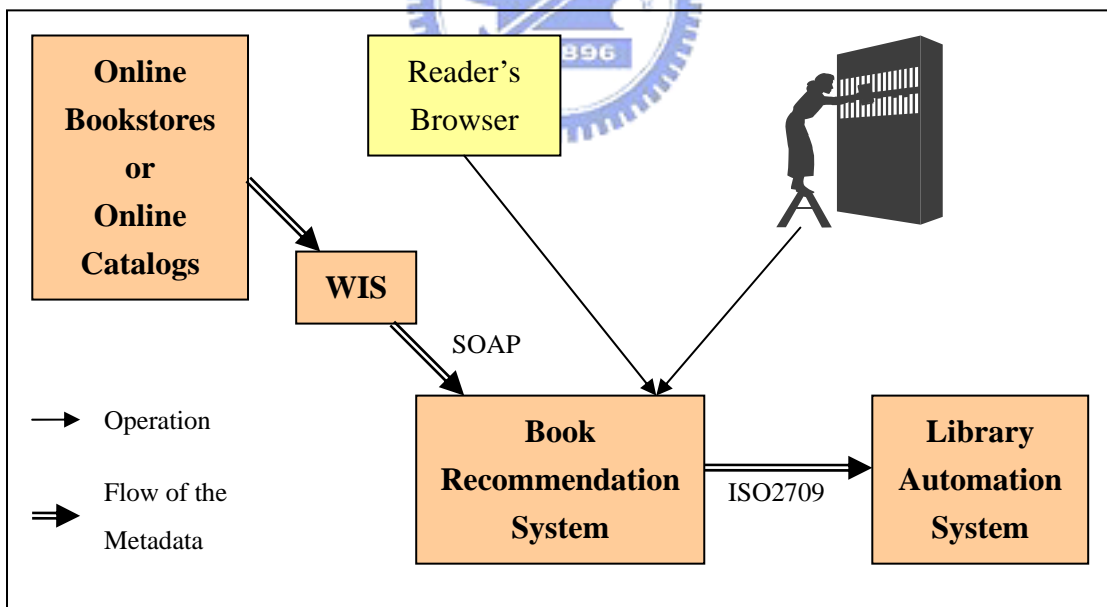


Figure 4-5: Automated book recommendation process

## 4.2.2 Application Overview

Consider the follow scenario. A reader finds out a good book in the Amazon bookstore. To apply a recommendation, the reader should login the system first. Then

he/she submits the URL of the book, as shown in the Figure 4-6.

