The University of Nottingham
School of Computer Science

A Framework for Interactive End-user Web Automation

Essam Eliwa

Thesis submitted to the University of Nottingham
for the degree of Doctor of Philosophy

February 2013

To my parents

Abstract

This research investigates the feasibility and usefulness of a Web-based model for end-user Web automation. The aim is to empower end users to automate their Web interactions. Web automation is defined here as the study of theoretical and practical techniques for applying an end-user programming model to enable the automation of Web tasks, activities, or interactions. To date, few tools address the issue of Web automation; moreover, their functionality and usage are limited.

A novel model is presented, which combines end-user programming techniques and the software tools philosophy with the vision of the "Web as a platform." The model provided a Web-based environment that enables the rapid creation of efficient and useful Web-oriented automation tools. It consists of a command line for the Web, a shell scripting language, and a repository of Web commands.

A framework called Web2Sh (Web 2.0 Shell) has been implemented, which includes the design and implementation of scripting language (WSh) enabling end users to create and customise Web commands. A number of Web2Sh-core Web commands were implemented. There are two techniques for extending the system: developers can implement new core Web commands, and the use of WSh by end users to connect, customise, and parameterise Web commands to create new commands.

The feasibility and the usefulness of the proposed model have been demonstrated by implementing several automation scripts using Web2Sh, and by a case study based experiment that was carried out by volunteered participants. The implemented Web2Sh framework provided a novel and realistic environment for creating, customising, and running Web-oriented automation tools.

All praise be to Allah, the most gracious, the most merciful

Acknowledgements

It is hard to acknowledge everybody that helped me to start, continue, and finish this work. I attempt my best and apologise dearly to those I did not mention.

I would like to extend particular thanks to my supervisors, Dr. Colin Higgins and Dr. Peter Blanchfield, for their support, guidance, and comments on this thesis, and for their moral support over the time of my study.

Foremost, I express my deep gratitude to my dear friends Ashraf AbdelRaouf, Mohamed Sabri, Rami Ghorab, P. Christofi, A. Soaifan, I. Balčaitė, and M. Al-Momani for all their support in many aspects during my study years.

The help of a number of computer science researchers has been essential at the early years of my studies, so special thanks to A.Tsintsifas, S.Bagley, I.Celik, M.Meccawy, M.Begum, S. Myo Htwe, and C.Schoreels.

The Phd for an international student is a lonely journey without great friends. I was lucky to meet so many of them in Nottingham; they have been like a second family away from home, so special thanks to A.Sabra, A.Khalifa, M.Mansour, A.Freewan, A.Basiouni, N.Wahba, Y.Saber, M.Sobhi, S.Abdulkadir, M.Ali, A.Soghier, K.Saleh, G.Frisso, L.McCormack, J.Durkin, A.Galati, E.Bachis, and A.Mangood.

Special gratitude to all the Egyptians who have participated in the 25th January revolution, mostly the ones who lost their lives in all of Egypt's Tahrir squares.

There are no adequate words to express my gratitude to my parents for supporting me emotionally, spiritually, and financially. I am also forever thankful to Ayman and Ola Eliwa for being great and supportive siblings in my most difficult moments. I am indebted to my whole family, especially to Dr. Hesham Khalil, my cousin who encouraged and inspired me since I was a very young. Finally, sincere love and gratitude to my wife Halah Yasser, whom I was blessed to marry, and who supported me to complete such a long life-changing journey.

Thank you all, Essam Eliwa

Table of Contents

List of Tables

List of Figures and Illustrations

List of Symbols, Abbreviations, and Nomenclature

Symbol	Definition
AJAX	Asynchronous JavaScript and XML
AST	Abstract Syntax Tree
CUI	Character User Interface
CSV	Comma-separated values
EBNF	Extended Backus–Naur Form
GUI	Graphical User Interface
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
JSF	Java Server Faces
MIME	Multipurpose Internet Mail Extensions
OOP	Object-Oriented Programming
Web	World Wide Web
Web2Sh	Web 2.0 Shell (Framework)
WSh	Web Shell (Scripting language)
XUL	XML User Interface Language
YQL	Yahoo Query Language

Chapter One: **INTRODUCTION**

This thesis presents an investigation that builds upon advanced and modern techniques for enabling customised automation of the Web interactions. It introduces a novel and innovative approach, which enables more useful and effective implementation of end-user programming to automate simple and sophisticated Web activities such as communicating, querying, and aggregating information.

This enabling approach was put into practice by designing and implementing a Web-based framework, which was named Web2Sh (Web 2.0 Shell). This framework empowers end users to write, customise, and run Web automation scripts that reflect their own needs and experiences. In addition, users can share these tools. With time, the aggregation of the users' generated scripts will extend the framework leading to constant growth of Web automation tools.

Web2Sh presents an enabling technology for the shift towards the Web as a platform. It interconnects with various research fields. Yet, the focus of this research is the study and investigation of applying an end-user programming model for the automation of Web tasks, activities, and interactions. This thesis refers to this research topic as *end-user Web automation* as a focused area that combines end-user programming with Web standards and enabling technologies.

One of the premises of this research is that the Web is now the most important source of general information and one of the most important technological developments of our generation (Brin and Page 1998, Jeffrey I. Cole, Suman et al. 2003). It is also the most widely used platform for querying, accessing, and researching information, among other usages (Jeffrey I. Cole, Suman et al. 2003, William, Ellen et al. 2009). Hence, there is a need for improved and innovative approaches for searching, retrieving, and manipulating information over the Web and namely for approaches enabling casual and experienced programmers to achieve more expeditious, efficient and customised Web automation.

Most people experience the Web as an *end-user* of packaged services/resources. Unfortunately, creators of these services cannot know the details of each job an end-user is attempting. It is simpler and more efficient to the end users when they can

directly build and customise tools that they need. An end-user (casual programmer) is a user who has limited or no programming experience. However, this type of user may need to write or modify programs in order to accomplish some repetitive Web-related tasks.

End-user programming is a recognised approach to allow end users to create novel programs in a simple manner. Spreadsheet programs are often considered to be a very good example of success in end-user programming (Margaret, Curtis et al. 2003), allowing them to write, edit, and use formulas and automation programs such as Microsoft Excel.

End-user programming environments are also useful for expert users and even programmers. This is due to the use of scripting languages, which are fast and easy to learn compared to high-level programming languages. In platforms such as UNIX shell, considerable amounts of time and effort can be saved by writing task automation programs.

The hypothesis of this work is that the development of an interactive end-user programming UNIX-like model for the Web, as a platform in its own terms, is feasible and useful. Such a model would facilitate the process of automating both simple and sophisticated Web users' tasks. Interactive in this context refers to the ability to accept and respond to commands from the end-user.

Web browsers are the most commonly used client for accessing, navigating, and interacting with the Web. Various Web browsers are currently in use, and even different versions of the same browser are common. This leads to the co-existence of a wide variety of browsers. Hence, for a Web end-user (hereinafter, *end-user*) programming environment to be widely used, it must operate on all, or at least the majority of, Web browsers and on any type of computer or smart device that can be used to access the web.

This thesis presents research on a new model of end-user Web automation. The model is evaluated through the design, implementation, and testing of a Web-based framework (Web2Sh) for writing, customizing, and running Web automation tools.

This chapter presents the motivation, objectives, and approach used in this work and its scope. At the end of the chapter, an overview of the implemented framework and the structure of the thesis are provided.

1.1 Motivation

This is an innovative research endeavour targeting a novel area of research. A main objective of this research has been to investigate and experiment with the design, development, and usage of an extensible Web-based framework for end-user Web task automation.

The presented framework explores a new approach, which allows data on the Web to be more effectively utilised through better discovery, integration, and automation. This would enable the end-user's Web interactions to achieve new and greater possibilities.

The approach implements a successful model that has been available and tested on various platforms, most notably UNIX, for automating end users' repetitive and tedious tasks. This model is useful for achieving automation through the effective use of scripting languages for end-user programming, which runs over a platform-specific shell via a command line interface.

This research shares the vision of the *Web as a platform* (O'Reilly 2007a). Hence, this work integrates the development of Web2Sh framework, which targets the Web as a platform and uses the thin client model to provide a Web command line for the Web. It also presents a specific Web shell scripting language (WSh) in addition to a variety of Web automation tools in the form of *Web commands* that can be linked together using WSh to cater for sophisticated interactions.

The thesis emphasises that the development of effective end-user Web automation tools is important, useful, and feasible; the current available environments for building such tools are both limited and hardly practical for the majority of users.

It provides a solid foundation for future research into end-user Web automation, and has potential wide uses for diverse Web research topics. Web2Sh offers a new development framework that can be used by researchers to facilitate the implementation and testing of new innovative ideas in various domains. This was demonstrated in this research by the implementation of some useful Web commands inspired by research from the data-mining domain.

In addition, available academic research focusing on the automation of Web tasks is limited. Although major Web companies, most notably Yahoo, have implemented systems related to the automation of end-user Web activities, they have hardly produced a lot of pertinent academic literature. This indicates the importance of and the need for greater research effort that focuses on this novel research topic. The potential of this field has inspired this research to investigate, building on the current literature and systems, in order to expand it so to provide the academic research community with a more solid foundation for future research.

This research and the implemented framework takes a step forward towards fulfilling the *Web as a platform* concept (O'Reilly 2007a), which is a major scientific challenge to the Web research community. The design and development of Web2Sh framework enables this concept by providing an environment for building useful and flexible automation tools (Web commands). Another challenge that this thesis addresses is the investigation and development of a robust and powerful repository of useful, customisable, connectable, and extendable Web commands, which can handle a wide range of users' activities.

The main motivation for this research was therefore to investigate such a novel research topic of end-user Web automation, which can be of great use in the current shift towards fulfilling the vision of the Web as a platform. This research topic could offer many potential benefits for various other Web research domains. In addition, the work provides a significant benefit for end users with the development of an extendable Web commands repository that facilitates the automation of a wide spectrum of Web tasks.

1.2 Background

There has been a clear shift towards utilising the Web as a platform (O'Reilly 2006a, O'Reilly 2007a) in order to provide many services and applications previously only available on desktop computers. There is an obvious need to provide mechanisms and tools on the Web that support the move towards the full vision of the Web as a platform.

Whois Internet Statistics[1] estimated 28.7% of the world population in 2010 were active Web users, with a 444.8% usage growth in the prior 10 years. It also states that the Web serves a total number of approximately 131 million domain names in 2011.

Researchers have identified a variety of motives for Web usage, although the specific number and content of the categories of those motives differ. According to (Jackson 1999), the five main motives for using the Web are: interpersonal communication, personally valued information, self-expression, entertainment, and consumptive motives. A more recent study by (Kennon, Rodgers et al. 2002), involving journalism college students via an open-ended questionnaire, identified four primary types of motives for using the Web: communicating, researching, shopping, and surfing.

In 2002 the ten most popular Internet activities according to (Jeffrey I. Cole, Suman et al. 2003) were:

1. E-mail and instant messaging;
2. Web surfing or browsing;
3. Reading news;
4. Accessing entertainment information;
5. Shopping and buying online;
6. Hobbies;
7. Travel information;

[1] http://www.whois.sc/internet-statistics/

8. Medical information;
9. Playing games;
10. Credit card tracking.

Recent research in the United Kingdom (William, Ellen et al. 2009) reported that Internet users were more likely to use search engines compared to previous years. It also stated that the Internet has become their primary source of information. A significant increase of the use in social networking sites in 2009 is also obvious in the report. It stated five main categories of reasons for using the Web: seeking information, communication and social networking, entertainment, finance and government services, and production.

Most Web applications offer their services through a GUI based interface, usually a Web browser, although programmers can write their own programs to automate any customised task they require. The majority of end users cannot easily achieve such automation due to the major lack of customisable automation tools.

Numerous Web crawlers and bots roam today's Web. These are programs that browse the Web, and follow links and process the Web in order to perform a variety of services such as: full text indexing (Brin and Page 1998), product comparison (Yun, Satya et al. 2007), Web data mining (Raymond and Hendrik 2000), downloading, and visualisation. However, this list of services is by no means sufficient for users growing needs. Users often require custom services that no existing robot provides.

With the rapid growth of the Web, a parallel growth in demand for personalised Web automation is expected. This will allow a user to delegate repetitive tasks to a robot or a script that can generate alternative views or summaries of Web content that meet the user's needs (Miller and Bharat 1998).

Both Semantic Web and Web 2.0 communities aim to achieve a superior Web, by investigating novel approaches that allows data to be used for more effective discovery, integration, collaboration, and automation (Berners-Lee and Miller 2002b, O'Reilly 2007a).

The research (Bolin, Webber et al. 2005b, Berners-Lee, Hall et al. 2006, O'Reilly 2007a, Ankolekar, Krötzsch et al. 2008) indicates that for the "Web as a platform" vision to be realised, end users still need more efficient, accessible approaches for automating the Web. However, the semi-structured nature of the Web, its vast size, variety of formats, languages, and standards, and above all the GUI-HTML based interface of the Web all present challenges for automating Web access, querying, and manipulation.

According to (Bolin 2005a), end users still need an efficient, unified and simplified way to accomplish Web-based tasks, particularly tedious tasks of repetitive nature. There is still no universally applicable and flexible system envisaged that fulfils such needs. Thus a system, with built-in extensibility to allow users to add and adapt new tools, is highly desirable.

There have been recent and current efforts on the Web (Miller and Bharat 1998, Pilgrim 2005, Bolin, Webber et al. 2005b, O'Reilly 2007b, Beard 2008) to provide users with the required environments and tools to enable them to achieve a higher level of personalised Web automation. However, it should be noted that this research approach is different and novel in the sense that it investigates the use of simple command line operation from within any modern Web browser, which is by far easier, faster and more convenient to use. Furthermore, it utilises the use of data streams, filters, and pipes similar to the UNIX shell approach that has already been tested and widely used by various computer communities (Delaglio 1995).

One of the aims of this research is to offer broad tools that can access data in its various formats on the Web. Few known tools offer end users with environments for building tools to aggregate and manipulate Web content. The most recognized web-based tool is Yahoo Pipes[2] (Fagan 2007), yet it can only access specific formats, mainly RSS feeds, with limited support for raw HTML pages - by far, the most commonly used format for Web pages. The development of advanced automation tools is a major scientific challenge to the research community. Such tools need to cope with the ever-increasing complexity of the Web and be flexible enough to adapt to its frequent and rapid changes.

[2] http://pipes.yahoo.com/pipes/

The introduced scripting language for Web2Sh framework, named WSh, enables end users to aggregate Web commands in innovative approaches to accomplish tasks that the original commands designers may have not even foreseen. This could later support the development and extension of robust, flexible, adaptable, and powerful automation tools.

This provides significant benefits for end users. Indeed, such tools would help to alleviate the tedious effort and save time currently needed to process Web retrieved data. It also presents new opportunities to exploit the Web for both scientific and business purposes. Improvements in this field will lead to significantly more efficient use of resources, both of the users' time and the advanced utilisation of the resources themselves.

Developers of desktop applications have relatively full control over their user experience. However, on the Web, there is virtually no control over how clients access, view, and use Web pages. Having this control on the desktop made it easier for developers to build tools that end users can use to write and customise scripts to automate a variety of tasks on platforms and applications. A good example of such a platform is UNIX, where users could use any available shell scripting language to automate a certain task. End users also have automation tools in various applications, such as Microsoft Office, which offers the Visual Basic scripting language.

Most end users search for information using Web browsers with search engines as a starting point; however, this retrieval mechanism does not necessarily address all of the users' needs. The end users main concern is to find all the information they seek swiftly and accurately. However, with the vast size of the Web, achieving this is usually a challenge (Madria, Bhowmick et al. 1999).

The huge growth of the data accessible on the Web presented two fundamental issues regarding the effectiveness of accuracy of information search: *mismatch* and *overload*. Mismatch means some useful and related data has been disregarded, while overload means some assembled data is not actually related to what users need (Yuefeng 2006).

In many cases, end users may require information that they need to filter and aggregate from various search engines or specific Web pages, e.g. users may need to retrieve a set of articles from various news sites related to a certain topic. Current browsers and search engines give users very limited options or tools to program or customise their own automation programs that may fulfil their own information needs.

Recently there has been an increased research interest by major commercial Web players such as Yahoo, Google, and Microsoft. They have started to provide Web-based tools that facilitate Web activities automation for end users. These systems include, Yahoo Pipes[3], Microsoft Popfly (Foley 2009), and Google Mashup Editor[4]. Although both Popfly and Google Mashup Editor were shutdown in their beta stage, they showed potential and provided new ideas that did meet a recognizable success. The only available statement by Microsoft about the closing of the Popfly project stated that they made great progress and learned many lessons yet had to close it due to the economic situation (Bishop 2009). As for Google, they announced that they used the gained experience from the Mashup Editor project in their App Engine[5] project (Tholomé 2009).

Even in its beta version, PC World magazine picked Popfly as one of the most innovative computing and consumer electronics products of 2007 (PC-World-Staff 2008).Yet, it was clear from the type of tools created by Popfly users, that it was more suited as a fun gadget rather than serious Web automation. The situation was different with Google Mashup Editor; it was more suitable for expert users and actual programmers. This made it more powerful compared to other systems, but at the same time would have been harder to utilise by the wider end users community. The system was not even published to the public and was only available to be tested through requesting invitations.

[3] http://pipes.yahoo.com/pipes/
[4] http://code.google.com/gme/
[5] http://code.google.com/appengine/

Other research teams also developed other tools. For Example:

- Chickenfoot (Bolin 2005a), which was developed as a plugin for the Firefox Web browser.
- cURL[6], is a desktop-based command line utility for executing a variety of functions with URL-oriented protocols.
- Fiddler HTTP Debugging Proxy (Lawrence 2008), which logs all HTTP traffic between a computer and the Internet.

Such approaches offer significant promise for solving specific problems, however, the investigation of such approaches and building more robust general system should not be underestimated. These systems have been targeting special areas, rather than as a universal solution to a wider range of problems, and as such, they provide only limited functionality, although they demonstrate the feasibility of work in this topic.

One of the first general tools targeting this domain was Yahoo Pipes, which introduced a service to aggregate Web content (O'Reilly 2007b). Similar to UNIX pipes, Modules can be combined together to create the desired output. Yahoo launched this service in early 2007. Since then many users have been creating pipes with various functionality. (O'Reilly 2007b) considers Yahoo Pipes as a milestone in the history of the Internet, and describes it as "still a bit rough around the edges, it has enormous promise in turning the Web into a programmable environment for everyone."

At this time, Yahoo Pipes is perhaps the most useful Web automation tool that has been publicly available to end users (Fagan 2007, O'Reilly 2007b). However, it is particularly limited in intent and functionality; it mainly works for RSS feeds, with minor exceptions. In addition, it depends on a graphical interface for building users' pipes, which affects the pipe creation process in terms of performance and speed. It also makes it suitable only for powerful computers and close to impossible to utilise over smart devices. While Yahoo Pipes users can combine multiple searches into a

[6] http://curl.haxx.se/

single feed, there is limited functionality for exploiting extracted data in other, more flexible and powerful ways.

The Yahoo Pipes graphical user interface (GUI) makes it easy for novice users to create modules. However, for more advanced users and programmers it would be faster and more efficient to have a text-based editor where users can write code rather than using the drag and drop GUI interface. Hence, it is preferable and desirable to offer a faster text-based command line especially when creating scripts that are more complicated.

The author of this thesis shares the view of (O'Reilly 2007b) that "Most of the non-graphical utility programs that have run under UNIX since the beginning, some 30 years ago, share the same user interface. It is a minimal interface, but one that allows programs to be strung together in pipelines to do a task that no single program could do alone. Most operating systems including modern UNIX and Linux systems have graphical interfaces that are powerful and a pleasure to use. But none of them are so powerful or exciting to use as classic UNIX pipes and filters, and the programming power of the shell." This indicates a preferable non-graphical interface that can provide similar and more advanced functionality.

This research provides an interface suited for both regular and advanced users. Many Web2Sh Web commands have the option to produce an RSS feed (by piping to an RSS filter at the end of any command) so that the resulting feed may then be used from within Yahoo Pipes or any other similar tool.

Calls for combining Web services as the Web analogue to UNIX pipes have occurred (Jeffrey 2007) but have largely gone unheeded. Expanding this concept to a general system where the Web can be seen as a library of callable components has also been partially envisaged by Jon Udell who had a vision of Web sites as "Data sources that could be re-used, and of a new programming paradigm that took the whole Internet as its platform" (O'Reilly 2007b). This research not only investigates, improves, and implements these ideas, but also tries to take the concept further.

To meet its objectives, this research investigates and utilises major current Web trends, such as Web 2.0 that work towards enabling the Web as a platform mainly by providing usable desktop-like applications and enabling further the move towards more interoperable applications that reside on the Internet space rather than on users own computers.

In the Semantic Web approach, end users are mainly consumers. Semantic Web applications mostly depend on developers and experts rather than end users to participate in supporting the conversion or to add semantics themselves. On the other hand, one of Web 2.0 objectives is to empower end users to be more effective.

Web 2.0 facilitates the adding of users' content and customises their own views of the Web. This could be one of the main reasons for the popularity of Web 2.0 jargon; however, functionalities offered to end users by Web 2.0 applications usually limit their capabilities. They usually cannot extend or modify beyond what the developers of those applications offer.

This research also aims to investigate how to empower end users further in order to play a more effective role in the transition to the Web as a platform. Web2Sh provides a set of simple and advanced tools that end users can extend in unlimited ways using the piping system. This enables end users to create their own tools or simply to use tools created by others. A text-based scripting editor provides a command line approach similar to what is available on any other platform and that the Web still lacks. Although many may consider the Web address bar as a command line in itself, but it has very limited functionality.

1.3 End-user Web Automation

This research uses the term *Web Automation* to refer to the enabling systems and technologies, which support end users to write and run automation tools that express their own personalised Web experiences.

An end-user programmer (casual programmer) is defined as a user who is not concerned with programming, yet who needs some of the power of programming to accomplish tasks effectively (Nardi 1993). On the Web, end users normally use GUI-based applications, mostly Web browsers, for various objectives like information

retrieval, shopping, communication, and entertainment. Any task on the Web usually involves a set of link clicks and forms submission to reach a required goal. For similar goals, users would usually follow a sequence of similar steps. Usually the users' own experiences are the main factor affecting the speed and efficiency of reaching a goal.

The research topic of *end-user Web Automation* in the context of this thesis is defined as *the study of theoretical and practical techniques for applying an end-user programming model to enable the automation of Web tasks, activities, and interactions.*

The end-user Web automation is a novel research topic, which connects together various computer sciences fields. This work involved the study of following topics:

- Web Enabling Technologies;
- End-user Programming;
- Programming Language Processors;
- Software Engineering;
- Concurrency;
- Information Retrieval.

1.4 Research Issues

The current state of the Web leads to the conclusion that there is still plenty to be done to fulfil the move towards achieving the "Web as a platform" vision (O'Reilly 2005, O'Reilly 2006a). The current embracing of the Web has fundamentally changed software development approaches. In the past few years, the Web has become the most used deployment environment for software systems and applications.

At present, end users utilise a Web browser more widely to run desktop-like applications. In the near future, developers will write the vast majority of software applications as a Web application or at least with considerable Web integration (Taivalsaari, Tommi Mikkonen et al. 2008). However, there is still a significant need for environments and tools to enable end users to achieve the same level of task automation they find using desktop-based platforms such as Operating Systems, Spreadsheets environments, and many other applications that offer the use of scripting languages and basic command line tools.

1.4.1 Scope

The main scope of this research is to investigate the feasibility and importance of enabling end users to automate and share their Web experiences. This research shows the main aspects of the proposed Web2Sh programming platform. Web2Sh is a framework for writing and running Web commands. It offers a Web-based shell scripting language, Web commands repository, and a Web-based command line interface.

To accomplish such a platform, this research must explore many computer science domains to reach the best technical approach and design for achieving the thesis target. However, the focus of the research is *Web automation* for end users.

In addition, the research investigates a set of Web commands to execute browsing, data mining and filtering tasks. These commands provide a multiplicity of functions such as retrieving Web pages, string manipulation, data mining, and Web page segmentation.

1.4.2 General Aims and Objectives

The main aim of this research is to investigate a new approach for Web task automation. Such an approach will provide Web users with flexible and fundamentally more powerful ways for automating their Web interaction and in particular for information retrieval, data mining processes, and constructing Mashups. This approach enables Web end users to create, test, and run script-based tools that can execute a sequence of steps required to achieve a given goal.

The overriding aim is to offer a more flexible approach in order to create and share automation scripts in the form of "Web commands" that offer an innovative way for users to interact with the Web, automate their activities, and share their Web experience with other Web users.

In this context, this research investigates the development of a framework named Web2Sh (Web 2.0 Shell, based on the term Web 2.0), which offers a command line interface for the Web, a simple scripting language and a user friendly environment for building Web commands in a similar approach to UNIX platform shell scripts.

To achieve such Web automation, Web2Sh offers further flexibility that allows end users to interact with the Web and a great level of generality that facilitates the development of tools that end users can employ in a wide range of situations and environments. Furthermore, Web2Sh offers end users a new model to retrieve information, create customised pages from multiple resources, and automate various Web tasks.

Web2Sh also offers a new model for sharing users' Web interaction experiences in the form of customisable Web commands. It also offers new approach for Web information retrieval and automating Web activities through a command based interface rather than the common GUI link click common model.

For achieving those goals, Web2Sh emphasises two major paradigm shifts in the Web:

1. User-generated content: This relates to enabling individuals to become an effective part of the Web and to produce their own information, i.e. blogs, wikis, social bookmarking, and social networking sites.

2. Thin client computing: this indicates the model where data and applications are stored on Web servers, and users can access them from any computer via a Web browser. Cloud Computing is a clear and exact application of this model.

In the light of these paradigm shifts, this research investigates how far the proposed model can enable end users to achieve Web automation. In addition, an ambitious objective is to offer end users new, easy, and fast methods to interact with the Web especially with the use of small devices like smart phones. Web2Sh is able to operate fundamentally faster and in fewer steps to retrieve information from the Web; once a user creates or learns to use a Web command he/she can then immediately and easily put it into use (e.g. to retrieve information) by typing this command in the Web2Sh command line. In addition, the user can easily manipulate or customise the retrieved information by extending the Web command.

Another ambitious aim of the research is to investigate other domains of Web technologies in order to support the development of a useful Web commands repository, such as a set of commands specific for Web data mining.

Data quality is an important issue that represents a problem when applying data mining techniques to the semi-structured data of the Web, as the unstructured Web data requires the development of powerful and more efficient tools. Such tools should support the discovery and analysis of interesting, unforeseen or valuable structures in the Web data as defined in (Madria, Bhowmick et al. 1999).

The work presented in this thesis offers useful and extensible tools that implement some data mining techniques. This offers the end-user community with direct access to utilising such techniques themselves.

1.4.3 Specific Objectives

The key objectives of this research are:

3. Investigate a new model for end-user Web automation. This model should offer an easy to use approach to automating Web tasks.

4. Design and implement a new simple and effective scripting language focused on the development of Web automation tools.

5. Design and implement a Web-based framework to enable end users to create, run and share Web automation tools.

6. Design and implement a repository of Web commands, as a reusable, efficient and customisable Web automation tools.

1.4.4 Potential Users

End users are a diversity of unique individuals each with their own experiences, goals, skills, and interests. The development of a command-line-based interface for the Web, a Web scripting language, and a framework for developing Web commands, would provide the community of Web end users with powerful tools that would enable them to both save time and share their Web experience. The ability to extend the system by collaboratively developing and sharing Web commands would allow Web users extra flexibility by giving them the tools they require to search, interact and customise the Web according to their own needs.

In defining the end-user (Nardi 1993) does not classify them as novices and experts, but by the way which they represent various domains, such as academics, librarians, doctors, students and researchers. He also defines end-user programming systems as software that end users can use to develop and run applications; a good system should allow end users to build their tools with the minimum amount of work.

One of the general aims of Web2Sh is to serve the wider possible set of users. Hence, the framework offers a set of general use commands that meets the most common activities shared among end users. However, more specific Web commands that target specific end-user communities were developed, in order to demonstrate further the feasibility and usefulness of the approach implemented by Web2Sh framework.

An important advantage of Web2Sh is that end users can create and share Web commands in a simple way. The success of social bookmarking, blogging, wikis, and other sites, where users are encouraged to participate actively, implies that giving end users the chance to contribute to creating Web shell tools would be very promising.

One community of prospected users of Web2Sh is the wide researchers' community. Almost all researchers in various fields use the Web to research, access and publish information. In many cases they may need to collect data from variety of Web sites for certain analysis reasons, for example, collecting a huge number of images, then extract their `img` tags' `alt` attribute value for further accessibility analysis. For researchers with a computing background it would be easy for them to write or customise an existing program for the task they require, however, for the majority of researchers they could waste time doing such jobs manually and repeatedly.

Another type of prospective user is Web administrators, who work with diverse tools to manage, monitor, configure, troubleshoot, and continually update their Web sites, which can be distributed across multiple networks and servers, in a rapidly paced growing and changing Web of information and services.

They may in some cases build and share custom tools for automating specific tasks that are not supported by out of the box tools. The recent movement towards dealing with the Web as a platform, in addition to using desktop-like Web-based management consoles, offers many advantages. Yet, this puts an extra burden on Web administrators, as achieving customisation and automation requires a decent level of Web programming experience, which in many cases will be beyond their skills, hence making it difficult to write tools with the level of customisation they might need.

Field studies in (Eser, Eben et al. 2005) indicate the importance of collaboration among system administrators and end users in the development and use of custom automation tools in organisations.

The methodologies this research employs aim to provide an accessible, usable and productive framework, which empowers end users to automate their Web experiences. This framework should serve end users with little or no formal

programming training as well as users with more expertise. Hence, the design of Web2Sh caters for the general need of regular Web users, particularly for casual programmers. For these, programming is not their main trade; they only use it to achieve certain tasks they may need. This type of user is not willing to spend a long time learning a sophisticated programming language, but can invest a few hours in learning a language that might help them to save time and increase their efficiency by automating repetitive systematic tasks they need to do.

1.4.5 Approach

This research took the following approach in order to achieve the objectives specified in section 1.4.3:

First, to acquire theoretical and practical understanding, the research investigates the best known tools and Web applications that offer various approaches to full or partial support for end users to create and share customisable automation tools (Miller and Bharat 1998, Aquino 2005, Pilgrim 2005, Bolin, Webber et al. 2005b, Fagan 2007).

Secondly, an investigation is made of the end users' most common activities and possibly useful basic Web commands that can empower them to automate their most repetitive tasks in a fast and effective manner.

Thirdly, the research investigates state of the art Web development technologies and design techniques, which may serve the design and implementation of Web2Sh framework.

Fourthly, Web2Sh framework implementation was accomplished through multiple iterations of design and development, in order to realise the highest possible level of performance, extensibility, effectiveness, and usability.

Fifthly, the framework was initially evaluated by writing, testing, and running a set of Web commands that target possible common scenarios, which various domains of end users could require.

Finally, a mixed method approach was used to examine the hypothesis of this thesis. An experiment encompassed fulfilling five tasks that complemented each other. The overall objective of the five tasks was to integrate data from Google product search

and Amazon Web services in order to create a mashup of a set of dynamic links that contain an associate id of an amazon affiliate. In addition to completing their tasks, each participant was asked to provide feedback through online questionnaires, in which both quantitative and qualitative data was collected.

1.5 Contributions

The primary contribution of this research lies in the novel area of *Web Automation*. The combination of end-user programming technique, and the power of the Web as a platform, facilitates the creation and running of user-oriented automation scripts in the form of Web commands. A flexible and powerful framework (Web2Sh) for creating, customising, and running Web Automation scripts has been provided. In addition, a repository of useful Web commands is provided for end users.

A second contribution is in the field of end-user programming. A Web scripting language (WSh) has been developed for the generic use of creating and customising Web commands. WSh is a task-oriented language that applies the concepts of the software tools philosophy.

1.5.1 Web2Sh Framework Overview

This thesis presents a new Web-based programming platform "Web2Sh" for creating, sharing, and evaluating Web commands. Web2Sh framework, in the context of this thesis, refers to components including a command line Web interface, Web WSh shell script, interoperability and sharing model and a repository of core Web commands that handles many basic and composite Web activities.

The proposed system aims to serve and support the move toward the concept of "Web as a platform." The research investigates this concept and the limitations regarding Web automation in terms of ease of use, reusability, flexibility, and limited tools, which prevents utilising this concept to its full potential.

A main component of Web2Sh is a UNIX-like shell scripting language for creating Web scripts. Extended Backus–Naur Form (EBNF) is used to create the formal grammar for the WSh scripting language; the language itself is initially generated

using the JavaCC tool (Kodaganallur 2004), and then extended using the Java language.

Web2Sh introduces a repository of *Web commands*, which is the main programming unit of the shell; a Web command is a tool that executes a certain task over the Web platform. Combining multiple Web commands is easy using the pipeline concept. The pipe idea is to pass a command's output into another command for extra processing.

In addition, Web2Sh supports the thin client-computing model, hence, allowing Web commands to utilise many server-side resources that have been usually only accessible by developers. This model also makes Web2Sh more usable as all end users need is a Web browser to write and run their Web commands.

End users can further utilise and customise Web commands to construct new reusable Web commands. A primary advantage is that much of the functionality needed by Web users can be accessible in one place. Furthermore, The WSh scripting language provides the ability to extend Web commands repository offered by the framework to cover new domains and functionalities.

Web2Sh utilises the power of the Java language and offers end users a usable, extensible, and powerful set of Web commands enabling them to increase the efficiency of how they accomplish various tasks on the Web, giving them the possibility to customise a set of their own commands to serve individual needs. Web commands are, from the deployment and execution point of view, server-side programs and services. Running on the server side, allows Web commands to access superior resources and make them easily accessible for the wider end-user community.

Many Web sites make their services accessible to developers through different interfaces, most commonly Web services. However, developers need to locate the right service URL and documentation, before being able to use it. Web2Sh empowers end users to access and utilise data across the Internet through one simple language, eliminating the need to learn advanced programming languages and how to locate and call different APIs.

While some systems exist to help search, retrieve, and process this information, there is a dearth of advanced, powerful, and flexible applications for both novice and expert users. Hence, this research investigates the ability of introducing a new approach for information retrieval, creating Web Mashups, sharing users' Web experience, creating Web-mining tools, and automating various Web tasks.

This research proposes a novel Web-based environment for end-user programming to enable end users to create their own specific automation programs targeted for the World Wide Web as *Web command* to achieve an improved utilisation of Web resources. This environment consists of a command line for the Web, a series of Web commands that targets a variety of Web activities, and a simple scripting language.

1.6 Organisation of the Thesis

This chapter presented the motivation, background, scope, and objectives for this research. It introduced the definition of the end-user Web automation, explained the approach, and highlighted the contributions. It also provided an overview of the developed framework (Web2Sh) and its potential users.

Chapter 2 introduces the concept of the Web as a platform, and provides an overview of Web history, architecture, and other enabling technologies with the focus on Web automation-related technologies and systems. It also provides an investigation of available systems and tools that provide partial support for end-user Web automation. It highlights and considers the advantages and disadvantages of each tool's approach. Finally, it provides an overview of the concept of end-user programming, scripting languages, and command line interfaces. It also provides an overview of the UNIX shell environment as a recognised example that implements end-user programming and shell scripting.

Chapter 3 present the Web2Sh framework. It presents the main design challenges, and describes solving the problem of developing a concrete useful framework for end users Web task automation. It presents in more detail the framework architecture and design. The implementation details of the pipeline concept are provided.

Chapter 4 provides a detailed description of the Web2Sh command line user interface. This is the client side component of the framework.

Chapter 5 describes the building process of Web2Sh dedicated scripting language (WSh). It explains the need for a scripting language to enable Web automation. It presents the software tools philosophy, and the background needed for defining a new scripting language and implementing its Parser; the design of WSh language is explained, and the language informal and formal specifications are introduced. Finally, it explains the parsing, translation, and interpretation of the code.

Chapter 6 presents Web command as the basic building block to writing Web automation scripts. A detailed description of the syntax, structure, and general characteristics of Web command is provided with detailed descriptions of some of the commands provided by Web2Sh. The Web2Sh implementation of the pipeline concept is discussed. Finally, the integration of external tools and services in Web2Sh is discussed.

Chapter 7 reports on the use of the framework, it demonstrates the usefulness of the framework through the implementation of various possible Web related activities. This shows that the framework facilitates the creation and running of Web commands in a wide range of domains.

Chapter 8 presents the evaluation of the experiment that was performed to examine the hypothesis of this thesis. The evaluation was carried out as a user trial, involving twenty-four participants. The user trial included the use of both Yahoo pipes and Web2sh to produce a set of modules/Web commands, with the objective of automating information integration between Google products and Amazon Web services.

Chapter 9 describes the foundations and arguments for the design and implementation decision throughout the development of Web2Sh framework, lays out the conclusions of this research, and suggests future work.

This thesis supports the hypothesis that the development of an interactive end-user programming UNIX-like model for the Web, as a platform in its own terms, is feasible and useful. Chapters 8 and 9 highlight these claims by reviewing the thesis main points and linking the model with the design and implementation of Web2Sh framework and meeting the objectives. It also discusses the contributions while indicating areas for future work.

Chapter Two: **RELATED WORK**

2.1 Introduction

This chapter provides the research background related to the novel area of end-user Web automation. It starts with a general overview of the Web, Web architecture, and its main enabling technologies, followed by a survey of related Web automation systems and tools, and finally examines the end-user programming model and UNIX shell scripting. A section is presented for each item.

Section 2.2 introduces an overview of the Web and the shift towards the vision of the Web as a platform. The advantages and limitations of the Web are considered, with a focus on the enabling technologies leading to Web task automation. A brief historical overview of the Web is provided, followed by research background in the fields of scripting languages, Web mining, Semantic Web, Syndication Feeds, Web 2.0, Rich Internet Applications, and user-generated content .A sub-section is presented for each field.

Section 2.3 defines the term *Web automation*, and then provides an overview of related systems and enabling techniques. A sub-section is presented for each of the best-known systems and tools. The advantages and disadvantages of each system's approach are considered.

Section 2.4 examines in more detail the concept of end-user programming. The role and characteristics of scripting languages are presented. The usefulness of command line interfaces is demonstrated. Finally, an overview of the UNIX shell environment is presented as a recognised example that demonstrates the utilisation of end-user programming and shell scripting.

2.2 World Wide Web

The World Wide Web (WWW, or simply the Web) is a vast information space in which the items of interest, referred to as resources, grow further every day encompassing more services and information in different languages and formats. At present there is a clear shift towards utilising the Web as a platform (Anderson 2006,

O'Reilly 2006a) to provide many services and application that used to be available only as desktop applications.

Successful utilisation of the Web as a platform unlocks many new possibilities and challenges given the number of simultaneous users, the numerous services, and the ever-growing information repositories. Web developers and demonstrators are usually the main information producers on the Web. They add new content, servers, and applications quite often, providing an endless supply of knowledge accessible by millions of users.

Since 2005, a new approach that encourages end users to produce information themselves proved very successful. This approach is known as User Generated Content (UGC). It covers a wide range of Web publishing and media content Web sites available in a range of modern technologies. It was successful in many domains including blogging, podcasting, forums, products reviews, social networking, and wikis (Marianna, David et al. 2008).

In desktop applications, developers expect to have control over the end-user experience. However, on the Web, there is much less control over how the user might view a Web page and on which browser. Hence, Web developers face many challenges taking into account all the variables, such as the Web browser type and version, which may have an effect on the end-user experience.

The use of well-known standard formats and protocols helps to solve these problems. However, many developers, vendors, and even users might not always adhere to these standards. This leads to another major challenge for creating general tools that automate Web-based tasks. Committing to tools that utilise Web standard protocols and formats is perhaps the best approach to cater for the majority of Web resources. Yet, it is also important to provide end users with some tools that can also handle the sites that may not adhere fully to the Web standards.

The Web will be more powerful when it becomes an environment which allows data to be used for more effective discovery, integration and automation (Berners-Lee and Miller 2002b). Both Semantic Web and Web 2.0 communities aim to achieve almost the same goals using different approaches. They appear to be competing rather than collaborating. However, future Web applications may make use of both of them, to

retain the Web 2.0 focus on community and usability, while utilising Semantic Web infrastructure to facilitate information sharing, integration and processing via automated tools (Ankolekar, Krötzsch et al. 2008).

Research by (Bolin, Webber et al. 2005b, Berners-Lee, Hall et al. 2006, O'Reilly 2007a, Ankolekar, Krötzsch et al. 2008) indicates that end users require automated tools that are efficient, accessible, customisable, and useful to accomplish various Web tasks. The semi-structured nature of the Web, the diversity of formats and languages, and above all the GUI-HTML based interface of the Web, raise many challenges for achieving such Web automation.

The Web is likely to be the logical choice as a platform for building and running a Web automation framework. Hence, the next section will investigate several aspects of the Web. First, it will offer a summary of the Web history, architecture, and models. Secondly, it investigates enabling Web technologies that should be considered for the architecture of such a framework.

2.2.1 A Brief History of the Web

The Web is a worldwide-shared medium for information, communication, broadcasting, and many other services. Web pages are usually arranged into Web sites, which are identified with domains, and connected using hypertext links. Information on a Web page can be hard coded directly as HTML content in static pages or retrieved dynamically from a database or XML-based documents allowing for a higher level of interactivity with its users.

Users are a major facet of the Web, the more users the Web has the more powerful it grows. Users participate in various Web activities through different devices, most commonly computers, mobile phones and smart devices. According to the World Wide Web Consortium and (Berners-Lee 2004), the Web history goes back to 1945 when (Bush 1945) published an article in Atlantic Monthly about a photo-electrical mechanical device called “Memex”. It had the advantage that it could make and follow links between documents on microfiche. later on, (Nelson 1965) coined the term "Hypertext".

A major milestone in Web history was on March 1989, when (Berners-Lee 1989) wrote "Information Management: A Proposal" at CERN labs. This introduced the main principles that the Web was built upon. By 1990, Tim Berners-Lee developed the tools necessary for a working Web: the first Web browser, the first Web server, and the first Web pages that described the project itself. By the end of 1993, Various browsers where introduced including the Mosaic browser for all common platforms, over 200 HTTP servers were available, and the Web (Port 80 http) traffic measures 1% of National Science Foundation Network backbone traffic (Berners-Lee 2004). In May 1994, the first International WWW Conference, organised by Robert Cailliau, at CERN labs took place. The World Wide Web Consortium was founded at the Massachusetts Institute of Technology (MIT) In September 1994, with Tim Berners-Lee as director.

Between 1996 and 2001 the Web expanded rapidly, many e-commerce Web sites were available and most major companies had a presence on the Web. The dot-com bubble occurred when many new companies started solely on the Web without having a realistic administrative ability; most of them lacked the required characteristics for a successful business model but were able to sell their ideas to investors because of the innovation of the dot-com concept. Inevitably, in 2001 the bubble burst, and many dot-com companies went out of business (Yann 2005).

At present, the Web is being used everywhere; according to statistics by Miniwatts Marketing Group (Group 2011), the Web was being used by 30.2% of the world population in March 2011, this percentage increased by 4.3 in June 2012 to become 34.3%. This percentage represents 2,405,518,376 users.