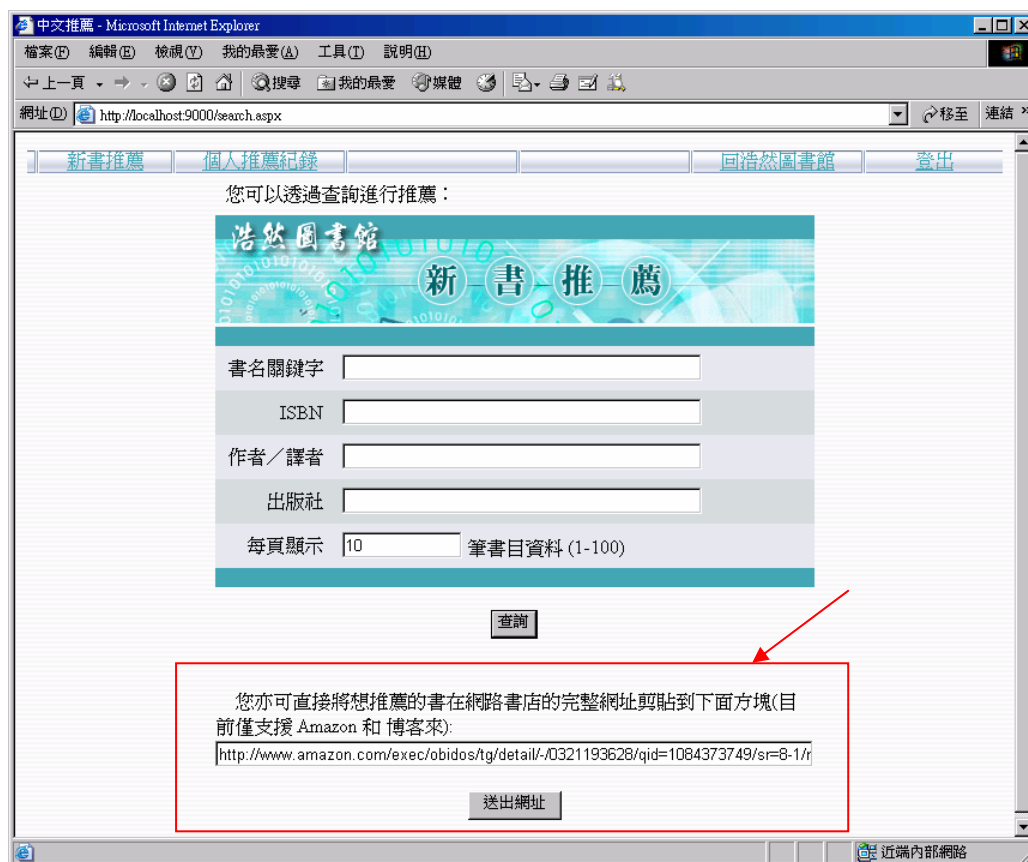


Figure 4-6: Submitting the book URL

The librarian can use WIS with the complement Web automation application for the recommendation system to transform the URLs submitted by readers into metadata (Figure 4-7). It is worth noting that this task could be given to the reader. For example, if the reader has WIS in his/her computer, he/she could enter the recommendation system, and then open another session to an online bookstore, search for the book and submit the metadata displayed in the WIS window directly to the recommendation system. But at the moment, this task is for the librarian. It is still convenient because the conversion is made in batch mode.

If the reader takes a look to his/her recommendations after the librarian performed the metadata extraction procedure, he/she can see the metadata of the book, without having to type anything (Figure 4-8).

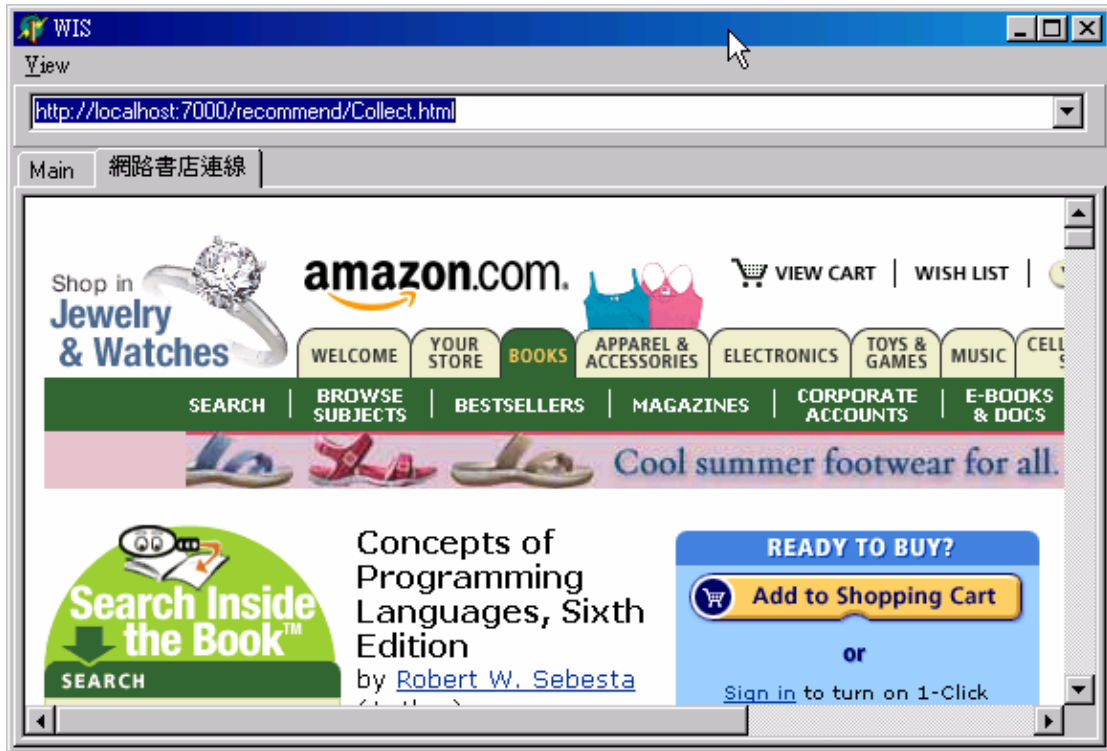


Figure 4-7: Extract metadata from the recommended URLs



Figure 4-8: Reader's recommendation list

## 4.3 Web Site Checking System

### 4.3.1 Introduction

Many Web sites need to keep online 24 hours, without any interruption; otherwise business opportunities may be lost and users unsatisfied with the poor service quality. For simple sites, a connectivity check to the first page may be sufficient. But nowadays many sites have dynamic pages with complex code behind and a plenty of functions. A check to the first page is then no more sufficient; the manager of the site may have to go through many steps until he/she can be sure that everything operates normally. Manipulations such as login test and search test may be required.

This example check sites system uses WIS to make complete tests.

### 4.3.2 Application Overview

The Web site checking example checks two sites at the moment. One is the Taiwan mirror site of IDS, which is a mirror site of multiple online abstract databases produced by CSA and serves Taiwan users. Another is MyLibrary@NCTU, which provides information about various resources in National Chiao Tung University plus personal services. The manager first accesses the Web automation application using WIS, which is shown in Figure 4-9. The manager can define the interval between the tests. The start button begins the periodical test and the stop button pauses it. The result after checking the sites is shown in Figure 4-10.

Whenever a step fails during the process, a Web service is invoked to report the error to a server that will save the message to the database and send an E-mail (Figure 4-11) to the responsible system manager. A centralized log server is beneficial, especially if we want to make the test from different subnets to ensure that there is no problem when difference in location could affect functions of the site such as user access control. The server then can provide a report about previous tests (Figure

4-12).

The test for the mirror site of IDS is a search procedure. The test for MyLibrary is database browsing verification and a login procedure.

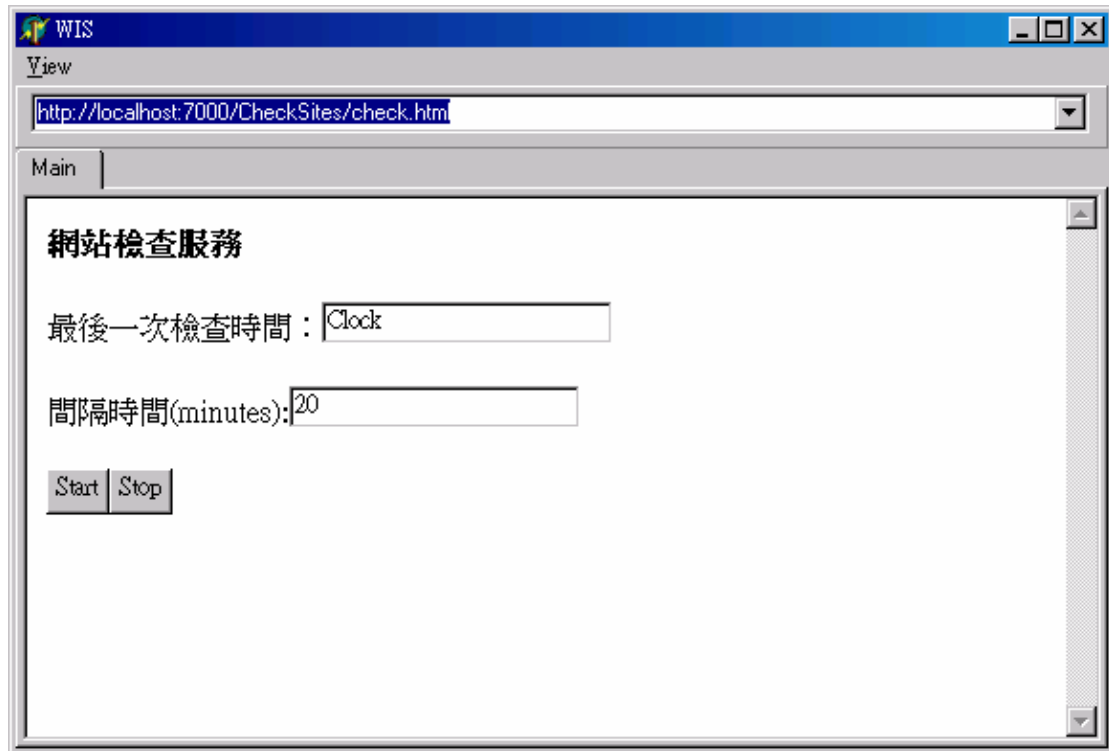


Figure 4-9: The site checking system main page

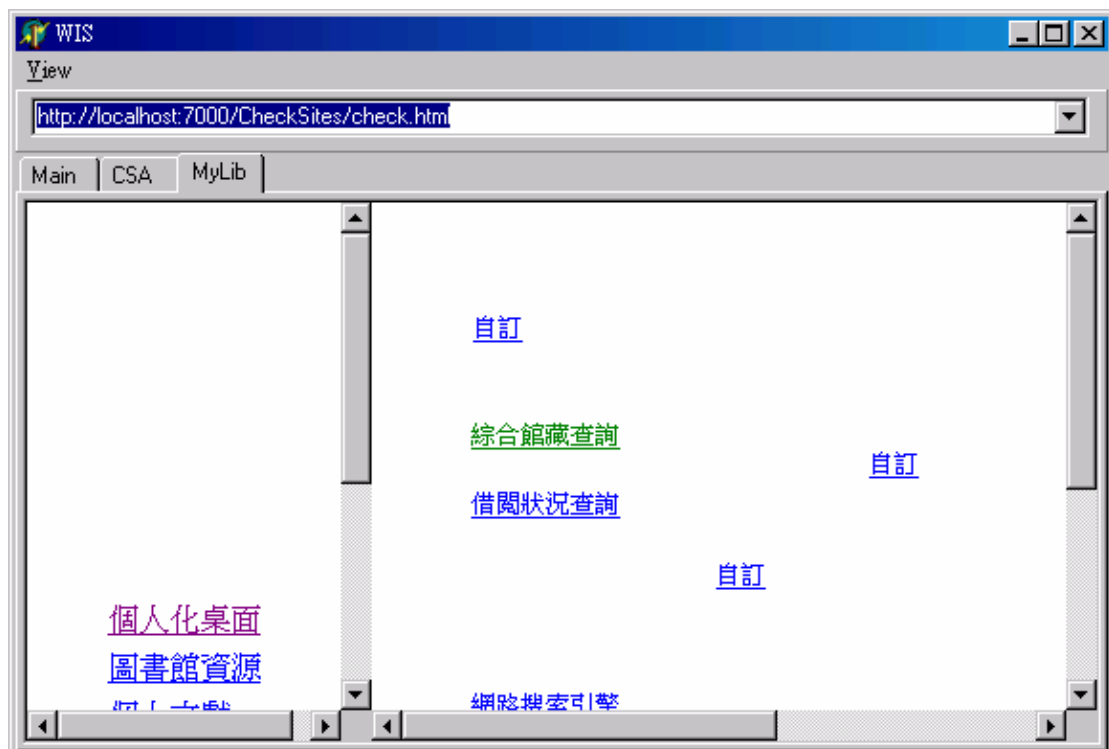




Figure 4-10: After checking the sites



Figure 4-11: An error notification

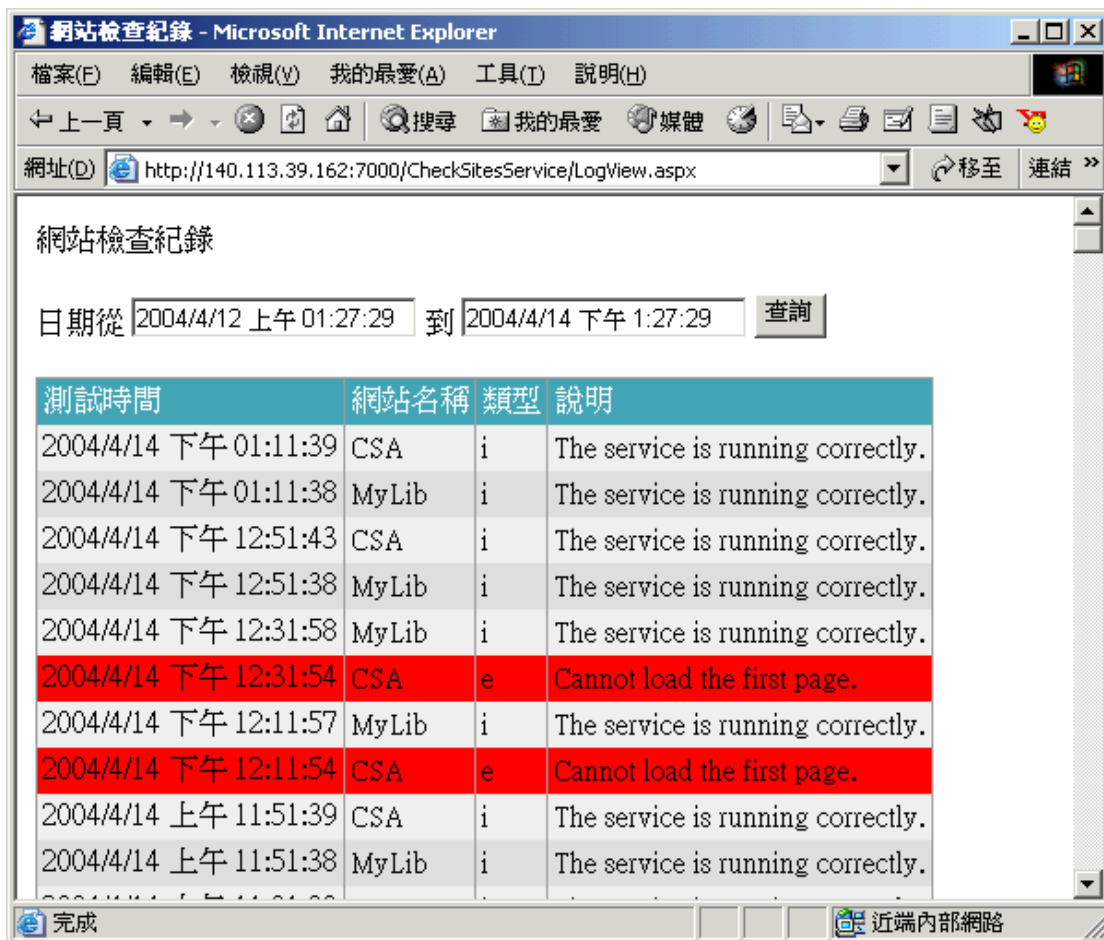


Figure 4-12: Report of test results

---

## CHAPTER 5 CONCLUSION and FUTURE WORK

---

### 5.1 Conclusion

This thesis discusses Web automation applications and the common needs of a Web automation creation solution. On the basis of the common needs, this thesis proposes a general platform to create Web automation applications, without being limited to any specific application.