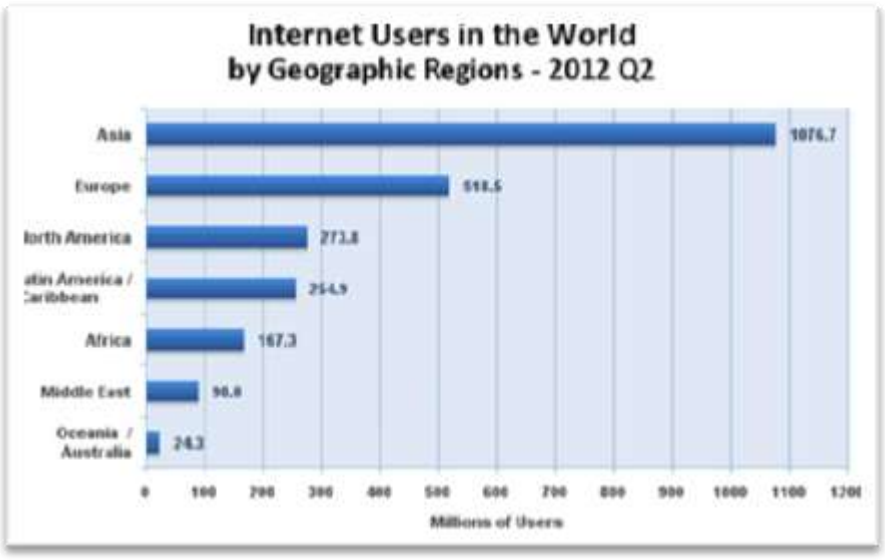

Figure 2-1: Internet users estimate for June 30, 2012. Source (Group 2011)

An important facet of the current era of the Web is social networking Web sites, such as MySpace, Facebook, and Twitter. UGC-dependent Web sites such as YouTube, blogs, and wikis are also a notable facet of the current Web.

2.2.2 Web Architecture

Various Hypertext Systems were proposed in the early 1990s. The World Wide Web was just one of those systems but it became the most widely used hypertext system on the planet (Hall, Hill et al. 1993). A defining characteristic of the Web is that it allows embedded references to a resource via a Uniform Resource Identifier (URI). The simplicity of creating hypertext links is a key reason for the success of the hypertext Web at the moment (Jacobs and Walsh 2004).

The Web is developing into an extraordinary platform (Berners-Lee, Hall et al. 2006), which keeps growing every day. The technologies that the Web was based on are remarkably simple, effective, and scalable. This enabled the Web to become a remarkable and growing information space of interconnected resources and services.

This section demonstrates a brief review of the main technologies underlying Web architecture; Information is mostly based on the W3C Recommendation (Jacobs and Walsh 2004), which outlined the core design components of the Web.

The three main architectural bases of the Web that were discussed in (Jacobs and Walsh 2004) are:

2.2.2.1 Identification

The Uniform Resource Identifier (URI) is used to identify resources. By entering a URI into the a Web browser address bar, or simply clicking on a Web link, the resource that this URI points to is retrieved alongside with any other embedded resources, such as images.

2.2.2.2 Interaction

Web agents communicate using standard protocols, e.g. HTTP, which enables interaction between the client and the Web server through an exchange of messages that adhere to a defined syntax and semantics. These messages would usually contain a representation of data and metadata. It is also important to provide handling for possible errors. For example, if the resource that the URI points to is not available on the server that would return an HTTP 404 error.

2.2.2.3 Representation

The Web supports a wide variety of formats; the most well-known format is HTML. Due to the nature of the Web, end users use various devices to access the Web. Therefore, ideally, the formats used for the Web should be device independent. The choice of an interaction protocol places limits on the formats that can be used to transmit these messages. For example, HTTP, which is the most commonly used protocol on the Web to retrieve Web pages, typically transmits a single octet stream plus metadata, and uses the Multipurpose Internet Mail Extensions (MIME) "Content-Type" and "Content-Encoding" header fields to further identify the format of the representation (Freed and Borenstein 1996).

To illustrate how these three elements work together, a URI refers to a resource on the Internet, which the client Web agent dereferences to try to access the representation of the resource identified by this URI. The client Web agent interacts with other agents to request and transfer the identified resource; this is done according to a set of standards and protocols, e.g. TCP/IP and HTTP. The resource

itself is represented in a specific format, e.g. HTML or XHTML, to allow for the right representation on the client Web agent, e.g. a Web browser.

URIs do not always refer to retrievable resources; some resources can be identified by a URI, yet cannot be retrieved from the Web, while other resources, such as Information resources, are resources that exist on the Web and can be retrieved from it (Jacobs and Walsh 2004).

The three bases of identification, interaction, and representation should be separated out to allow autonomous modular evolution of the Web architecture. Likewise, the various technologies should be able to evolve, independently, without affecting their interoperability with other technologies (Berners-Lee, Hall et al. 2006).

2.2.3 Web Scripting Languages

Scripting languages have been a very important player in the Web. Since the early start of the Web, scripting languages like Perl (Wall, Christiansen et al. 2000) were used to build the backend of Web sites.

Scripting languages proved their importance and power over the years, as they offered developers the power to implement their ideas quickly. Many scripting languages have been used on the Web, most notably PHP, Perl, ASP, ColdFusion, JSP, JavaScript, and Ruby. Scripting languages have the advantage of a fast learning curve and the ability to produce powerful programs with minimum programming knowledge (Ousterhout 1998, Loui 2008).

Scripting languages tend to be loosely typed (typeless) compared to other high-level languages such as Java, hence, are easier to learn. The typeless nature of scripting languages results in simplicity of connecting components, less code and more flexible programs. In addition, it minimises significantly the programmer's write time (Loui 2008), leading to the usefulness of using scripting languages for automating various activities on a verity of platforms. By offering a scripting language to their users, platforms can enable users to customise the behaviour of the service and utilise resources according to their needs.

Scripting languages, such as UNIX shells, operate on the basis that a set of useful components already exists in other languages. Scripts built using UNIX shells are often used to pull together a set of filter programs into one program. This is achieved through the utilisation of pipelines. UNIX shell Languages are string-oriented, hence provide a uniform representation for components and values, allowing straightforward interchange of data. All filter programs in UNIX shells read a stream of bytes from an input stream and write a stream of bytes to an output stream. This allows any two programs to be connected using a pipe. Such a piping process can be flexibly utilised by users for purposes that the program designers may not even have anticipated.

A variety of mainstream modern programming languages can be used for building Web applications. Several frameworks, such as JavaServer Faces (JSF), are built to facilitate the development process and provide out-of-the-box libraries with all

commonly needed functions. Other more specific frameworks are also available for certain types of solutions like Oracle's ATG or Microsoft Commerce Server for building e-commerce applications.

Deciding on which language or framework to use is an important question. The best way to answer this question is to decide on the available resources and the magnitude of the required application. Using scripting languages would usually require fewer resources in terms of cost, number of developers and time; however, for complicated applications, with a huge number of transactions and performance requirements, high-level programming languages are definitely more suitable.

Scripting Languages are preferred when production speed is important and programs are subject to frequent changes. Ousterhout notes that, in a scripting language, a single statement executes more basic machine instructions by comparison to a system programming language (Ousterhout 1998). However, the increase in CPU speed and complexity of modern Web applications are making scripting languages more and more important. Scripting languages allow complex tasks to be performed in fewer instruction lines (i.e. less code) compared to high-level programming languages, hence scripting is considered ideal for creating mash-ups or, in other terms, gluing Web applications together.

"If recording and replaying macros simulates a kind of autopilot, then scripting offers a kind of proxy for human decision-making" (Loui 2008). Even when Macros recording is used, as the case in Microsoft Office, the macro is still recorded as a Visual Basic (VB) script. Thus, if end users are familiar with writing and customizing VB scripts, this would give them more flexibility to effectively achieve their goals.

2.2.4 Web Mining

Web data mining focuses on three aspects: Web structure mining, Web content mining (Shian-Hua, Chi-Sheng et al. 1998) and Web usage mining (Cooley, Mobasher et al. 1997, Zaiane, Man et al. 1998, Srivastava, Desikan et al. 2005). From its very beginning, the potential of extracting valuable knowledge from the Web has been quite evident. Interest in Web mining has grown rapidly, both in the research and practitioner communities (Srivastava, Desikan et al. 2005).

While the potentials of Web mining techniques are quite evident, researchers and developers are usually the ones who utilise those techniques. End users often have no direct access to such techniques and often only experience its power through search engines or other application that may make use of them. Hence, implementing some Web mining techniques and making them available for the end-user, would involve the end-user community and utilise its collective experience for extending those tools. This enables a collaborative development by end users to utilise powerful techniques through an easy to use system, which could even present further potentials for the Web data mining research community.

2.2.5 Semantic Web

The Semantic Web was defined by (Berners-Lee and Miller 2002b) as “an extension of the current Web in which information is given well-defined meaning enabling computers and people to work in better cooperation”. Since (Berners-Lee 1998) wrote his Semantic Web road map in 1998, researchers have been working on defining standards and technologies for the Semantic Web.

The main goal of the Semantic Web is to enable machines to interpret, manage, and integrate knowledge all over the Web in order to help humans to accomplish their tasks in less time and with better quality. The problem is that the Web has enormous amounts of information. In addition, it has a decentralised repository of data that changes and grows each day. These problems make it very hard to query, find, access and manage data on the Web. However, it must be noted that the free and decentralised nature of the Web enabled its great success. Therefore, solutions that enforce limitations on this freedom or attempt any kind of centralised control on the Web might not be successful.

The current Web is not easy to search well. Although search engines are powerful tools of information retrieval, the semi-structured nature of the Web is still a major limitation. Most search terms would return a long list of results (links to documents), where many of them might not be relevant to the user's information needs. It all depends on the user's experience in writing the best search phrase and using the advanced options of the search engine in order to get the most accurate results. However, in the case of narrowing the scope of search, some relevant results might be missed in the result list if they did not match the search terms.

Until recently, Search engines used to ignore what is called the "deep Web" or "invisible Web" according to (Bergman 2001). The notion of invisible Web refers to information buried far down on dynamically generated sites, and most standard search engines are not able to index it.

The Semantic Web architecture as proposed by W3C consists mainly of eight layers:

- Uniform Resource Identifier (URI) & Unicode;
- Extensible Markup Language (XML) & Namespaces;
- Resource Description Framework (RDF);
- RDF Schema;
- Ontology language;
- Logic;
- Proof;
- Trust.

These layers of the Semantic Web start from solid layers such as URI, Unicode, XML and RDF, other layers like RDFS and ontology language layers are being developed at a fast pace. Logic, proof, and trust layers are more abstract and need additional research efforts. Digital Signature and encryption are required for identification and security purposes.

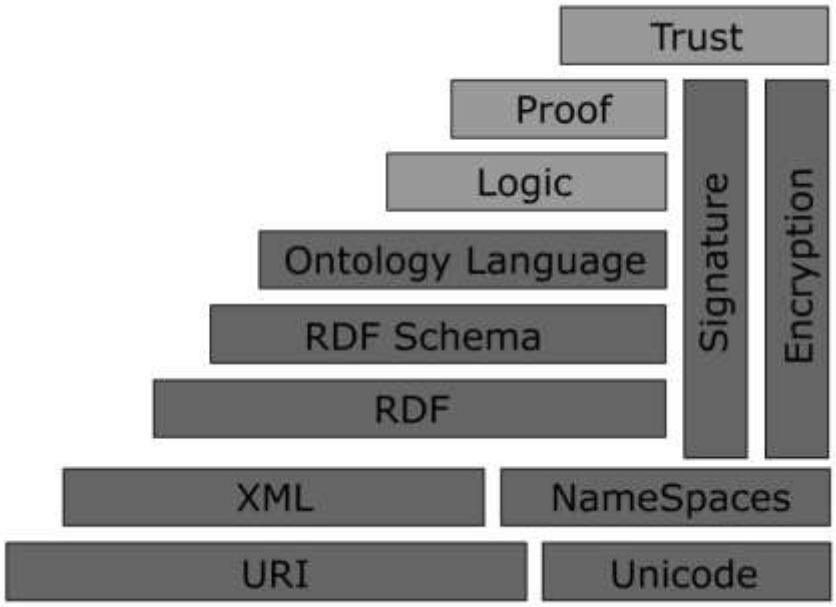

Figure 2-2: The Semantic Web layers, reproduced from (Berners-Lee 2002a)

2.2.6 Syndication Feeds

One of the major problems of the Web is the lack of proper metadata. According to (Powers 2005) Syndication Feeds, a current standard format on the Web, is a step on the way to providing Metadata for Web pages. The feeds are XML-based documents that offer a form of syndication, which facilitate the provision of summaries or latest updates of Website material to other users or applications.

There are three main common formats for Syndication Feeds on the Web:

- **RDF Site Summary** (**RSS 1.0**) (Beged-Dov, Brickley et al. 2000) was introduced by RSS-DEV Working Group in 2000, It conforms to the W3C's RDF Specification (Manola and Miller 2004).
- **Really Simple Syndication** (**RSS 2.0**) (Winer 2003) was released by Harvard Law School in 2003. It evolved from Netscape's RSS 0.90 and 0.91.
- **Atom** (Hammersley 2005) is an XML-based document format that describes lists of related information known as "feeds".

An RSS file is an XML based document describing a "channel" consisting of URL-retrievable items. Each item consists of a title, link, and brief description. Generally, RSS design has the following goals:

- Lightweight and Extensible: RSS is simply an XML document.
- Syndication and Multipurpose: it meets many growing application needs.
- Metadata: it provides clear structure and information about the presented data.

Syndication Feeds are one of the main enabling technologies for projects like Yahoo Pipes. Various Web sites, services, and applications utilise Syndication Feeds. At this moment, they are mainly most popular in Weblog (blogs) and news Web sites. This is due to the nature of Syndication Feeds, which offer an optimum way to represent lists of items in reverse chronological order. However, not all applications provide support for all feed types. For example, Blogger, a very popular hosted Web-logging tool, supports Atom and RSS 1.0 but not RSS 2.0. Google uses Atom within the Google Mail Application. RSS 2.0 is currently very popular and widely used by major news Web sites that provide an RSS 2.0 feed like the BBC and CNN.

2.2.7 Web 2.0

The Web 2.0 is a current trend that aims to utilise emerging Web technologies and design principles to facilitate information sharing and provide more creativity and collaboration among end users. It was defined in (O'Reilly 2006a) as “Business revolution in the computer industry caused by the move to the Internet as platform, and an attempt to understand the rules for success on that new platform. Chief among those rules is this: Build applications that harness network effects to get better the more that people use them”

However, the term Web 2.0 is still controversial, after the emergence of the term, some people started calling the older Web, Web 1.0, which in their view includes Web sites developed before 2005 (O'Reilly 2006a). One of the main facets of Web 2.0 is giving the users the ability to create their own content and share it. Some considered that Web 1.0 was focused more on linking pages and computers while Web 2.0 is more about linking people. Hence, most Web 2.0 applications are focused

on helping users. For many Web 2.0 applications, the more people use those applications the more influential and useful they get.

(Ankolekar, Krötzsch et al. 2008) advocate that both the Semantic Web and Web 2.0 communities can work jointly to realise their vision of the Web. While often presented as competing visions for the future of the Web, there is growing realisation that the two concepts complement each other and that in fact both communities need elements from the other's technologies to overcome their own limitations.

It is important in the context of this research to highlight the key phrase "*Web as a platform*." This refers to the notion of a platform that is not just a repository of pages and hyperlinks, but rather a whole platform where users can achieve better utilisation of available resources. The evolution of hardware, network speeds, and Web related technologies supports this vision.

Tim Berners-Lee has the opinion that Web 2.0 is actually only trying to realise what Web 1.0 was meant to be. When he was asked, in a podcast interview (Anderson 2007), if it was fair to say that Web 2.0 was about connecting people while the original Web was more about connecting computers, he answered that this was not a true assumption, and that Web 1.0 was supposed to be all about connecting people.

In the majority of older Web applications, end users were mainly consumers of information resources and services that were offered through various Web sites. However, even before the rise of the Web 2.0 term, some sites did make good use of user feedback and user generated contents, most notably Amazon.com. Now the most popular applications are those that enable end users to share their own content like YouTube, Flickr, and Wikis. In addition, Web 2.0 is actually mostly using the same standards implemented by the older Web such as HTTP.

The whole controversy about the Web 2.0 term is not the issue in the context of this thesis, except in so far as it shows that connecting people is the focus of the Web architecture and technologies. In addition, it is the objectives that the Web community tries to achieve that matter, which according to (Berners-Lee, Hall et al. 2006, Hinchcliffe 2006) is to make a firm shift towards allowing Web users to more effectively utilise Web resources in order to better employ the Web as a medium to publish, communicate, and socialise.

This leads to the importance of enabling more efficient reuse of data on the Web, which requires a significant increase in the availability of information and services. This is usually achieved by providing APIs (Application Programming Interfaces) or Web services that enable users, primarily developers, to access and reuse data according to their own needs.

All the major Web players are moving towards this approach, for example, applications like Flickr, YouTube, and Amazon do provide many APIs for developers to make it easier for them to reuse a wide range of items of interest they provide. A good example is Facebook Graph API[7], which enable users through basic links to access text-based information from their accounts.

The sharing of data and the integration between Web applications, which enable the reuse concept, is also useful for users. For example, if a user has an account on Blogger, YouTube, and Picasa, it is just a matter of a few clicks to share any video from YouTube on his blog or images from Picasa or Flickr.

According to (O'Reilly 2006b, Murugesan 2007) most Web applications derive their effectiveness from the inter-human connections, as more people make more use of them they become more effective. A popular example is eBay, where its current main competitive edge is the amount of information that users add to it.

2.2.8 Rich Internet Applications (RIA)

Rich-Internet Applications (RIAs) are applications that employ techniques like AJAX, Adobe Flash, Flex, Java, and Silverlight to improve the user experience in browser based applications. Such techniques enable building Web applications that can match the power of desktop applications. RIAs enable the replication of many features of desktop-based applications (Noda and Helwig 2005).

RIAs combine the Web's lightweight distribution with desktop-like interfaces and interactivity. With RIA support, Modern Web applications enable sophisticated end-user interactions. The “Web as platform” vision is strengthened by the emerging Web technologies.

[7] https://developers.facebook.com/docs/reference/api/

According to (Piero 2010), the term RIA refers to a diverse family of applications, which have a common goal of adding new capabilities to the conventional hypertext-based Web.

A significant characteristic of RIA is that both the client and server can start a communication. The client stands ready to receive and execute asynchronous server commands. This is enabled by the Asynchronous JavaScript and XML (AJAX) technology, which allows such communication to happen without refreshing the Web page on the client browser. This technology represents a fundamental shift in the Web applications development. Google Maps and Google search keyword suggestion are good examples of implementing AJAX.

Other Web browser-based approaches, such as Mozilla XML User Interface Language (XUL)[8], support a rich interaction natively. However, this type of solution is browser dependent. This usually makes such browser-based approaches accessible only to a specific browser's users.

2.2.9 User-Generated Content (UGC)

User-generated content refers to various kinds of information and media content, which are usually publicly available, that end users produce. According to (Marianna, David et al. 2008) "In particular social networking and community Web sites have changed the way people use new media, in creating personal profiles, sharing photos, videos, blogs and user generated content (UGC) in general." The most visited UGC sites in 2010 are Facebook, Wikipedia, YouTube and Twitter (Alexa 2012).

The User-generated content is changing Web applications towards a shift in design, focusing on more user-centric approach. Further research and development, mainly regarding UGC applications design and evaluation is a key factor that such an approach requires to reach its full potential. Several research topics are of relevance regarding UGC by non-professional users. Thousands of personal status, blogs, images, vlogs, or accounts on sharing sites are created every day even though a lot of them may have a very short time-span (Meeyoung, Haewoon et al. 2007, Marianna, David et al. 2008).

[8] http://www.mozilla.org/projects/xul

In the context of this thesis, such success and promising potentials of the User-generated content in multiple domains, suggests that it is useful and desirable to investigate a similar approach towards enabling users to share their Web experiences in form of automation tools that can save time for both themselves and the end users communities

2.3 Web Automation Tools

The Web is a dynamic information environment, which is becoming a platform on its own. Web content changes rapidly and regularly. People revisit same Web pages frequently to execute repetitive tasks. However, the tools used to access the Web, including browsers and search engines, hardly offer any support for end-user automation. Some systems aim to solve various parts of the Web automation problem, such as Yahoo Pipes, YubNub[9], and Chickenfoot[10]. This section summarises these systems and other older approaches that have addressed a range of Web automation tasks.

Most of the investigated systems are focused on certain tasks, like building customised portals (Atsushi and Yoshiyuki 1998), building customised Web crawlers as in Web Sphinx (Miller and Bharat 1998), or recording macros, as in LiveAgent (Krulwich 1997), which records a sequence of browsing actions as a macro to automate repetitive browsing tasks. Such a focused approach does not provide the level of automation that is available on other platforms using scripting languages such as UNIX.

Another approach envisages the use of external scripting languages, such as Perl or Python, to build site-specific programs to achieve a certain task. This approach is very powerful for Web developers; however, it is not as accessible or easy to use to most end users.

The idea of having a command line for controlling the Web emerged in various ways. Some may consider the browser address bar as a command line, given that users can write a complicated URL to execute queries. In many Web applications, users can even write JavaScript Commands in the address bar like:

[9] http://yubnub.org/
[10] http://groups.csail.mit.edu/uid/chickenfoot/index.php

```
Javascript: alert ("hello");
```

This would display an alert window with the message “hello”. This approach is very limited and complicated as users are always limited by the size of the address bar and the fact they need to write all their code on one line.

A more concrete implementation of a client side Web command-line which was integrated in a Web browser to create a browser shell was presented in LAPIS (Miller and Myers 2000, Miller 2002), this makes use of a pattern language called text Constraints (Miller and Myers 1999) utilised for matching text and HTML tags and enabling data extraction from Web pages.

In LAPIS, an embedded scripting language for automating Web browsing actions is utilised where the browser address bar accepts commands as well as URLs. Users can create their own scripts and call them from the browser command-line (address bar). The output of a command is displayed on the browser window, and a command can take its input from the browser window. Executed commands are included in the browser history, which can be saved as a script for later use.

Other tools were also developed by research teams, e.g. Chickenfoot (Bolin 2005a), which was developed as a Firefox plugin, while cURL was implemented as a desktop-based command line utility for executing a variety of functions with URL-oriented protocols.

Chickenfoot is built as a Firefox extension. It extends the browser’s built-in JavaScript language to enable users to achieve some level of Web automation. The Chickenfoot approach has the advantage that it does not worry about how data is retrieved from a Web page. In addition, it avoids the hassle of some Web sites responding in a different way to Web agents than they do to a Web browser. Yet, this approach is browser dependent and it is limited by the capabilities of the JavaScript language.

Web2Sh uses JavaScript for part of its commands, and if used on Firefox browsers it can make use of Chickenfoot among other plugins. In addition, many of the commands provided can be executed on the server, where one can build sophisticated commands using the full power of any language of choice such as Java, C# or PHP.

The idea of using a command line to call services offered by Web applications was presented by Jonathan Aquino in his application YubNub (Aquino 2005), which offers a helpful way to create shortcut commands to Web applications. The YubNub service enables the redirection to any published Web resource. Developing commands is very simple in YubNub; all that the user needs to do is to choose a unique command name and write the URL for the command. For example, one of the most famous commands is "`g`" that redirects the user to this URL:

```
http://www.google.com/search?q=%s
```

The "`%s`" is used to specify that users can add input parameters to the command, so the command "`g Nottingham`" will generate the URL:

```
http://www.google.com/search?q=Nottingham
```

Another approach for automating data extraction is Web crawlers. A crawler is a program that follows hyperlinks to locate HTML pages and extract the required information. Most search engines use crawlers to collect Web pages to index them, but even the development of a simple Web crawler requires a serious programming effort.

An attempt to create a framework for personal site-specific Web crawlers was tried in SPHINX (Miller and Bharat 1998), which provides a Java-based interactive development environment and toolkit for creating customised site-specific Web crawlers. SPHINX supported multi-threaded page retrieval and crawl visualisation. It also introduced the idea of classifiers, which encapsulate site-specific crawling rules to be reused in content analysis. The crawler workbench GUI available in SPHINX supports developing, running, and visualising crawlers in a Web browser and enables normal users to create their own personalised crawlers, often without the need to learn an advanced programming language. SPHINX offered powerful tools and a GUI interface for experienced end users to create their customised crawlers. However, to extend or fully utilise it users need good programming experience.

Mercator (Heydon and Najork 1999) is a scalable and extensible Web crawler written entirely in Java. One of the main targets of the Mercator design was to enable the user to scale the system to the whole Web, while SPHINX was targeting site-specific crawling.

The C3W system (Jun, Aran et al. 2004) follows another approach for facilitating composite search tasks. It attempts to reduce users' burden by allowing them to create their own custom interfaces through clipping, connecting, and cloning elements from existing Web applications.

The Internet Scrapbook (Atsushi and Yoshiyuki 1998) on the other hand, targets users with little programming skill and provides them with the facility of automating recurring Browsing tasks. Internet Scrapbook is an information personalization system that enables users to create customised personal pages by extracting only the necessary parts they need from multiple Web pages (recently referred to as a Mashup). Once the personal page is created, the system updates it automatically, as with each call it will retrieve its data components from the original sources. The system is based on the idea that a Web page's main structure does not change much, i.e. headings and positions of articles are rarely changed.

The Rainbow project (Svatek, Kosek et al. 2003) targets the Web site analysis domain. The main aim of Rainbow was to develop a reusable, modular architecture for Web site analysis. It provides resource classification, using an ontology to classify Web resources. It also offers an extraction service, which extracts information from Web pages through conversion of HTML unstructured files into XML or structured database records. Yet the system was only used for small business Web sites, and it did not have high precision results.

Well-known companies such as Yahoo, Google, and Microsoft started addressing this domain. They offered Web-based tools that can make it easier for end users to automate various tasks on the Web. In 2007, Mashup creation tools became very popular. These started with Yahoo Pipes, later on Google introduced its Mashup Editor for experienced developers, and Microsoft joined the fray by introducing a Silverlight-supported Mashup visual creator named Popfly. However, Google shut down its Mashup Editor later in 2009 and migrated to Google App Engine. Similarly, in August 2009 Microsoft discontinued support for its popular Mashup creation application Popfly.

Yahoo Pipes is an example of a Mashup composition tool that was made available to end users in 2007. In Yahoo Pipes, services can be 'piped' together to create a data

composition Mashup. Yahoo Pipes utilise RSS feeds as their main source of data, using a user-friendly GUI interface. Yet, while the GUI interface allows more novice users to utilise the system, it has a negative effect on the speed of building new tools and limits the potential of the system (O'Reilly 2007b). An evaluation study of a Command-line Parser-based Order Entry Pathway for the Department of Veterans' Affairs Electronic Patient Record showed that the command-line interface was easier to learn, and faster to use, than the usual menu-driven system (Lovis, Chapko et al. 2001).

Each of the tools had certain advantages, yet no fair comparison can be made due to the discontinuation of Microsoft Popfly and Google Mashup Editor while still in their beta versions. However, general comments about the use and approach of each system may prove useful for the design and development of a new Web automation framework.

Microsoft Popfly, being based on the Silverlight technology, enabled very impressive visual effects but it took a long time to produce automation tools when compared to Yahoo Pipes. Another advantage was offering its users a quota of disk space. However, it did not offer a reusable way to output its results compared to Yahoo Pipes, which offers its output in the form of an RSS feed. Microsoft eventually announced that the project was closed for economic reasons (Bishop 2009).

Finally, Google Mashup Editor was more suitable for experienced developers and cannot be categorised as an end-user environment. It was also discontinued in its early stages when it was only available to test through invitation requests to Google. They subsequently announced that experience gained from the project was migrated to its new App Engine project.

Some tools use JavaScript to write customised automation commands. JavaScript is well known to the majority of Web developers, yet this is not the case for many users. It also requires some knowledge of Object Oriented programming and thus cannot be the primary choice for systems that target the end-user. For example, the grammar for extracting Web data from the HTML source would be more complex to write in JavaScript than in the WSh scripting language.

2.3.1 Web SPHINX

Web SPHINX (Site-oriented Processors for HTML INformation eXtraction) is a Java-based toolkit and interactive development environment for building Web crawlers. SPHINX explicitly supports personal crawlers that are site-specific and re-locatable (Miller and Bharat 1998). It has a library of Java classes built for Web crawling, which includes HTML parsing and pattern matching support.

The SPHINX framework supports multithreaded page retrieval, it provides a Java applet called the crawler Workbench that allowed users to create and invoke simple personal crawlers without the need for programming knowledge. Re-locatable crawlers in the SPHINX system are crawlers that are capable of executing on a remote host or on the client machine. The main target is to use a remote server with higher processing power to overcome possible limitations of the client machine. In addition, a site-specific crawler would better utilise these site resources, especially if it was allowed to execute on the site server.

SPHINX represents the Web as a directed graph of pages and links, which are reflected in Java as `Page` and `Link` objects. To write a crawler in SPHINX, the programmer needs to extend the "`Crawler`" class by overriding the two methods:

- `boolean shouldVisit (Link l)`, which decides whether `Link 1` should be visited or not.
- `void visit (Page p)`, which processes a `Page p` that is visited by the crawler.

The implementation of the `Crawler` class used multiple threads to retrieve pages and links. The crawler contains a queue of links, which the `shouldVisit` method has approved but not retrieved yet. A crawling thread retrieves the page and passes it for further processing according to the user's needs. At the same time, the list of links is retrieved from this page and passed to the `shouldVisit` method. If a link was approved it is added to the queue.

SPHINX provides a facility for encapsulating knowledge needed for classifying Web content in reusable objects called classifiers. It represents a set of rules for

interpreting Web pages. For example rules that can help the crawler to choose which links on the site to follow and which to avoid. Moreover, there are rules that can specify which parts of pages to process and which to ignore. The building of site-specific crawling commands would be a significant part of a Web commands library.

SPHINX is a useful tool for advanced users and programmers to build customised Web crawlers. However, it requires programming knowledge to install, run, and extend. The program runs on the client computer making it limited to the client machine resources. Although, it has support for executing on a remote host, this is hard to configure and use. The system was last updated in 2002.

2.3.2 YubNub

YubNub (Aquino 2005) was introduced as a social command line for the Web. It was developed by Jonathan Aquino to participate in the 2005 "Rails Day" 24-hour programming contest. All the system itself does is simple redirection. It reads the command and redirects the user to the mapped URL with the ability to pass parameters in the URL using the HTTP `get` method; later on one of the YubNub users added `post` method support via a PHP script that converts from `get` to `post`.

Developing commands is very simple in YubNub, all the user needs to do is choose a unique command name and write the URL for the command, for example the command `yahoo` simply generates the URL:

```
http://www.yahoo.com
```

Another example, the command `gmap Nottingham` displays a Nottingham map using Google maps, and returns the URL:

```
http://maps.google.com/maps?oi=map&q=nottingham
```

Advanced Web users can create their own commands by writing a Web dynamic page in any available language like PHP, then adding a YubNub shortcut to it, e.g. the command: `strLeft Nottingham 3` It will return the URL:

```
http://fromrocks.com/yubnub/strings/stringmanip.php?
command=strleft&input=abcdefg%203
```

This URL will execute the code on `stringmanip.php` and passes the parameters `command=strleft&input=Nottingham 3` in the URL using the `get` method. The result will be the first three characters of the sent string, the browser will display "Not."

Another example is the command `back`, which was developed using JavaScript to act as if the users pressed the back button; the command corresponds to the following URL:

```
http://www.cs.nott.ac.uk/~eoe/back.html?n=-1
```

The user may supply the number of pages to jump back to as a parameter; if no parameter was supplied the default value will be the value 1.

YubNub provides an online way to save shortcuts to various Web sites, however on the scale of providing a generalised Command Line for the Web it still has long way to go. The basic problems of YubNub are as follows:

- There is no standard for creating commands;
- It does not offer a method for users to create private commands. This is very important for users who want to create commands that include their personal login information to a certain site;
- It does not support secure transactions for the users to send login information in a command;
- Commands may be easily broken. If the user removes a command page from the server, this will cause it to return the infamous "404 page not found" error.

2.3.3 Mashups

A Mashup is a Web application that integrates information from more than one source into a single tool to create a different Web service that was not originally intended by either of the original sources.

The very first example of a Mashup was “Craigslist + Google Maps” by Paul Rademacher's HousingMaps. It combines information about real estate from Craigslist with Google Maps to provide a distinct service.

Many other interesting Mashups followed like chicagocrime.org which from was one of the original map Mashups. It combined crime data from Chicago Police Department with Google Maps offering a page and RSS feed for every city block in Chicago (Holovaty 2008).

Creating similar Mashups is becoming easier, ever since Google published its Google maps API for public use. Many major services providers are providing various APIs for their applications, thus making the process of creating Mashups easier.

Marmite, a tool developed by (Jeffrey 2007), provides an end-user programming tool that allows end users to create Mashups. It was implemented as a Firefox plugin using JavaScript and the XML User Interface Language (XUL). Marmite enables end users to create mashups that repurpose and combine existing Web content and services in an approach similar to Spreadsheet scripting. It presents *Operators;* these are pieces of code that access or operate on data. Operators can be chained together in a data flow where the results of one operation are passed as input to the next.

2.3.4 Yahoo Pipes

Yahoo Pipes is a service that provides an easy way to aggregate and mashup Web content. Similar to UNIX pipes, modules can be combined together to create the desired output. Yahoo launched this service in early 2007. Since then many users have been exploring it and creating pipes with various functionality.

Tim O'Reilly considers Yahoo Pipes as a milestone in the history of the Internet (O'Reilly 2007b), He described Yahoo Pipes as follows: “While it's still a bit rough around the edges, it has enormous promise in turning the Web into a programmable environment for everyone”. Although most modules are still targeted to RSS feeds, it seems that eventually Yahoo Pipes aims at creating a GUI based programmable environment. Its focus is to allow users to aggregate data and build Mashups.

Jon Udell talked about a similar concept in the 8th International Python Conference, He talked about the Web as a *library of callable components* (Udell 2000) and the possibility of combining Web services as the Web analogue to UNIX pipes. In 1997, he had a vision of Web sites as “Data sources that could be re-used, and of a new programming paradigm that takes the whole Internet as its platform” (O'Reilly 2007b).

Yahoo Pipes provides a simple interface. It provides a drag and drop editor that graphically allows users to connect Internet data sources (mostly Web feeds). It allows its users to do many things like aggregating and filtering multiple news feeds. Many widely used Web sites now offer search results as a feed. Examples include Google News[11], Technorati[12], and Delicious[13]. Users can combine searches on various items into a single feed.

When it was first launched, Yahoo described it as “an interactive feed aggregator and manipulator” later on they describe it as “A powerful composition tool to aggregate, manipulate, and Mashup content from around the Web.” That was due to the addition of new modules like fetch page and many site-specific modules under the sources category.

[11] http://news.google.com/
[12] http://technorati.com/
[13] http://www.delicious.com/

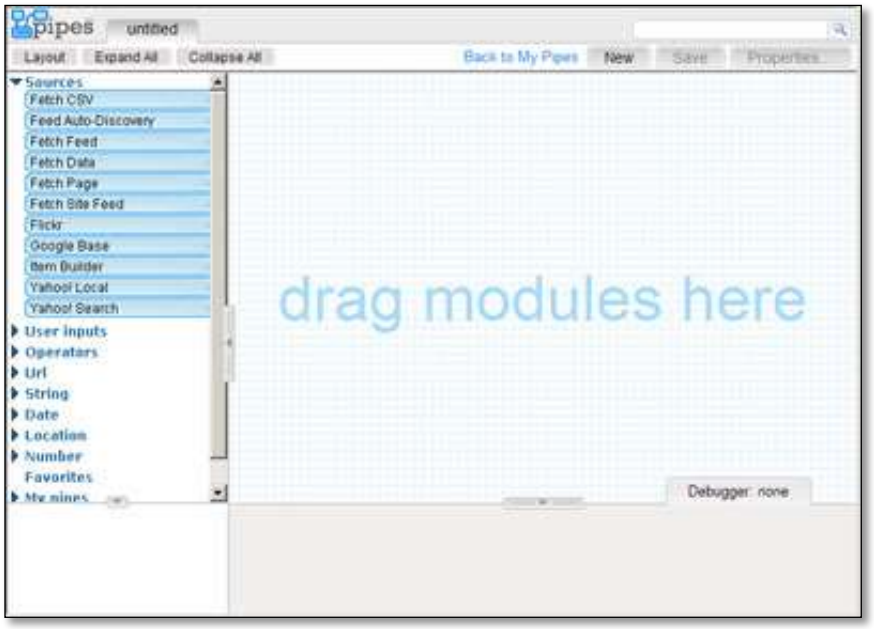

Figure 2-3: Yahoo Pipes Editor

Other sites that do not provide any feeds can be parsed to extract the required information to be presented as an XML syndicate feed. The Feed43 engine offers this service; it converts free-form HTML or XML documents to valid RSS feeds by extracting snippets of text or HTML by means of applying search patterns. End users need basic knowledge of regular expressions to use similar services.

The YQL module facilitates the use of the Yahoo Query Language[14], which is a SQL-like language that lets the user query, filter, and joins data across Web services. This enables Yahoo pipes to retrieve data from various APIs through the utilisation of open data tables[15], which are XML files that describe how the YQL language can be mapped onto a Web service or internet data source. (Bozzon, Brambilla et al. 2010). For example, to return top 10 images from Flickr related to Egypt, the following YQL query can be used:

```
select * from flickr.photos.search where text="egypt"
and api_key="api_key_value" limit 10
```

[14] http://developer.yahoo.com/yql/
[15] http://www.datatables.org/

2.3.5 Microsoft Popfly

Microsoft describes Popfly as a "fun and easy way to build and share mash-ups, games, gadgets, and Web pages." PC World magazine picked Popfly as one of the most innovative computing and consumer electronics products of 2007, the tool was rated as 21 in the 25 most innovative products of the year list (PC-World-Staff 2008). Popfly was powered by Microsoft Silverlight technology and aims to provide an easy to use environment for an end-user to create a variety of Web-based applications and mash-ups.

Popfly utilises Silverlight to achieve superior optical effects. It had modules that could show the data from various sources with nice visual effects (e.g., Microsoft Virtual Earth).

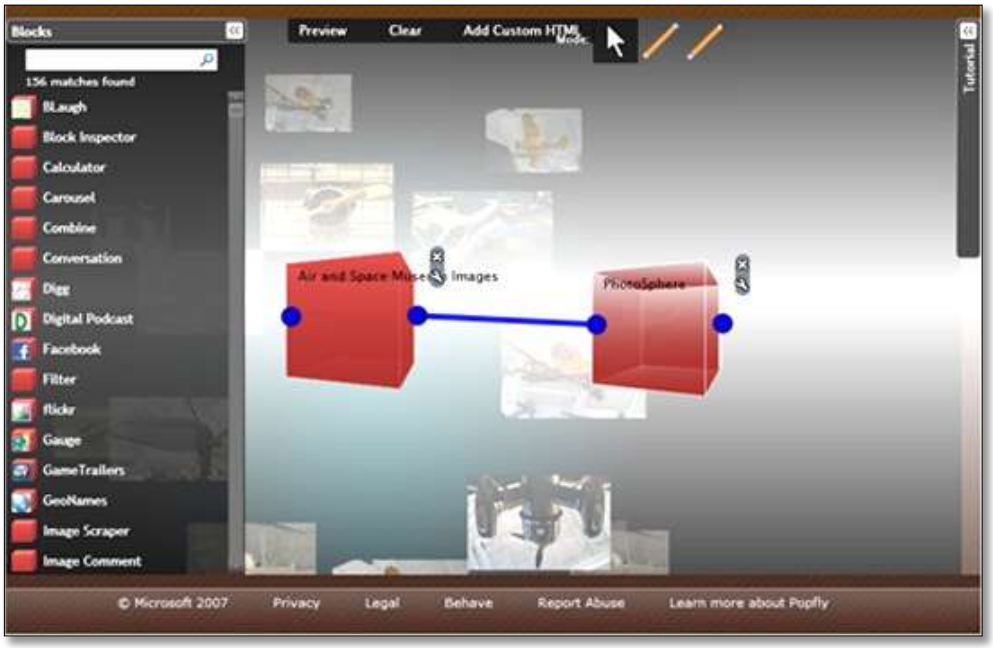

Figure 2-4: Microsoft Popfly

In 2009, Microsoft discontinued Popfly taking down all sites, references, and resources that users had created. Popfly was visually impressive; however, it could hardly be used to create useful Mashups. Popfly was mostly used as a fun tool to create games or entertaining applications.

End users did not require any programming skills to use Popfly, just a good understanding of data formats. Figure 2-4 shows a screen shot of a Popfly interface.

Tim O'Reilly criticised Popfly in (Markoff 2008) saying that "Popfly shows me that Microsoft still thinks this is all about software, rather than about accumulating data via network effects, which to me is the core of Web 2.0, They are using Popfly to push Silverlight, rather than really trying to get into the Mashup game."

2.3.6 Firefox plugins

Firefox offers a plugin add-on approach, with various tools for the automation of customised Web browser-based tasks. Some plugins are targeted toward a specific task such as *DownThemAll* that allows its users to download all the links or images contained in a Web page. Other plugins allow for a wider range of task automation such as iMacros, Greasemonkey, Accessmonkey, and Chickenfoot.

Greasemonkey is a Firefox extension which allows users to customise and manipulate how Web pages are displayed on Firefox, by using pieces of JavaScript (Pilgrim 2005). It allows Web users to write scripts that alter the Web pages they visit. Advanced users of Greasemonkey can import, combine, and modify data from a set of Web sites to create Mashups that meet their specific needs. However, writing such scripts is very complicated and may take a long time to master.

Accessmonkey is an extension to Greasemonkey aimed for collaborative Web accessibility improvement; It provides a common platform on which end users can create, share and use scripts that specify accessibility improvements (Jeffrey 2007) and also uses JavaScript to manipulate Web page content.

Chickenfoot is easier to use compared to the above-mentioned plugins. It adds a programming environment in the Firefox sidebar so the user may develop scripts to manipulate Web pages and automate Web browsing (Bolin, Webber et al. 2005b). However, scripts are written in a superset of JavaScript called "Chickenscratch" which includes commands that are suitable for operating on the rendered model of a Web page. Writing advanced scripts in Chickenscratch would require good programming knowledge.

iMacros[16] imitates the macro recording in Spreadsheets to automate any repetitious task on the Firefox Web browser; one of its main uses is automated form filling.

[16] http://www.iopus.com/imacros/firefox/

Such an approach can be useful for simple tasks but is tedious, hard to update and tightly coupled to the specific Web browser it works with.

Writing scripts that fills out Web forms is a common goal for various plugins, yet the complexity of most of these plugins, and the fact that they are tightly coupled with a specific Web browser, plus the lack of supporting specialised easy-to-use scripting language make them suitable primarily for experts.

Ubiquity (Beard 2008) is another plugin for the Mozilla Firefox browser. It is an experiment by Mozilla labs, not developed by a third party. The original idea started with the Enso[17] project, a cross-platform, graphical, and linguistic based command-line interface written in Python.

Mozilla Labs, later on, applied the same approach within the Firefox browser as an experiment aiming to connect the Web with natural language in an attempt to make it possible for everyone to do common Web tasks more quickly and easily. The current development status of Ubiquity is an indefinite hiatus. Ubiquity commands are also JavaScript based and require good programming knowledge to extend by end users. The interface is easy to use and user friendly. Ubiquity showed a great advantage using natural language-based commands that were given aliases, making it very easy to users to call commands, yet with no easy way to connect together the commands it offers. Authoring any new command requires a very good knowledge of Ubiquity itself and the JavaScript language.

Chickenfoot and Ubiquity stand out among the above plugins because they have easy to use commands that can be used for automating the filling of Web forms, among other tasks, with a simple supporting visual pane for writing and running commands. However, they also depend on the browser limiting all processing to the client side.

The browser-based plugins approach is generally useful in some cases, yet it has many limitations. A browser-based plugin would work only on a limited number of

[17] http://code.google.com/p/enso/

browsers; a plugin needs to be updated with each new browser version release. Furthermore, a browser-based plugin can be executed only on the client side with a limited set of functions.