A general Web automation creation solution should consider the follow issues:

- Interoperability: the core concept of Web automation applications is to reuse existing Web resources. From the interoperation aspect, Web automation applications can adopt standard interoperation protocols (such as OAI, Z39.50), or rely only on the common HTTP protocol. Standard interoperation protocols are not supported by many Web resources; on the other hand, using the common HTTP protocol gives access to all Web resources, but problems such as complexity of Web sites and volatility of the interfaces need to be solved.
- Parallel processing: Web automation applications usually need to interact with several Web resources at the same time.
- Server-side or client-side Web automation execution: execution in server side has the advantage of easier deployment, maintenance and management, but puts limit to performance scalability. Execution in client side gives the contrary.
- Flexible presentation: a flexible interface is necessary so that the tool can fulfill different design needs.
- Integration with corporate systems: Web automation applications may need to work with enterprise servers.

- Scheduling: Web automation tasks are usually repetitive and thus may need scheduled execution.
- Integrated development environment: modern application creation tools provide integrated debugging, wizards and others.
- Intelligent tools: Web automation applications are to replace human labor, so intelligence is a desirable characteristic of an agent.

Feature	Support by WIS
Interoperability	Needs HTML and HTTP only; data extraction facility
Parallel processing	Multiple session, multiple context
Server-side or client-side	Mix the advantages of both server and client approaches
Flexible presentation	HTML interfaces; round-trip free by using DHTML
Integration with corporate systems	Uses Web Services