2.4 UNIX Shell Scripting

"UNIX shell is both a command interpreter, which provides the user interface to the rich set of UNIX utilities, and a programming language, allowing these utilities to be combined. Files containing commands can be created, and become commands themselves." (Ramey 1994)

A common environment for UNIX and Linux platforms users is the shell-based console. Users often open a number of shell consoles, execute command-line instructions, and automate tasks through scripts in one of the available shell scripting languages such as C shell.

"A shell is a program that acts as the interface between you and the UNIX system, allowing you to enter commands for the operating system to execute. In that respect, it resembles DOS, but it hides the details of the kernel's operation from the user. So, file redirection just uses < and >, a pipe are represented by `|`, output from a sub-process by `$(...)`, and the implementation details are handled for you. In that respect, it's a high-level programming language for UNIX itself." (Stones, Matthew et al. 2000)

A very good advantage of UNIX is that it has a rich set of generic tools for operating on text files like: `grep, sort, awk, sed`, etc. These tools present a very good example as generic text processing tools for Web shell commands. However, the generic nature of these tools can also be a weakness, because generic tools can make only limited assumptions about the format of text files. Such generic tools are not efficient enough and would need significant programming experience and effort when dealing with the variety types of structured and semi-structured text on the Web, such as HTML pages, email messages or Source code (Miller and Myers 1999, Miller 2002).

Many Web pages use some kind of custom structure represented in HTML; for example, search engine results, shopping catalogues, and sports scores. Developing

similar tools that utilise the advantage of a large base of existing UNIX shell users, and can make better use of the semantic information available in structured text in Web pages would be very useful for Web users.

UNIX shell main features are:

- Command interpreter
- Programming language
    a. Variables
    b. Flow control (Conditions and loops)
    c. Constructs
    d. Quoting (using single and double quote for strings)
    e. Functions
    f. Environment Variables
- Shell scripting is easier to learn compared to high-level languages.
- Input and output redirection is achieved using < and > characters.
- Processes piping using |, e.g. `ps | sort > processes.txt`

  This takes the output of `ps`, sorts it into alphabetical order, and then writes the output to the `processes.txt` file.

```
ps | sort | grep -v bash | more
```

  This takes the output of `ps`, sorts it into alphabetical order, and then uses `grep -v bash` to remove the process named `bash`, and finally displays it on the screen in screen size blocks.

2.5 A Programmable Web

Nardi argues that “task specific programming languages are both useful and powerful”, showing that end users can easily learn formal languages when they are motivated and when they have programming languages which reflect their task domain, and a suitable framework to view the output of their programs. This principle can be seen in spreadsheet scripting, which require specific and complex commands to be grasped by users in order to achieve their goals (Nardi 1993).

An extensible task-specific programming language targeted to automating tasks over the Web platform needs to be able to do the following:

- Access Web resources using different protocols;
- retrieve any required Web resource;
- understand and be able to process the different data structures on Web pages;
- execute basic regular programming operations.

Early Web pages were mainly written in simple HTML code. However, nowadays rendering most Web pages involves dealing with many other languages and technologies, such as CSS, XML, Flash player and JavaScript. More advanced techniques are also used for better interactivity such as AJAX.

Achieving the required automation requires collaboration between the server-side and the client side. The server-side of a Web application needs to offer the data in way that is accessible and which other programs, when needed, can easily process. Web services are a very good example. Many major vendors offer APIs and Web services to access their data without all the fancy looks and interface burden. Other programmers within their own application are mainly the ones who use most of these services; Mashups are a very good example of this.

On the client side, normal users need enabling tools to access such services and data directly and a scripting language in which they may express any algorithm for task automation if they require automating any repetitive task.

2.6 Summary

This chapter discussed the most popular tools for Web automation. It is a novel field in which more research is still needed. Currently, the most noted tool, Yahoo Pipes, is only used by a small fraction of the Web user community. It also provides limited functionality that handles only certain formats and protocols. Yahoo defines its tool as a composition tool to aggregate, manipulate, and Mashup content from around the Web (Yahoo 2007).

The GUI-based interface of Yahoo Pipes is easy and fun to use, yet it requires high-performance devices and is much slower for developing tools compared to other text-based development environments.

Other tools implemented the approach to work over the Web browser as its platform. These tools were developed as Firefox plugins such as Chickenfoot and Ubiquity. While this approach is very powerful and useful for manipulating content acquired by the browser, it suffers many limitations. These tools are mainly focused on the page customisation and automation of the browser experience. Furthermore, they operate on the client side only, limiting access to only some of the many Web available resources.

The Web is moving towards becoming an independent platform. Furthermore, end users are moving from being passive actors to becoming an integral part of Web information growth as the user-generated content concept is pushing the Web towards a new era where Web sites adopt a more user-centric approach.

There is little documented academic literature that focuses on end-user Web automation as defined in this thesis. Most available tools are of commercial nature and target only few aspects of Web automation. This shows that there is a still a lack of tools for end users who want to automate and customise their Web interactions. The growing complexity of Web pages and standards prevents most users from realising such automation.

Chapter Three: **WEB2SH FRAMEWORK DESIGN**

3.1 Introduction

This chapter describes the design of Web2Sh. It presents the main design challenges, and describes solving the problem of developing a concrete, useful framework for end-user Web task automation. It presents in more detail the architecture, design, pipeline concept and characteristics of the framework. Finally, it discusses the implementation decisions in Web2Sh.

The move towards the Web as a platform, where the main environment for computer user interaction is the Web browser, makes it essential for such a framework to run through the Web browser as the client. However, locating the main components of the framework over the server tier is important. This saves end users the burden of having to install additional software components on their machines. It also facilitates the use of resources offered by the public cloud, which are services and resources offered and owned by providers, like Amazon and Google, who offer the users access to these resources only via the Internet and operate the infrastructure; i.e. data-center hardware and software (Armbrust, Fox et al. 2010).

It is not the purpose of this chapter to describe the fine details of the design; rather it focuses on describing the most interesting and important features of the framework.

3.2 Design Objectives and Requirements

The main objective for the design of the proposed Web2Sh framework is to investigate and evaluate the feasibility and usefulness of end-user Web task automation. Aiming for an infrastructure that achieves such a goal enforces design choices that take into consideration the issues of scalability, usability, and extensibility. The idea of developing a framework for writing, sharing, and using customised Web automation tools, exhibits considerable benefits. It helps end users to utilise powerful resources that have so far only been accessible to programmers. It also empowers end users to achieve better efficiency, to save time and effort, and to share their Web experiences.

The Web2Sh framework introduces a one-stop Web-based interface for end users, consolidating many Web automation tools in one place. Such a framework needs to offer many advantages, including a consistent usable interface with a homogeneous view of various Web resources. The implementation of the Web2Sh framework has considered the following general requirements.

Scalability, the system needs to be able to handle increasing number of users and to work efficiently under increased load.

Parallelism, the system should operate on the principle that problems can be divided into smaller ones, which can then be solved concurrently.

Extensibility, the system design must allow for future growth by supporting the ability to enhance the system via the addition of new functionality or modifying existing ones.

Usability, The system needs to be easy to learn and to use for end users, allowing them to accomplish various tasks at a satisfactory speed.

Web2Sh also aims to make use of other available tools that end users may want to integrate into the framework. For example, a wrapper was built to enable the passing of data between the UNIX `bash` shell and Web2Sh to make use of a number of its text-processing tools, such as `grep`, `sed`, and `awk`, which could be used for processing textual data. In the same way, Web2Sh can also utilise any accessible resource or tool that generates text-based output, by implementing the appropriate wrappers.

The implementation of Web2Sh as a Web application allows it to be accessible by Web users from various modern browsers. It offers a set of commands, which are usually executed on the server-side using Ajax Technology to transfer commands and results between the client and the server without the need to refresh the page.

In addition, Web2Sh also utilises the Web browser by implementing some simple Web commands on the client side. These Web commands aim mainly at providing the user with access to basic browser functionalities, such as history and bookmarks via the use of JavaScript.

3.3 The Web2Sh Architecture

Web2Sh has five main components:

7. An AJAX enabled Web interface.
8. An `Execute` Servlet to handle communication with the Web interface.
9. The WSh scripting language Parser.
10. The WSh scripting language Interpreter.
11. A repository of Web commands.

End users can use the Web interface to write Web commands. A user terminates writing a pipeline, or a single Web command, by typing the semicolon character. The client then sends the code to the Web2Sh server using AJAX to exchange data with the server without reloading the whole page. The use of a semicolon to indicate the end of a pipeline allows users to write a pipeline across multiple lines over the interface. Figure 3-1 demonstrates the main components and the flow of data through the system.

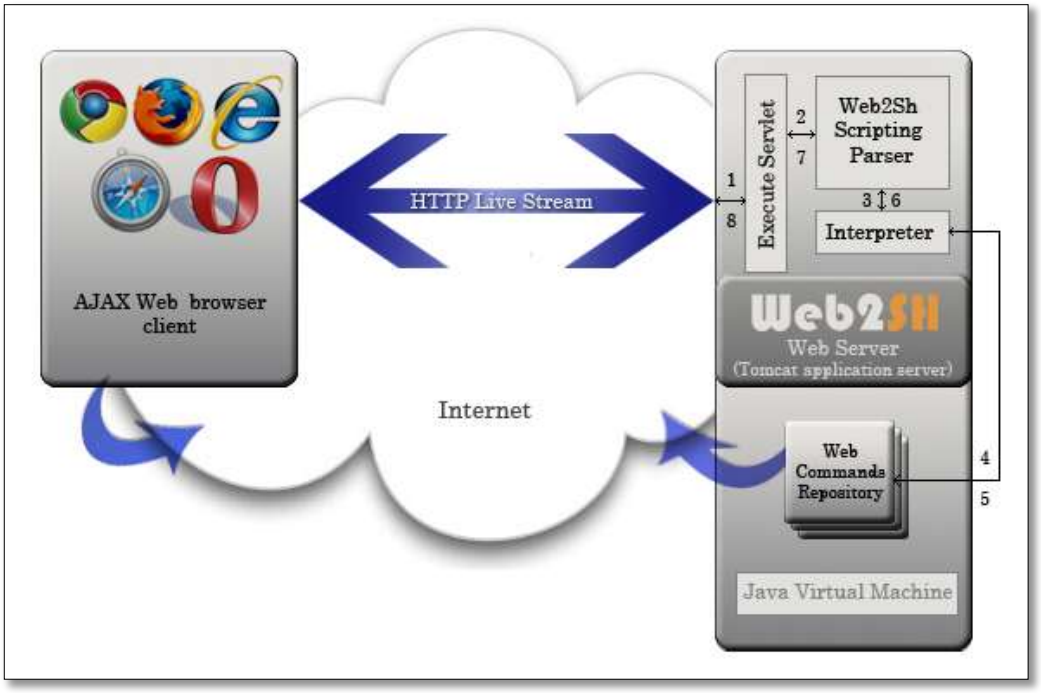

Figure 3-1: Web2Sh Architecture

1. The client interface sends the code using an `XMLHttpRequest` object; the `Execute` Servlet processes the HTTP request to extract the Web commands.

2. The `Execute` Servlet creates a Parser instance and sends it the code.

3. The Parser validates the code syntax. If it is valid, it will call the Interpreter and pass an AST (Abstract Syntax Tree) representation of the code.

4/5. These steps occur repeatedly according to the number of commands in a pipeline. The Interpreter handles the invoking of the called Web command. If a pipeline exists, then each command executes returning a reference to its standard output stream to the Interpreter, which connects it with the next command's standard input stream.

6. After the last Web command executes, the Interpreter returns the result to the Parser.

7. The Parser object passes the result back to the Execute Servlet.

8. The `Execute` Servlet writes the result over the HTTP stream and sends it to the Web browser, which processes the returned data using JavaScript and updates the page content.

The Web2Sh framework utilises the `java.net` library to enable Web commands to access the Internet to retrieve resources. The AJAX Web interface in some cases may access the Internet directly to retrieve information through the Web browser.

3.3.1 UNIX Tools Web Wrapper

Web2Sh utilises various tools provided by the UNIX `bash` shell script (Ramey 1994) through an embedded UNIX shell wrapper which directs some Web commands to be executed using the UNIX shell:

```
1   fetch "url" | txt |
2   unix.grep '[0-9] [0-9] [0-9]';
```

This command will retrieve the given URL (`fetch`), remove all HTM tags (`txt`), and then pass the text to the UNIX tool wrapper, which will call a UNIX shell to execute the `unix.grep` command.

Many UNIX tools can be accessed through the wrapper, such as `sed`, `awk`, `grep`, and `sort`.

For example:

```
1   fetch pageURL | unix.sed 's/</\&lt;/g' |
2   unix.sed 's/>/\&gt;/g'sed 's/</\&lt;/g';
```

This command fetches any URL and passes the content to `sed` tool, which replaces all '<' characters with `<` and then all '>' characters with `>` the result will then be displayed on the browser which would be a text representation of the whole HTML file. This is a simple command that can enable users to view any page source within the normal browser window.

The Web2Sh shell will search for any Web command on its list of commands, if the command is one of the `bash` commands the UNIX tools wrapper would handle passing it to the `bash` shell and return the results to Web2Sh.

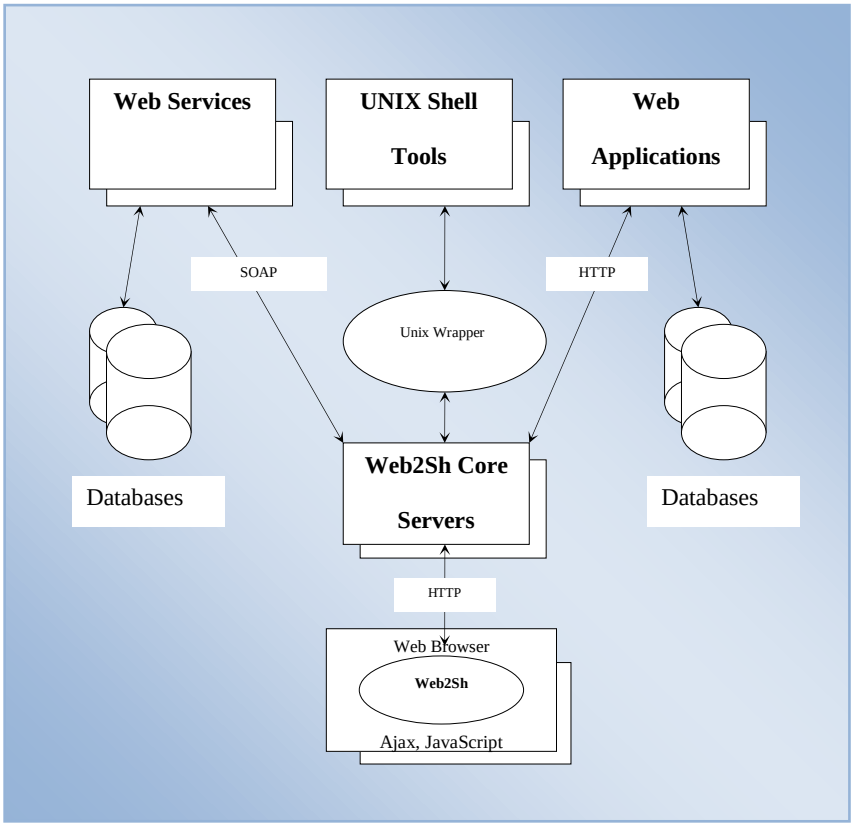

Figure 3-2: Web2Sh Interactions

Web2Sh Wrappers are written and used in an innovative way. Its aim is to offer an interface through which the end-user can utilise useful tools available in other systems and platforms. It acts as a Web command for other Web2Sh commands, yet it sends the instructions to be executed on its target platform, i.e. UNIX, and then it return the results back to the standard output stream.

3.4 Web2Sh Framework Design

Object-oriented programming (OOP) is a practical and useful programming technique that supports many features such as data abstraction, encapsulation, messaging, modularity, and inheritance. It encourages modular design and software reuse (Alan 1986, Gamma, Helm et al. 1995).

Recurring patterns exist in object-oriented programming, which provide simple, tested, and elegant solutions to specific design problems. A design pattern is a general design-solution to common design problems, which describes classes and communication between objects. Design patterns assist developers to structure their

applications, giving them a common vocabulary to describe design concepts (Gamma, Helm et al. 1995, Freeman, Freeman et al. 2004).

Christopher Alexander says, "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" (Alexander, Ishikawa et al. 1977, Gamma, Helm et al. 1995)

The use of design patterns helps to avoid subtle issues that may cause major problems. In addition, it improves code readability for programmers who are familiar with the patterns, allowing them to easily use and extend the system.

Web2Sh utilises the use of OOP and design patterns in order to speed up the development process while meeting the system’s requirements. In addition, design patterns represent tested and proven development paradigms for solving common development challenges.

The Web2Sh design utilises the following design patterns (Freeman, Freeman et al. 2004):

- The Visitor Pattern.
- The Command Pattern.
- The Abstract Factory Pattern.

These patterns interconnect together to handle interpreting, calling, instantiating, and executing the Web commands of Web2Sh.

The Visitor pattern is used to separate the Interpreter implementation from the Web command object (a node in the AST) that it uses and the parser. This facilitates introducing changes to either one of the components without the need to change or re-implement the other.

3.4.1 Web2Sh Implementation of the Visitor Pattern

The Visitor design pattern represents an operation to be performed on a node of the Abstract Syntax Tree returned from the parser. This allows the definition of new operations without changing the classes of the nodes. It also enables the extension of the Parser functionality by simply adding a new visitor class.

A Visitor pattern has the following participants:

- **Client** (`Web2ShInterpreter`)

  This Class uses only interfaces declared by `Node` and `Web2ShParserVisitor` interfaces.

- **Element Interface** (`Node`)

  This Interface defines an accept operation that takes a visitor as an argument.

- **Concrete element** (All AST nodes, such as `ASTFlag`)

  This Class implements an accept operation that takes a visitor as an argument.

- **Visitor Interface** (`Web2ShParserVisitor`)

  This Interface declares a visit operation for each class of `ConcreteElement.`

- **Concrete visitor** (`InterpreterVisitor`)

  This Class implements each operation declared by Visitor Interface.

The WSh scripting Parser receives the source code, validates it, and constructs an abstract syntax tree (AST), which represents the Web commands' structure of the source code. Each node (concrete element) in the constructed AST has a `jjtAccept` method taking a `Web2ShParserVisitor` (visitor) object for an argument to add the required functionality to the node. All nodes implement the `Node` (element) interface to guarantee the existence of `jjtAccept` method. A separate concrete visitor `InterpreterVisitor` class implements a visitor interface `Web2ShParserVisitor` to perform some particular operations, by

implementing these operations in its respective `visit` methods. The Interpreter decides on which visit method to call based on the type of the object passed to it as the first parameter.

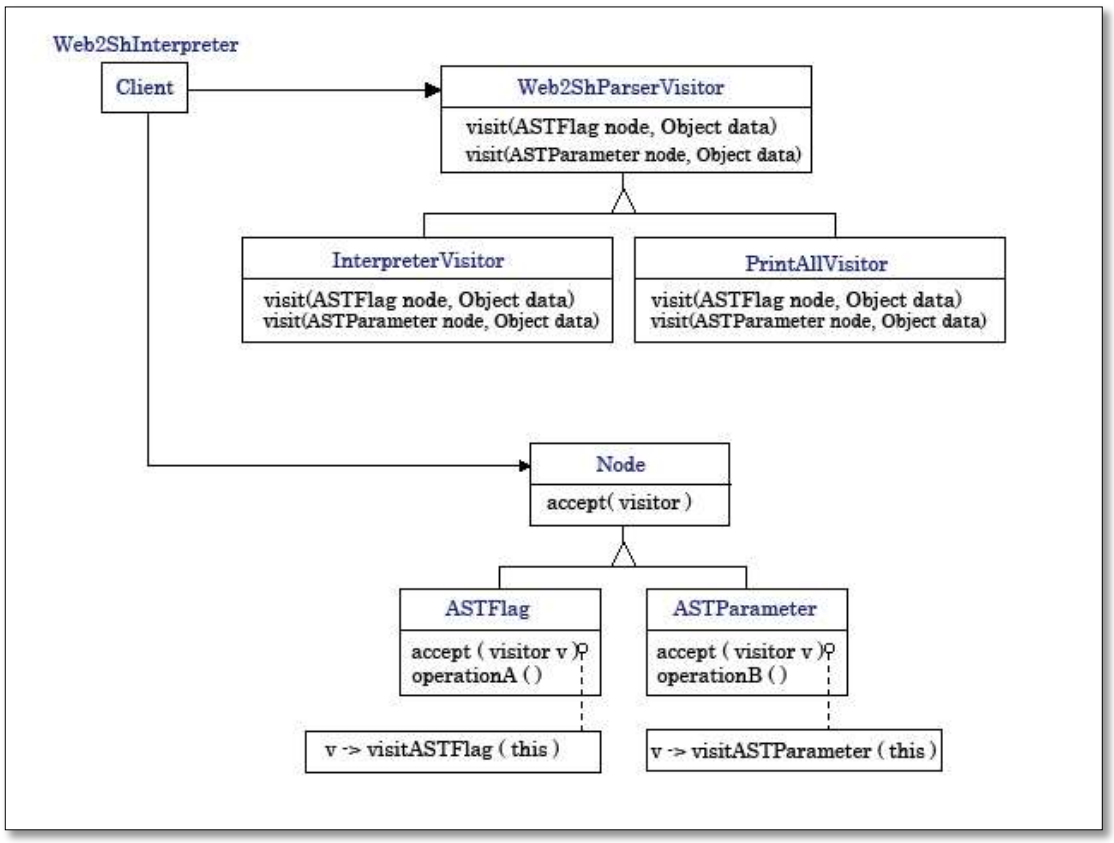

Figure 3-3: Web2Sh implementation of the Visitor design pattern

This is a `visit` method example:

```
1   public Object visit(ASTFlag node, Object
2   data) {
3     ........
4   }
```

The `data` object is used to save tracing data among visit methods calls for collecting information for debugging reasons.