Scheduling	Enable
Integrated development environment	None
Intelligent tools	None

Table 5-1: Design considerations and WIS support

## 5.2 Future Work

Web automation applications are useful both for common users and enterprises. For common users, Web automation applications can save time and make the use of the Internet more efficient and pleasant. For the enterprise, Web automation can reduce administration costs and human mistakes. The variety of Web automation applications seems to be unlimited, and the applications that can be created by WIS are only limited by the creativity of developers.

Most Web automation applications are built from scratch, which is the main

obstacle to its popularity. WIS aims to turn the creation of Web automation applications easier, but there is still much more to do.

An integrated development environment is a good target for the next step, which gives the possibility of massive creation of Web automation applications. The software industry has benefited much from IDEs, and so can be the field of Web automation applications, pushing us to the era of “service reuse”. There are several issues that can be considered in an IDE for Web automation application creation:

- Integrated debugger for easier troubleshoot of applications.
- Authoring tool: editors and GUI tools with drag-and-drop capability.
- Wizards: there are some related works mentioned in Chapter 2 that emphasizes the creation of Web automation tasks by learning the surfing steps from the user’s interaction with the source. But it is doubtful that the computer can realize everything that the user has done because the complexities of Web applications, which may contain scripts with dynamic content that are not so easy to be detected. So implementing this facility as a wizard is a viable solution, which serves as a starting point for the creation phase. The developer can then make changes to the generated code to fix the incorrect parts.