3.4.2 Web2Sh Implementation of the Command Pattern

The Command design pattern encapsulates a request as an object, thereby enabling clients to issue different parameterized requests (Gamma, Helm et al. 1995, Freeman, Freeman et al. 2004). Hence, it is used in Web2Sh for Web commands. This design enables each Web command, such as the `fetch` command, to operate in a different way according to the receiver type, which contains the actual action for the command. Hence, every receiver object can execute its own implemented action.

The receiver in this case is an abstract class `WebMimeType`. This class is inherited by all classes that represent a MIME type, such as `Text` and `Image`.

Each main MIME type is a parent class that is inherited by its subtypes; for example, HTML, CSS, CSV, XML, and Plain classes inherit from the Text class. This structure enables the parent class to implement all the common actions for its child classes, while, if required, each child can implement its response action for a given command.

A Command pattern has the following participants (Figure 3-4

- **Command** (`org.web2sh.cmds.AbstractWebCommand`), this is an interface for executing an operation.
- **Concrete Command** e.g. (`org.web2sh.cmds.fetch`), and similar Web commands classes. The Concrete Command defines a binding between a `Receiver` object and an `action`, enabling the Web command to implement execute by invoking the corresponding operation(s) on the `Receiver`.
- **Client** (`CommandManager`), this creates a Concrete Command object and sets its `Receiver`.

- **Invoker** (`Web2ShInterpreter`), this asks the command to carry out the request.
- **Receiver** (`WebMimeType`), this class contains the information on how to perform the operations associated with carrying out a Web command action.

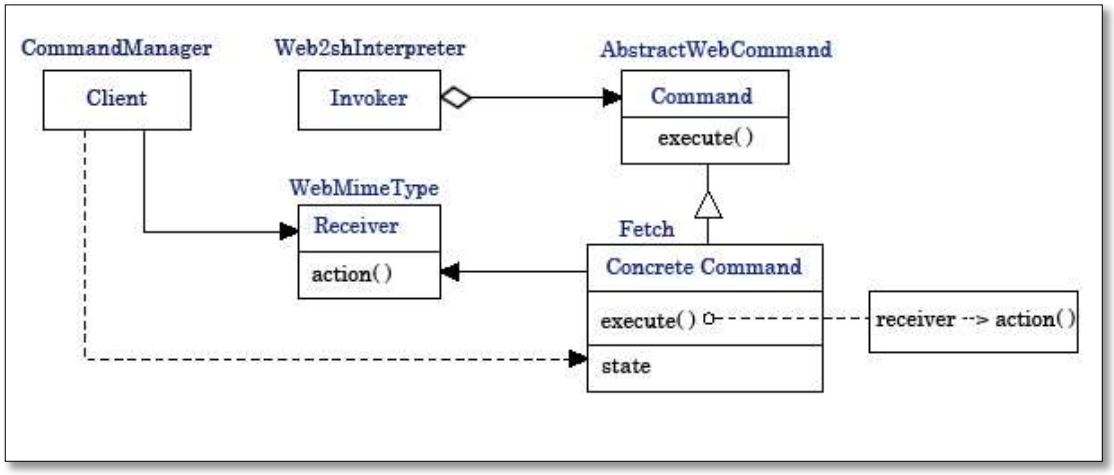

Figure 3-4 Web2Sh implementation of the Command design pattern

The implementation of the command pattern makes it easy to add new Web commands, as the developer does not have to change existing classes. This allow for extending the core Web commands repository without the need to alter the framework.

3.4.3 Web2Sh Implementation of Abstract Factory Pattern

This design pattern provides an interface for creating families of related or dependent objects (Web Resources) without specifying their concrete classes.

This is essential for Web2Sh design, as the framework in most cases does not know which type of resource it is going to deal with. This enables the Web2Sh framework to be independent of how Web Resources are created, composed, and represented. Web Resources are families of all the defined MIME types.

An Abstract Factory pattern has the following participants (Figure 3-5)

- **Abstract Factory** (`ResourceAbstractFactory`) , this is an interface for operations, which creates abstract products.
- **Concrete Factory** (`TextFactory`, `ImageFactory`), this class implements the operations to create concrete product objects.
- **Abstract Product** (`WebResource`), this interface is implemented by all Web resources, such as `Text`.
- **Concrete Product** (`Text`, `Image`), this class defines a resource object to be created by the corresponding concrete factory.
  It implements the Abstract Product interface.
- **Client** (`CommandManager`), this class uses only interfaces declared by Abstract Factory and Abstract Product interfaces.

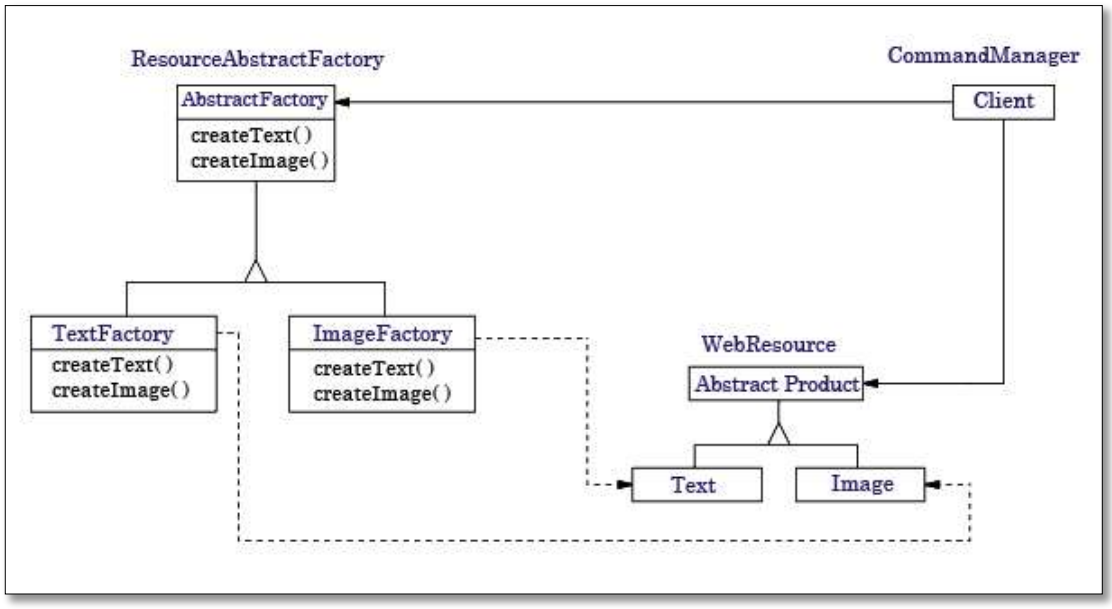

Figure 3-5: Web2Sh implementation of the Abstract Factory pattern

3.5 Framework Scalability

The literature has many definitions of the scalability term such as in (Weinstock and Goodenough 2006, Duboc 2009). It was generally defined in (Duboc 2009) as "the ability of a system to satisfy quality goals to levels that are acceptable to its stakeholders when characteristics of its application domain and design vary over expected ranges".

Scalability in the context of this research is the ability of the software to handle an increasing load in terms of data, processing and number of users. This can be achieved by employing both good design and development techniques to allow for the best use of resources. In addition, it has the capacity to use the increasing number of processors by supporting the ability to extend resources using horizontal clustering approaches. It is important for a Web application design to cater for efficient use of resources in order to offer a good level of scalability, especially concerning the number of users.

There are two methods of clustering, namely: *vertical scaling* and *horizontal scaling*. Vertical scaling involves increasing the number of servers on a single machine, while horizontal scaling involves increasing the number of machines inside the cluster. The horizontal scaling approach can be more reliable, since there are multiple machines involved in the cluster environment, as compared to only one machine in vertical scaling (Duboc 2009).

The scalability of a system depends on the characteristics of both the application design and the execution environment. A design that divides an application into units that could execute independently in parallel will allow for a superior performance when utilising the horizontal scaling approach (Williams and Smith 2004).

To ensure the scalability of Web2Sh the design utilises the Java Platform, Enterprise Edition (Java EE) and the Apache Tomcat application server as a well-known tested Web application server with clustering support.

Tomcat provides the ability to deploy a Web portal application in a cluster environment, which facilitates achieving the scalability, reliability, and high availability required for Web applications

3.6 Summary

This chapter presented the Web2Sh framework. It presented the main design challenges and objectives, the framework architecture, and key design features.

Web2Sh is a concrete, useful framework for end-user Web task automation. It offers end users the ability to build their own automation tools that target the Web as their platform. The next chapter introduces the Web2Sh user interface, where users can write and run their Web automation scripts.

Chapter Four: **WEB2SH USER INTERFACE**

4.1 Introduction

Web applications are becoming increasingly rich and flexible environments where users can easily search for information, access documents, stream media files, publish content, and play games; all from within the Web browser. Hence, any end-user Web-oriented application needs to be usable from within a browser.

This chapter provides a detailed description of the Web2Sh command line user interface, which is the client side component of the framework. One of the main influences on the design and development of the Web2Sh user interface is the need to be usable on the majority of available Web browsers and Web-enabled devices.

The interface could have been implemented as a browser plugin, e.g. for Firefox. However, such plugins work only on specific browsers and even specific versions of those browsers. It could also have been implemented as a separate client-based desktop application similar to the SPHINX environment; yet again, it would have needed to be installed on the client machine and would not actually make use of the Web as a platform.

The simplest solution, and most effective, was to use basic Web standards such as HTML, CSS, JavaScript, and AJAX to present a simple Web application that can be used through almost any Web enabled device, thereby gaining benefit from the 'Web as a platform' concept.

4.2 Web Command Line

The Web2Sh command line provides the user with the required interface to enter Web commands and write scripts. The command line itself is a simple HTML text area, where JavaScript is used to provide the required editing, running and HTTP streaming functionality. This takes place when the user presses *Enter* after typing a semicolon, indicating the end of a statement. The statement is then executed and a new line begins with the string `websh$`.

Figure 4-1shows a screen shot of the Web2Sh command line. It can be displayed on most modern browsers because it is a simple HTML page. However, JavaScript must be enabled since the AJAX technology depends on it to interact with the server.

Figure 4-1shows three main areas of display on the page:

- The Web command area, an HTML text area, is where users write and execute scripts.
- The result display area, which is where results, error messages, or feedback about executing a script is displayed.
- The download area, which enable the users to download files created during the execution of Web commands. This occurs mainly when the result is redirected to a file. Links to download those files are displayed in this area with feedback displayed on the main result display area.

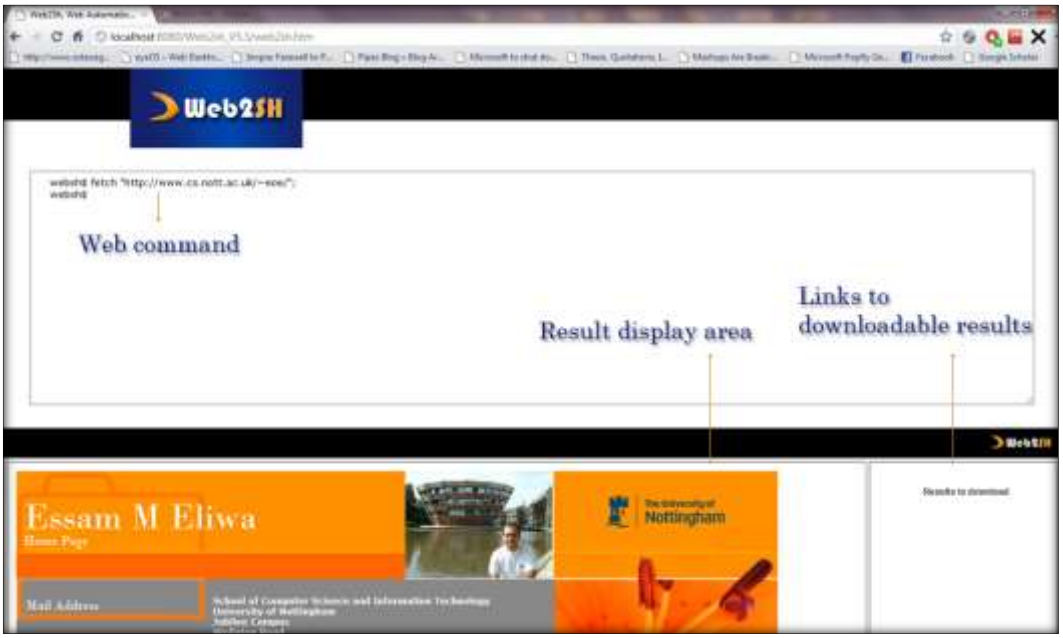

Figure 4-1: Web2Sh Command Line Interface

Users may also set the environment to display help for any command as soon as they write it on the command line. The command line also provides a set of basic functionalities that would be familiar to UNIX shell users. For example when a user presses the up arrow, it will print the last statement in the command history. If the user presses up arrow again the one before last in command history will be automatically written to the command line and so on.

Some of the commands offered through the command line are fully implemented on the client side using JavaScript. For example, the *clear* command does not need to be submitted to the server in order to clear the command line text area. Any command that is related to setting the command line environment itself is also implemented on the client side.

4.3 User Support

4.3.1 Login

The system does not yet support Hypertext Transfer Protocol Secure (HTTPS), but it can easily be added by obtaining a public key certificate for the Web server. Nevertheless, Web2Sh currently implements basic security procedures that enable the use of private personal content on the system.

On initial registration, the user enters an email address and selects a login ID. This will assign disk space on the server where any private data or personal Web commands can be saved. Users may use any core shell command without the need to login. Additionally, users may login to gain access to their personal private Web commands and their dedicated disk space on the server. Users can login using the login command on the shell.

The command takes one parameter, which is the login ID, and uses it to generate a random password that can be used for this session only. Web2Sh sends it to the user email and prompts the user to enter the sent password. Once the user has logged in,

he/she can write new commands, which will be saved, under the user's ID name (using the ID as the namespace for the command).

4.3.2 Web Commands Help

The user may use the `help` or `man` commands followed by a Web command name to display that command information with examples on how it may be used.

For example:

```
help "fetch";
```

This statement will display a summary describing the `fetch` command, its flags and parameters, and examples of its uses.

4.4 Error Handling

Web2Sh provides a variety of error handling techniques. All error messages are displayed back to the user's result area. The main types of Web2Sh error handling technique are:

12. For syntax and logical errors:
    - The basic syntax is validated by the parser, which reports any syntax errors;
    - Each Web command is checked by the Interpreter. If a wrong flag name or an unknown Web command was used it will generate an error message and terminate the pipeline execution.
13. For run time errors:
    - If a Web command generates an error, such as page not found, while operating, then it will return the error message back to the user but the pipeline will continue if there are any other available data on the stream.

4.5 Output Options

The Web2Sh default is to send a script output to the result area of the users interface. However, the user may change the values of some environment variables, using the `js.setEnv` Web command, to change the look of the interface or the output method.

For example, `js.setEnv $output = "new";`

This will change the user output mode so that output is redirected to a new window instead of being printed in the result area on the same page as the command line

4.6 Client Side Web Commands

Web2Sh uses JavaScript to implement a library of client side Web commands that may be better suited for executing on the Web browser. This enables Web2Sh to utilise the Web browser as a part of the Web platform. However, these commands are limited in functionality, since they cannot be used in a pipeline. Client side Web commands are usually used to effect user interface settings such as the `js.setEnv` command. Client side commands all start with the namespace `js` to indicate that this command is to be executed on the Web browser.

When the `js` namespace is detected at the beginning of a statement, the Web2Sh interface will execute this command locally. If the user tries to use a client command within the pipeline, an error message will be displayed.

For example, `js.fetch "http://www.google.co.uk":` will fetch the given URL and open the page in a new window.

`Js. getElementById ("item4"):` this command will search for an element with the id `item1` and display it on a new window. This can be useful to display only one part of the result from the last executed pipeline. For example, if multiple items were returned as a result for a pipeline, such as a set of links, each one will have a unique ID such as item1, item2, and so on.

`Js.print :` this command will print the page.

`Js.reload :` this command will reload the page.

4.7 Implementation Decisions

The Web2Sh framework targets the Web as its platform. Its use should not be limited to a certain Web browser or device. This means that the user interface needs to be generic in order to be used over the majority of Web browsers and Web enabled devices. The development of the user interface utilises only known Web standard languages and technologies, which are supported on all modern Web browsers. These languages/technologies are:

- HyperText Markup Language (HTML)
- JavaScript Language
- Cascading Style Sheets (CSS)
- Asynchronous JavaScript and XML (AJAX) technology

AJAX was a key enabling technology for the Web2Sh interface. It offers a set of JavaScript methods that are used on the client-side. With Ajax, Web pages can send data to, and retrieve data from, a server asynchronously without interfering with the display and behaviour of the existing page.

4.8 Summary

This chapter presented a description of the Web2Sh command line user interface. This is the client side component of the framework. A key feature of the user interface was to be generic and be usable over the majority of popular Web browsers. Hence, the development of the user interface utilises only known and popular supported languages and technologies such as HTML, JavaScript, CSS, and the AJAX technology. This enables the framework to be used without the limitation of using a specific Web browser or device.

The next chapter introduces the WSh scripting language that was created to support end users in creating and customising their own Web automation tools.

Chapter Five: **THE WSH SCRIPTING LANGUAGE**

5.1 Introduction

With the rise of Web-based applications in general, the Web is becoming an ideal platform to develop applications that were previously accessible only as desktop applications. For instance, Web users can use applications, such as eyeOs[18], which is an Open Source Platform (Web operating system and Web office) that resembles a desktop, but is actually in the cloud since the computing resources are delivered as a service over the Internet. The eyeOs system was licensed under the GNU Public License version 3 (GPL3); the latest version eyeOs 2.5 runs under The GNU Affero General Public License. It cleverly utilises PHP, JavaScript and Ajax technologies to present various desktop-like applications running on Web browsers. Google Web Applications such as Mail, Docs, Calendar, and Maps, are also very good examples of applications in the cloud

In today's Web, end users are shifting from desktop-based to Web-based applications and from local hard drive storage to network storage via Web servers. This, in turn, introduces the need for a shell scripting language for the Web, which would provide an adequate end-user tool to accomplish routine tasks. The availability of a Web shell and a scripting language to be used for building various Web commands/tools would be very useful for end users, programmers, and Web masters. A similar model is implemented on UNIX platforms where various shells and tools are available for users, programmers, and system administrators to facilitate the automation of various tasks.

This chapter describes the proposed WSh language; a Web-based UNIX-like scripting language for automating Web browsing tasks and for connecting together many diverse services that are available on the Web. WSh allows users to build, customise, and share their own Web commands to automate the execution of the variety of tasks that they may need to accomplish. The WSh scripting language can

[18] http://eyeos.org/

be considered as a step towards enabling end users to better automate and share their Web experiences.

The Web2Sh framework implements a thin client-computing model. This means that end users only need a Web browser in order to use, write, and run WSh scripts. The actual parsing and execution of the scripts is mostly handled by Web2Sh server. Web2Sh encompass the WSh scripting language. A significant advantage of the WSh scripting language is that it enables end users to customise and integrate Web commands in order to utilise many Web resources. For example, WSh facilitates the invocation of Web services functionality; a procedure that normally requires advanced programming skills.

Many scripting languages are used all over the Web; JavaScript, VBScript, and Perl are three examples of such scripting languages. These languages can also be used for other purposes, e.g. VBScript is widely used for writing macros in various Microsoft applications.

In UNIX, users can write scripts to be saved as a text file, called shell scripts. These scripts can be invoked by their names, just like built-in UNIX commands. Web2Sh follows a similar model where users can write scripts in WSh and save them on the Web server as Web commands. An important characteristic of WSh is that it utilises a similar pipeline approach to UNIX shells, which allows users to connect the output of a Web command to an input of another.

Some Web commands are able to accept different types of input, e.g. raw Text, HTML, or XML. In this case, the Web commands behave in a polymorphic manner. In other words, Web commands can alter their execution behaviour according to the MIME type of the piped resource. Users may choose to filter some resources to guarantee only certain parts of the resource are piped to the next Web command.

For example, if an HTML page has a list of students' ids, lab time and group, an HTML segment representing one student might resemble:

```
<tr>
 <td> ID2802 </td>
 <td> 4-6pm </td>
```

```
   <td class="purplegroup">Purple</td>
  </tr>
```

In this case, a user might prefer to strip all HTML tags and use a UNIX shell tool like `grep` to extract the piece of text that has a certain user's data in it. For example:

```
fetch "url" | txt | unix.grep "ID2802" ;
```

This script should print all lines containing a match to the given pattern. All HTML tags are stripped out because of the use of the `txt` filter.

After reading a line of text from a Web2Sh command line, the WSh Parser breaks the text up into words (tokens), and then determines word boundaries if quotes were used. The resulting tokens go through a check that substitutes values for variables and parameters.

Web2Sh interprets the resulting words as commands trailed by their flags and arguments. Web commands are either built-in commands, which are directly executed within the shell, or calls to external tools or programs, in which case, Web2Sh redirects the Web command to the appropriate wrapper (e.g. the UNIX Tool wrapper) to handle the command.

The WSh scripting Language supports a standard suite of programming constructs, including variables, lists, arithmetic operations, strings, quoting, and pattern matching. For flexibility and ease of use, the WSh scripting language is designed to be case-insensitive, to avoid common case related errors that inexperienced end users may make. This is in contrast to other languages, which are case-sensitive and take a JavaScript similar approach, such as the Chickenfoot scripting language.

The following is an example that demonstrates the ease of use of WSh in comparison to Chickenfoot. For the following Chickenfoot script:

```
go("google.com")
enter("Web 2.0")
click("google search")
links = find ("link in bold")
links.count
```

The alternative script in WSh would be:

```
Google "Web 2.0" | getLinks –p1 | find –i
"<[b|strong]>" | count;
```

The Web command `getLinks` will get all the links on the page returned by Google search, the `–p` flag indicates that links will be returned alongside their parent tags, the *find* command would search for any links that have the tag `<b>` or `<Strong>`. A word in the link or the whole link can be in bold. The `–i` flag indicates that `find` should match the pattern regardless of its case. The last command `count` would return the number of the links that matched the search criteria.

5.2 Scripting Language for Web Automation

Experienced programmers as well as casual programmers (i.e. those who are less-experienced), may use a programming language in order to implement algorithms to achieve a higher level of automation in their daily computer usage.

Basic programming knowledge can be useful for non-expert computer users in order to save them significant time doing repetitive tasks. Almost all major platforms offer simple scripting languages to users in order to empower them to better utilise the platform and enable automation. Web2Sh follows this model by providing a scripting language that can be used to create and customise Web commands that are focused on Web automation. One of the main advantages of the Web2Sh model over most other platforms' models is that Web2Sh is designed explicitly for the Web.

For example, consider a user who has compiled a list of laptops that he/she is interested in, through a set of shopping Web sites. The user wants to check which ones have received good reviews on sites like Cnet[19], Amazon[20], and PcWorld[21]. Moreover, imagine that the user also wants to check technical performance issues of some of the laptop parts on sites such as `notebookcheck.net`.

This could be done by visiting each of these sites, providing details of each laptop, then manually compiling the necessary information to carry out the comparison that would assist in making the decision. Some services may be available on specific Web sites such as Dell that make this comparison easier, yet this applies only for the devices they offer. This service may also be provided by a Web site that offers price comparison. However, if the user wishes to customise the process in ways, which truly reflect his/her own needs and mental model, then the possibility that a Web site can provide such specific format is not high. For this reason, end users might then need the ability to write scripts that would automate their Web tasks in a personalised manner. In other words, users may often need customisable tools that simplify the process of automating the Web. The ideal approach for such an objective is through a scripting language that allows the development and execution of scripts that take less time than it would take to do a task manually.

The Web is shifting towards being a platform. Most computer users spend significant part of their time interacting with the Web in different ways. Allowing end users to better utilise the Web is essential, and a goal that many researchers have pursued.

Many available tools for automating Web tasks (Rob, Paul et al. 1997, Bolin 2005a, Jeffrey and Richard 2007) require the user to work either with the raw semi structured HTML of a page, Document Object Model (DOM) (Hégaret 2005) of a Web page, or simply the rendered model of a Web page. There is also many Web scripting languages, however, they are still mainly targeted for developers and Web programmers. Regular Web users lack both the languages and tools to automate their Web usage.

[19] http://www.cnet.co.uk/
[20] http://www.amazon.co.uk/
[21] http://www.pcworld.co.uk/

A Web page can be represented as a long string of HTML, and users identify parts of the page either by location or by regular expression matching. However, as the HTML for almost all dynamic Web sites is generated by code executed on Web servers, it is often hard to understand by end users. So writing scripts can be both difficult and time-consuming.

Users may also access and manipulate a Web page's content based on its DOM, which is a standard model for Web page structure; JavaScript is the most popular scripting language utilising the DOM. However, this model requires adequate programming experience, including basic understanding of Object Oriented concepts. In addition, users will still have to understand the HTML structure of the Web page.

Furthermore, such tools or scripts would have a fragile dependency on the structure and text in a Web page, which may break when the page changes. Therefore, it is crucial to be able to update the scripts the user may use to adapt to possible changes in any Web page.

Hence, to address these limitations, users need to be able to have the option to make use of the HTML structure, Document Object Model, or to depend on the actual page content to achieve their goals. Users may choose a different methodology to achieve their goals, based on their programming experience and the type of task they need to achieve.

The WSh scripting language aims to offer a wide set of tools to give its users the ability to access and manipulate Web pages using the best model that matches their requirements and experience. Hence, the main objectives of the language design are simplicity, generality, usability, and expressiveness. Furthermore, writing and testing the language is done through a Web page, as it is the most common, well-known and simple interface to use.

Web2Sh enables users to work with different representations of the Web page. It provides tools that support the many MIME types that may be retrieved over the Web such as images, RSS, and PDF files.

Web users are always searching for easier and more efficient ways to instruct the Web platform to execute useful and complex tasks. For this purpose, this thesis argues that users need new powerful and easy to use tools. Such tools should allow the user to enter, edit, translate, and run Web commands on the Web platform. Web2Sh provide such tools for the users; all tools are Web-based, hence, the user only needs a modern Web browser to create, edit, share and run Web2Sh Web commands.

The Web can be considered as a high-level platform, on which programs are written and executed. It also comprises the use of many technologies and Web programming languages such as PHP, Java, Perl, Dot Net, among others. The users need not worry about any low-level operations, which are taken care of by protocols such as TCP/IP, HTTP, FTP, and similar protocols.

5.3 Software Tools Philosophy

The software tools philosophy is well exemplified by the UNIX operating system. The essence of this philosophy is to provide a small number of universal and simple tools, which can be used in a variety of combinations to achieve any number of super-tools for specialised work on individual problems (Lawler 1998).

Web2Sh applies the same philosophy, where it utilise the Web as its target platform. It offers well designed and intelligent Web commands, which can be connected together using the WSh Scripting language.

The basic principle is to offer a number of small, well-crafted, and bug-free tools, that:

- Do only one well-defined task;
- Do it intelligently and well;
- Do it in a standard and well-documented way;
- Do it flexibly, with appropriate user-chosen options available;

- Take input from, or send output to, other program tools from the same toolbox;
- Do something safe, and if possible useful, when unanticipated events occur.

5.4 WSh Language Deign

To design a programming language several aspects need to be specified (Watt and Brown 2000):

Syntax: defines what symbols (tokens/terminals) are allowed to be used in a program, and how phrases are composed of tokens.

Contextual Constraints: defines context related rules such as *scope rules* and *type rules*.

Semantics: describes the meaning of the program, semantics from an operational point of view is the program behaviour when it is executed.

The Web2Sh framework required the development of a WSh scripting language to deal with various input types of resources retrieved from the Web. The structure and meaning of the language needed to be defined by complex formal rules. Although developing this language from scratch is complex and error-prone, many tools such as Lex and YACC (Levine, Mason et al. 1992) offer a good approach to develop such language.

The most suitable available tool for this work was the Java Compiler Compiler (JavaCC). It utilises Extended Backus-Naur Form (EBNF) to specify the syntactic rules of the language. It generates the parser from the defined rules, where all the generated files are Java classes, making it easy to integrate with any Java project. It also had the advantage of available support for use within the eclipse integrated developing environment[22].

[22] http://sourceforge.net/projects/eclipse-javacc/

5.4.1 Top-Down Parsing:

Top-down parsing is widely used to parse language grammars. A Top-down LL Parser is usually used for context-free grammars. It parses the input from Left to right, and constructs a leftmost derivation of the sentence.

Web2Sh implements the top-down parsing strategy to implement a Parser for the WSh scripting language. It uses a top-down LL Parser originally generated by the JavaCC tool based on the grammar definition in Extended Backus–Naur Form (EBNF).

First, it indexes its start set which is the set of Nonterminals or Terminals that may start any production rule. Secondly, any input is compared with the start set to define which production rule may apply. Finally, if any two production rules share the same start the Parser may look ahead to see the next symbol until it matches a unique production rule, this behaviour is specified by using numbers, such as LL(0), LL(1) and so on.

LL(0) would be used if there were no need to look at any token, i.e. only one production rule applies. LL(1) would mean look only at the first token of the input, if used this implies that no two rules share the same starting token.

5.4.2 WSh Parser:

The shell Parser reads its input from the input stream, and partitions it into statements. The Parser splits each statement into tokens. The Parser works in these steps:

- Split the program by statements (statements are terminated by a semicolon)
- Split statements by tokens;
- Check for syntax errors or unrecognised tokens;
- If no errors are found, build a command Abstract Syntax Tree (AST) where each terminal/nonterminal is represented by a node (e.g. a Web command);

- The Interpreter traverses the AST, checks the semantics, and executes each instruction;
- If errors are found, return an error to the user and stop the script execution;
- Look for the longest match possible at each input point;
- If necessary, a look ahead is performed to choose the output format.

5.4.2.1 Command Execution:

The Interpreter traverses the AST passed from the Parser, and then it checks the semantics of instructions, i.e. it validates that the correct flags and parameters are used with each command. A pipeline will execute in the following manner:

- Read available input from the input stream;
- Pass the input to the first command in execution order;
- Runtime errors may be ignored if there are other Web commands in the pipeline that have available data to be processed;
- Run time errors will be passed to the user, such as “page not found error.” For example, in a script, that gets a list of links and does some work with them, if one page is not available, the rest of pages will be processed normally.

5.4.2.2 Input /Output:

- Each Web command has an input stream and output stream;
- Utilise the use of `PipedInputStream` and `PipedInputStream` as the raw input/output;
- The MIME type of each resource is retrieved; this reflects on the provided functionality. E.g., the `title` Web command will return a PDF document title if the mime type is `application/pdf` using the title entry in the

document information dictionary of a PDF document, while if the mime type is `text/html` the command will return the value of the `title` tag in the header section of the HTML page.

- If the command can produce various output types, then a negotiate method decides, for each command, which is the best output format to produce based on the following commands. If no preferred format exists, the default will be used.

The possible drawback of an interpreted language is slower execution compared to a compiled program. However, the advantage of flexibility and ease of use is very important for a language that is targeted for Web users.

5.5 WSh Scripting Language Design

WSh is a simple command language specialised, focused, targeted and utilised for automating Web tasks. The WSh environment (as part of Web2Sh framework) consists of:

An editor*:* tailored to WSh, this allows users to enter, modify, and save commands on the Web. The editor works inside any common modern browser with JavaScript support.

A parser*:* this runs fully on the server side. When the user writes a script, the Parser checks the syntax and reports any errors. If no errors are found, it constructs an Abstract Syntax Tree (AST) representation of the program and passes it to the Interpreter.

An interpreter*:* this is integrated with the Parser through the implementation of the Visitor design pattern. It visits any Web command node in the AST generated by the Parser. Then it runs it immediately by calling the Java class that executes the command task. The output is then returned to the User's Web browser, piped to another command, or redirected to a file.

5.5.1 Language Informal Specifications

5.5.1.1 Programming Concepts

These are the basic concepts of WSh (Web shell) programming. This section introduces the concept of *item*, *Web command*, *tee*, and *pipe* in the context of this research. Each discussion focuses on how these concepts relate to the real world, while simultaneously introducing the syntax of the WSh scripting language.

5.5.1.2 WSh item

The language will need in many cases to deal with objects with attributes, getting regular users to understand object-oriented concepts could be harder when compared to simple xml tags with attributes.

```
<book name= "..." author= "..." ISBN= "..." />
```

WSh calls this an `item`, where `name` or `id` attribute will be the item identifier. For simplicity, only this form will be used to create items. Not the nested tags form as in XML.

WSh integrates the mark-up language's tag structure as a valid way to define a variable; this has proved useful when handling much of the Web data that is included within both HTML and XML content. Hence, WSh introduce the concept of an *item* as a generic data type used to store variable values in WSh. It is similar to an object with no methods in Object Oriented languages.

It has the following characteristics:

- It has a sub-type (tag name), a set of attributes and values;
- Each attribute name must be unique within an item;
- It is represented with a similar syntax to that of a simple XML tag.

Example 1:

```
1   $film = <movie name = "Inception"
2   IMDBrank = "9.1"
3   genre = "Action, Mystery"
4   director = "Christopher Nolan" />;
```

This statement creates a variable of the generic *item* type named `$film`. It has the sub-type 'movie' and attributes `name`, `IMDBrank`, `genre` and `director`. However, default printing of `$film` prints an empty string as there is no value attribute and the tag has no content.

To print the whole item an 'item' flag option needs to be used. For example, the Web command `echo` can be used to print the values as in the following code:

```
1   echo $film //prints nothing to the screen
2   echo $film –item
```

Example 2:

```
1   $link = <a href = "url"> link text </a>;
```

Creates an item variable of sub-type 'a' (an anchor) with an attribute `href` and its value equal to `link text`.

```
1   echo $link; //prints link text
2   echo $link –item; //prints the XML tag
3   echo $link –href;
4   //prints the value of href attribute
```

5.5.1.3 Web Command

A Web command is the main building unit for executing tasks on WSh. It comprises a unique set of valid WSh instructions that carry out a certain task. A *Web command* takes by default an input stream `$STDIN` and passes its output to an output stream `$STDOUT`.

Web2Sh users may use available commands to execute a certain task or chain a set of commands together in a pipeline to construct a new Web command. Chapter Six explains this concept in more detail.

5.5.1.4 WSh Pipeline

Similar to UNIX, pipes are used to pass the output of a command to the input of the following command. The native nature of the input and output streams ($STDIN, $STDOUT) is a stream of characters, however, these are wrapped as items (XML-based structure) in order to enable users to easily access and utilise Web content.

The Web2Sh framework uses a pipe symbol "|" to connect the standard output of the preceding command with the standard input of the following Web command. The pipe also always acts as a for-each, so for example if the first command returns a list of links the pipe will process each link through the rest of the pipeline.

5.5.1.5 The tee Web Command

The `tee` Web command is a special Web2Sh command. Similar to the UNIX `tee`, it permits the construction of complex pipes. The idea is to split a command output into two threads for debugging or advanced processing. In addition, it can be used to save a pipeline output to a file.

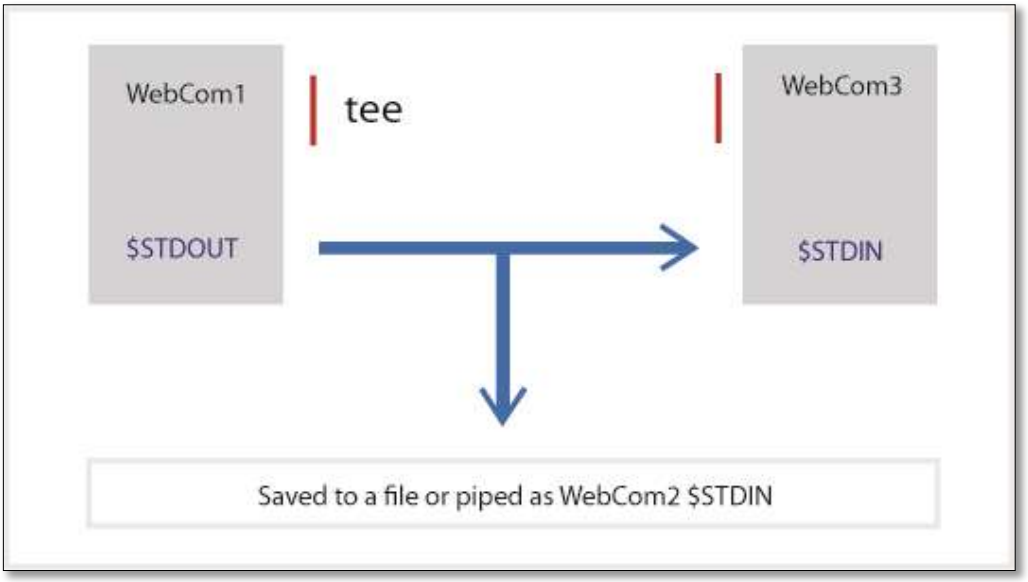

Figure 5-1: Using tee Web command

In the following example, `tee` is used to split the output of a Web command so that it can be saved in a file and piped to Web2Sh's command line interface result area at the same time. It is can also be used to capture intermediate output before the data is altered by another Web command or program. The `tee` command reads standard input, then writes its content to standard output and simultaneously copies it into a specified file, variable, or another Web command. The syntax differs slightly depending on the used command's implementation.

The following pipeline returns Hello World to be printed on the Web browser and save it to a file on server named file.txt:

```
1   echo "Hello World" | tee "file.txt" ;
```

The next script searches Google for `Egypt`, saves the result to `file.txt`, and then pipes the output to the find command, which will print to the screen any results that have `Pyramids` word in it.

```
1   google "Egypt" | tee "file.txt" | find "Pyramids";
```

`tee` is also useful in complex pipes debugging. Using the `tee` command anywhere in a pipe command sequence will divert a copy of the standard input (of `tee`) to allow analyses of the correctness of this stage.

```
1   google "Egypt" | tee print -title-b
2   | fetch | getLinks | print -item-ol ;
```

This piped Web commands sequence (pipeline) will run in this manner:

14. The `google` command to search for the word `Egypt` the result is a set of items of type `gRes`. Each item has attributes title, text, and URL.
15. The pipe will run a `forEach` on the returned item list.
16. `tee` passes a copy of the item in $STDIN to `print` command which prints the title of the item in bold.
17. Another copy is passed to the `fetch` command, which defaults to extracting the `url` attribute to retrieve the Web page source.
18. The `getLinks` command extracts all links in the page source then prints them.

The result will be an aggregation of each Web page title followed by the set of links on that page.

5.5.1.6 Syntax

This section presents the syntax of WSh and illustrates how symbols (tokens) are used to compose valid phrases.

5.5.1.7 Start Symbols:

- `start` : starts a multiline WSh script, the script is executed when the `}end;` token is reached;
- `new` : used to declare a new command; a command has to be called in order to be executed;
- Any Web2Sh command is a valid start for any sequence of command calls separated by the '|' operator.

5.5.1.8 Terminals (list of keywords):

- `tee`
- `true`
- `false`
- `null`
- `new`

5.5.1.9 Operators:

- Assignment: `=`
- Math: `+ , - , *, / ,%`
- Redirection : `=> , =>>` (the first to create new file, the second to append)
- Pipe : `|`
- The Web command flag prefix: `-`

- String concatenation operator: + (decided by the nature of the first operand, if a number, the second operand will be tested; if a number too, it will be used as math operator. Otherwise both operands will be concatenated as strings)
- Separator for parameters or items lists : `,`
- Mandatory variable name prefix : `$`

5.5.1.10 Production Rules

Production rules are a fixed set of rules that define how statements are composed from terminals. In WSh the production rules are:

- Statements are terminated by a semicolon `;`
- A block of code is started with an opening curly brace and terminated with closing curly brace.
- Web command parameters start with opening round bracket `'('` and end with a closing round bracket `')'`

5.5.1.11 Nonterminals:

Nonterminals represent a particular class of phrases in the language such as “program” and “expression.”

- Program: is a set of valid Web2Sh instructions.
- Expression: Arithmetic, Boolean, and String expressions are written in conventional mathematical infix notation, adapted to the Unicode character set.
- Web command: A Web2Sh command followed by optional list of flags followed be optional list of parameters
- Simple statement : a valid Web2Sh single expression or Web command terminated by a semicolon `';'`

- piped statement: A sequence of Web commands separated by pipes and terminated by a semicolon `';'`

5.5.2 Contextual Constraints:

Defines context related rules such as scope rules and type rules.

5.5.2.1 Global Built-in Shell Variables:

- $STDIN: a predefined variable that holds the input stream which is passed by default by the pipe
- $STDOUT: is a predefined variable that holds the output of the last executed command. The Interpreter directs the output to the screen or next command in a pipe sequence or a file.

5.5.2.2 Variable Naming:

These are the rules and conventions for naming a Web2Sh variable:

- Must start with `$` followed by a letter;
- Contains no whitespaces;
- The initial character can be followed any number of letters, numbers or underscore characters;
- Web2Sh variables are not case-sensitive. However, consistent capitalisation makes the code easier to read;
- It is recommended to use variable names in all lowercase letters if the name consists of only one word. If it consists of more than one word, capitalise the first letter of each subsequent word;
- Built-in variables are used in upper case by convention.

5.5.2.3 Naming Web Commands

- Web command names are globally unique, namespaces can be used to avoid conflicts;
- A core (global) Web command can be used by all users;
- When a user creates and shares a useful command it can be upgraded to a global command by a Web2Sh moderator;
- Periods separate the namespace prefix from the command name;
- Each user will have a unique user name on the system, which represent his namespace. When a user ‘user1’ creates a command ‘`testCom`’, it will be automatically named ‘`user1.testCom`’. Using `testCom` by user1 will first look for the command in the global command list, if not found it will be looked up in his name space;
- Using ‘user1.testCom’ and ‘`testCom`’ by user1 is the same if there is no global command with the name ‘`testCom`’.

5.5.2.4 Variable Types

- Variables are automatically defined the first time they are used;
- All variable are of the same general type ‘`item`’;
- A Web2Sh item is like an object but with attributes only;
- *Name* and *ID* attributes are used as the item identifier;
- *Value* attribute is used to set the item’s value.
- Each item has a *type* attribute that can be used to specify more metadata about the type of the *value* attribute. For example:

`$num =5;` generates the result:

`< number  name = "num" value = "5" />`

- The *tag name* (item sub-type) is defined automatically by the shell and default to string if no other type is detected;
- Item sub-types are transparent to users and mainly used by the shell, these types are useful for the command to decide on the action to take. Usually they are stemmed from MIME types, an example of an Item's possible types are :
    - String
    - Number
    - Double
    - Char
    - Html
    - Xml
    - RSS
- HTML content is processed first to replace <, > ,\ and " with \<, \> , \\and \" (using \ as escape character)

5.5.2.5 Variables Scope:

- Defining a variable on the command prompt makes it visible to all of the current session.
- For any block of code, a variable is visible only inside its block.

5.5.3 Semantics

This section describes the WSh scripting language semantics through navigating some Web command examples, showing various functionalities of the language and the behaviour of Web commands when executed.

5.5.3.1 Basic Web Commands

This simple Web command pipeline will return the first 10 links from a Google search result page, using `Egypt` as the search keyword, and auto loop to get all images available via each result URL.

```
1   google -10 "Egypt" | fetch |
2   getImages;
```

The pipe operator '|' will connect the $STDOUT from the command on the left side with the $STDIN of the command on the right side.

The semicolon character '`;`' indicates the end of command pipe sequence. It allows a pipeline to be written on multiple lines. When a user presses enter after writing '`;`' the Web commands are executed and the result displayed on the Web browser.

5.5.3.2 New Web command example:

```
1   new googleWordCount ($searchWord){
2    $str1 = "found ";
3    $str2 = " times";
4    google $searchWord | fetch | txt |
5    find $searchWord | count > $i;
6    echo $searchWord $str1 $i $str2;
7   }end;
```

Line 1: `new`, marks the beginning of a new command definition named `googleWordCount`. If the name already exists, the system will generate an error message.