Web automation applications are related with agents to work in place of human interaction, which in many cases need some sort of intelligence. For example, intelligent metasearchers should analyze returned results and organize them by reranking and summarizing. These intelligent features are usually designed for specific situations, a characteristic that keeps them out of the API set of WIS. With application specific intelligence, WIS will be no more a common client; having different flavors of WIS is unfavorable for wide deployment, so it is better implement

them using component technologies such as Java which can be downloaded by the application when needed. Intelligent tools of common purpose for Web automation applications are a matter of future research.



## BIBLIOGRAPHY

- [1] V. Anupam, F. Juliana, K. Bharat, L. Daniel, "Automating Web Navigation with the WebVCR", *Computer Networks*, Volume: 33, Issue: 1-6, pp. 503-517, June 2000.
- [2] A. Banerjee, A. Corera, Z. Greenvoss, A. Krowczyk, C. Nagel, C. Peiris, T. Thangarathinam, B. Maiani, *C# Web Services: Building Web Services with .NET Remoting and ASP.NET*, Wrox, 2001.
- [3] D. Box, *Essential COM*, Addison Wesley Longman, 1998.
- [4] J.E.F. Friedl, *Mastering Regular Expressions*, O'Reilly, 2002
- [5] E. Harmon, *Delphi COM Programming*, Macmillan Technical Publishing, 2000.
- [6] E. Selberg, O. Etzioni, "The Metacrawler Architecture for Resource Aggregation on the Web", *IEEE Expert*, pp.11-14, Jan.-Feb. 1997.
- [7] M.W. Spalti, "Finding and Managing Web Content with Copernic 2000", *Library Computing*, Westport, pp. 217-221, Volume 18, no. 3, September 2000.
- [8] A. Sugiura, K. Yoshiyuki, "Internet Scrapbook: Automating Web Browsing Tasks by Programming-by-Demonstration", *Computer Networks and ISDN Systems*, Volume: 30, Issue: 1-7, pp. 688-690, April 1998.
- [9] B. Krulwich, "Automating the Internet – Agent as User Surrogates," *IEEE Internet Computing*, Volume: 1, Issue: 4, pp 34-38, July-Aug. 1997.
- [10] M.G. Wales, "WIDL: Interface Definition for the Web", *IEEE Internet Computing*, Volume 3, Issue 1, Jan.-Feb. 1999.
- [11] W.H. Yen, M.J. Hwang, H.R. Ke, "Integrated Search of Digital Library", *Proceedings of 2000 Taiwan Area Network Conference*, pp.484-491, October 2000.
- [12] WIDL, <http://www.w3.org/TR/NOTE-widl-970922>

- [13] SOAP, <http://www.w3.org/TR/SOAP/>.
- [14] Web Services Description Language, <http://www.w3.org/TR/wsdl>
- [15] Web Services Activity, <http://www.w3.org/2002/ws/>
- [16] Microsoft Scripting Technologies, <http://msdn.microsoft.com/scripting/>.
- [17] Readerware, <http://www.readerware.com/rwreviews.html>
- [18] Meta-Search Engines,  
<http://www.lib.berkeley.edu/TeachingLib/Guides/Internet/MetaSearch.html>
- [19] BestBookDeal, <http://www.bestbookdeal.com>
- [20] BestWebBuys, <http://www.bestWebbuys.com>
- [21] Readerware, <http://www.readerware.com>
- [22] Metalib, <http://www.exlibrisgroup.com/metalib.htm>
- [23] A Gentle Introduction to XML, <http://www.tei-c.org/P4X/SG.html>
- [24] Scripting Technologies, <http://msdn.microsoft.com/scripting/default.asp>
- [25] “Document Object Model (Core) Level 1”, World Wide Web Consortium,  
<http://www.w3.org/TR/2000/WD-DOM-Level-1-20000929/>
- [26] “Document Object Model (Core) Level 2”, World Wide Web Consortium,  
<http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/>
- [27] MSDN Home Page, <http://msdn.microsoft.com>
- [28] Consortium on Core Electronic Resources in Taiwan,  
<http://www.stic.gov.tw/fdb/index.html>