The left parenthesis '`(`' marks the start of the parameters list, where `$searchWord` is the parameter that is needed by the command. The right parenthesis '`)`' marks the end of the parameters list. Finally, the left curly bracket '`{`' marks the start of the Web command code block

Line 2: `$str1`, defines a variable. The '=' assignment operator, assigns the value on the right side to a variable on the left side. The string value "`found `" is surrounded with double quotes. The semicolon '`;`' is used to terminate a statement.

Line 3**:** this has similar semantics to the second line.

Lines 4, 5: This is a Web2Sh pipeline, starting with the `google` command that is executed using the value of `$searchWord` as its input parameter. The command returns a list of items each representing a link returned by Google search. Then the pipe loops over all returned links to fetch them. The following Web command, `txt`, filters the html tags keeping only the text content of the Web page. The result is forwarded to the `find` Web command, which return all lines where `$searchWord` is found.

```
google $searchWord | fetch | txt |
find $searchWord | count > $i;
```

Web2Sh executes the pipeline in the following order:

19. `google`: use Google's search engine to search for the given term.
20. `fetch`: all URLs returned as the search result.
21. `txt`: pipe the page to the 'txt' command which strips all the tags and codes and leaves only the text content of the page

22. `find`: The result is piped to find command, which finds the occurrences of the search word inside the text and returns a list of items.

23. pipe the result of the `find` Web command to the `count` Web command which returns the number of times the search word was found inside the text

Line 6: `echo $searchWord + #str1 + $i + #str2 + $xurl;`

This statement will print to the default output (user's browser) the concatenated string.

Line 7: This is the close curly brace for the end of new command. '`end;`' is used to finish the '`new`'.

5.5.3.3 Artwork Importer, iTunes Example:

This command will use an XML file called 'iTunes library'; this will need to be uploaded to any online Web address and then it will extract the value of each album name and artist name. This will be used to build a search string for Google images to try to retrieve the album artwork. The first three images of each album search will be saved to a folder with the album name.

When the user chooses to create a new Web command, the script is saved to his private disk space on the Web2Sh server. Then the command can be called by the user assigned name space. For example, `eoe.egNews;`

All returned images are added to a zip file and downloaded to the user machine. After the command, finishes all the saved data on the server are automatically erased.

For example for this entry:

```
<dict>
    <key>Track ID</key><integer>42</integer>
    <key>Name</key><string>Sandra - Don't Cry</string>
    <key>Artist</key><string>Sandra</string>
```

```
    <key>Album</key><string>18 Greatest Hits</string>
    <key>Kind</key><string>MPEG audio file</string>
              ..............
</dict>
```

This search phrase will be generated:

"Here by Me" + "3 Doors down" + "Seventeen Days" + “Album”

```
    /* param $iTunesMusicLibrary : is the URL for the
    iTunes Music Library XML file.*/
1   new iTunesArt($iTunesMusicLibrary){
3      $list = XPath $iTunesMusicLibrary
4              “/plist/dict/dict”;
5      $list | find “ <key>Name</key>” |
6      getTagValue “ <string>” | $songNames;
7      $list |  find “ <key>Artist</key>” |
8      getTagValue “ <string>” | $artistNames;
9      $list |  find “ <key>Album</key>” |
10     getTagValue “ <string>” | $albumNames;
11     Echo $songName, $artistName, $albumName, “album” |
12     gImages -3 | Zip “artwork/” + $albumName;
13  }end;
```

5.6 Language Formal Syntax:

To specify a programming language syntax in a precise way an informal specification written in English is not enough; a formal specification in a formal language, such as Backus–Naur Form (BNF) or EBNF (Garshol 2003), is needed.

An EBNF language description consists of:

- A fixed set of *Terminals*: These constitute the symbols and tokens of the programming language, such as “new” keyword;
- A fixed set of *Production rules*: These define how statements are composed from terminals;
- A fixed set of *Nonterminals*: These are a particular class of phrases in the language such as “declaration” and “expression” Nonterminals;
- A *Start symbol*: This nonterminal denotes the start of the program.

EBNF is used to define Web2Sh syntax (Appendix-A) as it is more well-ordered than BNF, this is because EBNF offers three operators (`'?' , '*' , '+'`) for better handling of recursion and repetition.

5.7 WSh Language

Web2Sh is a simple Java based command language specialised, focused, targeted, and utilised for automating Web tasks. Web2Sh is actually two things: a language and a library of commands.

First, WSh is a scripting language based on the use of pipes, intended primarily for creating and issuing commands to automate Web tasks. It has a simple syntax and is easily extendable, allowing Web2Sh users to write new Web commands, customised to their requirements, to extend the built-in set of commands.

Furthermore, the shell offers a way to call UNIX shell commands and Web services and integrate its functionality with Web2Sh commands. Other similar systems can

also be integrated by implementing specific wrappers within Web2Sh to make external calls to those systems.

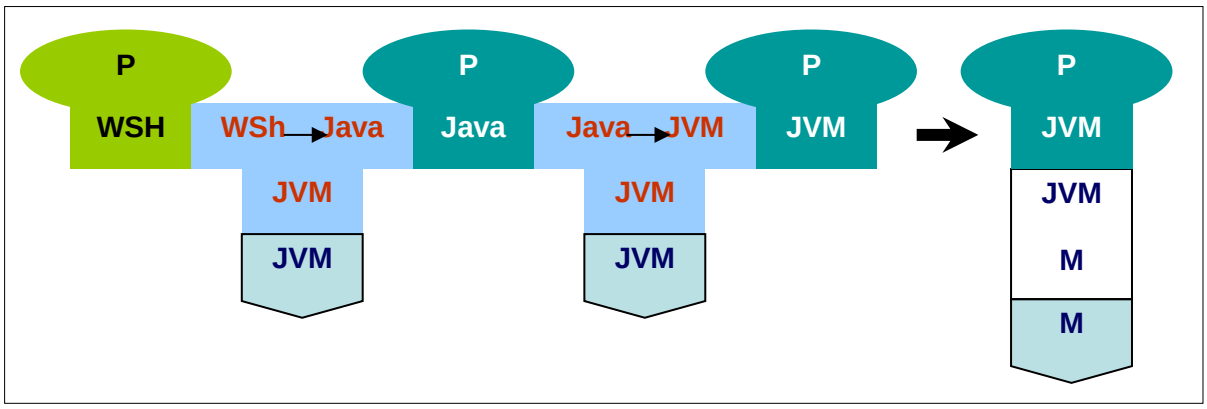

Figure 5-2: WSh Language processor using a tombstone diagram

5.7.1 WSh Parser and Interpreter

This section addresses the design and implementation of WSh Parser and Interpreter, which parse and execute WSh scripting language. Parser and Interpreter design has been a research topic for several decades and there exists a lot of literature about it. A well written and practical introduction is given in (Watt and Brown 2000).

JavaCC is a lexer and parser generator for LL(k) grammars (Kodaganallur 2004). The user needs to specify the language's lexical and syntactic description using EBNF syntax. The generated parser merely validates the Syntax of the language based on the defined EBNF rules. The actual functionality that each instruction executes needs to be implemented and integrated into the Parser.

The WSh Parser is implemented with the use of the JavaCC tool. It traverses the whole source program, parses the code, and reports any syntax errors. The Parser generates an Abstract Syntax Tree (AST) that supports the use of the Visitor Design Pattern. The WSh Interpreter interprets the script by traversing the AST generated by the Parser. Each node class in the AST includes the method:

```
public Object jjtAccept
      (web2shParserVisitor visitor, Object data) {
          return visitor.visit(this, data);
}
```

`web2shParserVisitor` is an interface that must be implemented by any class that needs to add functionality to each node, the Interpreter in this case. This was explained in more detail in Section 3.4.1.

Parsing is the first pass in the interpretation process, and its purpose is to parse the WSh script source and construct the abstract syntax tree (WSh AST), which is a tree representation of the source program.

5.7.1.1 WSh AST

The WSh AST is a tree structure of Java objects. The nodes of this tree represent the production rules of the recursive WSh grammar and the tree is a complete description of the source script. Constructing an abstract syntax tree representation of the source program brings the benefit that each of the following passes can operate on one and the same data structure which is easy to transform and to annotate with context information. Abstract syntax trees are explained in detail in (Appel 1998) and (Watt and Brown 2000)

5.7.1.2 WSh AST Parsing

First, the Parser checks the source program to verify that it is well formed, according to WSh syntax. Secondly, the WSh high-level translator performs a set of transformations on the abstract syntax tree. The transformations have the following purpose:

- Annotation of the AST with information needed such as deciding on each command input/output type.

5.7.1.3 Parsing Variables

The language needs, in many cases, to deal with objects with attributes, getting regular users to understand object-oriented concepts could be harder when compared to simple XML tags with attributes.

Hence, Web2Sh uses *item* as a general type, where the `name` or `id` attribute will be the item identifier. In addition, a sub-type is decided based on the tag name. Web commands are needed to be able to create and manipulate many markup tags; hence, the item structure is very useful.

```
1   $imag = <img src="url" alt="text"/>
```

The example above creates an item variable of sub-type `img` (an html image). Each variable is treated as a special command. If followed by a pipe, then the command will return its own value to the next command.

So both of these forms are valid:

```
1   $link  = <a href="url"> link text </a>
2   echo $link;
3   $link | echo ; // will print link text
```

Using any attribute name as a flag will return this attribute value

```
1   $link –href | fetch ;
```

- Item attributes must be unique ;
- An item value will be decided in this order:
    - The tag content may contain other items, any valid UTF characters, or empty;

- The attribute “value” content may contain any valid UTF characters or empty.

When an item command is called, its tag content will be checked and, if found, it will be returned. If not found, the Interpreter will look for the default attribute `value` content. If found it will be returned, otherwise, an empty string will be returned.

If an item has both a tag value and a `value` attribute, only the tag value will be returned. To access the `value` attribute, a flag with the attribute name must be used. For example:

```
1   $item = <string value=”test attribute”>
2            inner tag text</string>
3   echo $item; //will print “inner tag text”
4   $item –value |
5   echo ; //will print “test attribute”
```

WSh variables are able to hold a list of values. This feature was needed as many Web2Sh commands produce a list of items as their result.

For example:

```
1   fetch “http://www.bbc.co.uk” | getLinks
2   | $bbcLinks;
```

This will retrieve all available links on the BBC home page and store the values as a list in the `$bbcLinks` variable.

5.8 Summary

The work in this research required the implementation of a dedicated scripting language for achieving the goal of end-user Web automation. This chapter presented the building process of Web2Sh dedicated scripting language (WSh). It explained the software tools philosophy employed by WSh. It also presented the background for defining new scripting languages. The design and development of WSh language was explained, and the language's informal and formal specifications were introduced.

Chapter Six: **WEB COMMANDS**

6.1 Introduction

This chapter presents Web commands as the basic building block for writing Web automation scripts. A detailed description of the syntax, structure, and characteristics of Web commands is provided and the Web2Sh implementation of the pipeline concept is discussed.

Web commands are tools and filters created with the specific goal of providing solutions for automating a set of problems in various domains connected to user/Web interactions. Moreover, the generic approach of Web2Sh allows for connecting any set of Web commands using pipes. This provides general solutions for a large set of problems to meet Web users' needs.

Web2Sh offers a repository of basic commands in various domains as a proof of concept. The goal is to enable end users to develop new commands and be able to customise existing ones. In addition, the collaborative nature of the framework offers good potential to utilise end users' experiences and to develop and share commands in one location.

An important objective of the Web command class design is to offer support for different resource types. To this end, the functionality of many Web2Sh commands is designed in a way that supports different approaches based on the MIME type of the resource piped to it. Web2Sh achieves this by implementing the command pattern to handle a command execution, where the action of a command is decided by the receiver class (Web resource).

The receiver class in the command pattern can be any class representing a valid MIME type. For additional generality, as the resource type is only decided at run time, the receiver object is generated dynamically through the implementation of the factory design pattern. This thesis refers to this concept as *polymorphic MIME types* support. This implies that each command may function in a logically related but different way according to the type it handles.

According to this design feature (i.e. *Polymorphic MIME types*), each Web command may support different functionality for different resources. For example, the `title` Web command will return the title tag value in the case of an HTML page, while it will return the value of the file property title in case of a PDF file.

The context of the Web command may also alter the actions it takes. For example, the `getImages` Web command performs a look forward to see if there is a following Web command in the pipeline. If there is not, it will extract all `img` tags in its input stream and write them to the output stream. Otherwise, if the following command does some image processing functionality, the `getImages` will retrieve the actual image, save it, and writes its URL on Web2Sh server to the output stream.

The super class `WebCommand` specifies that a Web command imports a set of java library built in classes, most significantly the `java.io.PipedInputStream` and `java.io.PipedOutputStream` as the standard input and standard output for each command. This allows commands to deal with input as a piped stream of bytes and produce output as a stream of bytes. The `PipedInputStream` and `PipedOutPutstream` classes allow a Web command to convert an `OutputStream` into an `Inputstream` of the following command in a pipe. The idea is that at one end of the pipe, a writer thread writes to a `PipedOutputStream`. A `PipedInputStream` thread concurrently reads whatever is written on the other end.

The use of `PipedInputStream` and `PipedOutputStream` classes has many advantages making them ideal to use to implement Web2Sh pipes. A piped output stream can be connected to a piped input stream to create a communications pipe. This technique is useful as it supports communication between threads, allowing each command to run as a separate thread, produce its output and pass it simultaneously to the following command in a pipe sequence.

6.2 Internet Media Types

As the IETF Network Working Group specified in its Request for Comments: 2046 (Freed and Borenstein 1996), there are five discrete top-level Internet media types :

24. text
25. image
26. audio
27. video
28. application

Each of these top-level types has sub levels of more specific types such as the sub levels for text:

- text/plain
- text/html
- text/css
- text/javascript
- text/csv
- text/xml

The Network Working Group can update these types according to the new needs that may emerge over the Web.

All command classes reside in the `org.Web2Sh.mimetypes` package. This package offers implementation of the five top-level Internet types as super-classes and implementation of some of the sublevels as child classes.

In Web2Sh, many Web commands will perform the same action for the same family of MIME types (i.e. text or image); hence, the command action is implemented in the super class `Text`. In the case of a Web command that may have different actions for

different members of the same parent MIME type, then the action of the command is implemented in the child classes (i.e. HTML, CSS, or CSV)

6.3 Polymorphic Mime Types

The *Polymorphic MIME types* design feature gives the Web commands the ability to change their behaviour to adapt to a resource's Internet media standard types known as the Multipurpose Internet Mail Extensions (MIME) standard (Freed and Borenstein 1996). For example, the `fetch` command when applied to a resource of type text this would mean to retrieve the HTML resource and simply display it on the result frame. If the command was called for a resource of type `image`, this will mean by default to retrieve the actual image on the server then offer to download it for the user. Of course, such functionality can be altered according to the flags used with the command, as explained later in this chapter.

At this point, Web2Sh covers mainly `Text` and `Image` types as a proof of concept in addition to the `application/pdf` type. However, supporting other MIME types only needs relevant code implementation. The design of the framework enables such extensions easily without changing the framework itself.

6.4 Web Command General Syntax

A Web command has the following syntax defined in EBNF:

```
WebCommand ::= WebCommandName {Flag}{Parameters};
WebCommandName ::= Letter {LetterOrDigit | '_'} ;
Flag ::= "-" LetterOrDigit {LetterOrDigit};
Parameter ::= (String | VariableName);
Parameters ::= Parameter {"," Parameter };
```

This means a Web command may consist of the command name, optionally followed by one or more flags, which may optionally be followed by one or more parameters.

The Web command name can be any valid unique identifier. It must start with a letter followed by any combination of letters or digits or underscore _. Note that, the WSh scripting language is case insensitive.

Each flag needs to start with a hyphen character '-' followed by any combination of numbers and letters. A flag is used to change the way a Web command may behave, thereby giving the end users more flexibility when using the commands.

A parameter can be a valid string or a variable name. A set of parameters can be used with comma "," as a separator. A parameter is a value that a Web command may need in order to operate. Parameters can be simply written at the end of a Web command statement or they can pass to a Web command through a pipe connector.

For example:

```
1   fetch "http://edition.cnn.com/","bbc";
```

This Web command has no flags and two parameters of type string. The command will identify the first parameter as a valid URL and retrieve it. As the second parameter is not a valid URL, the `fetch` command default is to search for the string on Google search and then to retrieve the first returned result.

```
1   fetch -r2 " www.cs.nott.ac.uk/~cah" |
2   getLinks -e;
```

The `fetch` Web command has one flag and a single parameter. The command `getLinks` has only one flag; it retrieves all links in the piped input from `fetch`. The `-r2` flag instructs the `fetch` Web command to recursively retrieve links up to two levels. The '`-e`' flag instructs the `getLinks` Web command to filter the results so that they only include external links.

6.5 Web Commands Input and Output

Each Web command has a standard input and standard output stream. The framework uses the `PipedInputStream` and `PipedOutputStream` Java streams to handle Web command input/output. Data is written as a stream of bytes; however, the framework adds a few Metadata values at the beginning of each item placed on the stream in order to allow better handling of the data.

Web resources have a MIME type, where each type reflects certain attributes or structure of the resource data. Hence, the knowledge of the MIME type offers Web commands the ability to handle each type in a different way.

6.6 Building a Pipeline

Web2Sh supports a UNIX-like piping system to allow end users to build their simple customisation of automation scripts. A Web2Sh pipe-sequence or pipeline is a set of Web commands chained by their standard streams, so that the output (STDOUT) of a command feeds directly as input (STDIN) to the next one (see Figure 6-1).

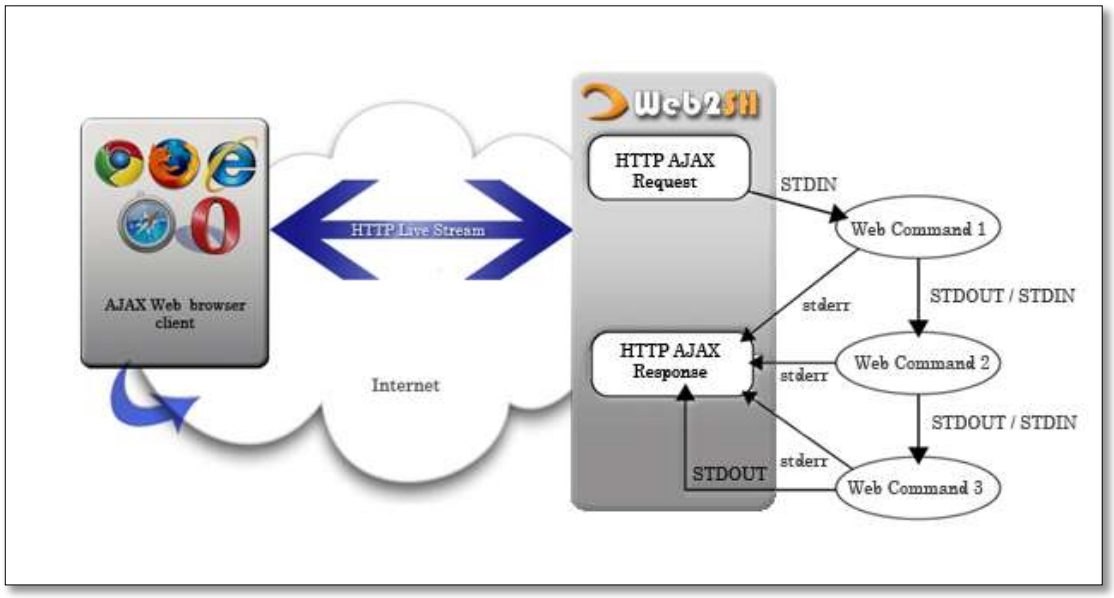

Figure 6-1: Web2Sh pipeline

The Web2Sh framework uses a pipe symbol ‘|’ to connect the standard output of the preceding command with the standard input of the following command. The pipe also always acts as a for-each, so for example if the first command returns a list of links the pipe will arrange that each link is processed through the rest of the pipe sequence.

The following section covers examples of the currently available Web commands covering various general Web task activities. Each command is explained in terms of objectives, flags, and parameters.

6.7 Internet Browsing Commands

This set of commands performs basic Internet browsing tasks, such as retrieving a Web resource and filling and submitting Web forms.

Web2Sh users will need usable Web commands to achieve the automation of various Web common browsing tasks. For example:

- Clicking on hyperlinks.
- Filling and submitting Web forms.
- Extracting Web information.
- Aggregating information from multiple Web pages.

6.7.1 Fetch

The `fetch` command takes one or more Uniform Resource Locators (URL); retrieves the MIME type of the resource the URL refers to, then acts according to that type. The output differs based on the MIME type of the retrieved resource; the default action for all text-based MIME types is either to output the text content as a string to the user’s Web browser or pipe it to the next Web command. For other MIME types, the command will fetch the resource and offer the user a link to download it or pipe the resource as a stream of bytes to the next command.

The most common resource on the Web is `text/html`; a `fetch` command will simply retrieve the page HTML source code content, and display it on the result frame or pass it to the next command for extra processing. Size limitations can be enforced to avoid overloading the system. According to the (Ramachandran 2010) report, the average size transferred over the network per Web page is 320 KB, this usually includes other resources , mainly images, with a size of around 260 KB per page.

Another supported type is `application/pdf`; a simple `fetch` will prompt the user to download the file. If the output was piped to a `find` Web command specifying the title, the PDF will be downloaded to the server and processed to extract its title and return the title to the user. Any downloadable resources such as images will simply prompt the user for a place to save them on the user's machine.

Another type is `text/csv` resources; the user needs to enter the character that separates fields in each row, a comma by default, as a parameter following the URL or as a single parameter if the URL was piped to the command. Often CSV files may have a few lines of irrelevant text, usually a description. Also using flags these lines can be skipped.

For other types of resources, the system offers the user links to download them. If the user uses the `-zip` flag, then the system compresses all retrieved resources to a single zip file on the server for more convenient use.

Web command name	fetch
Synopsis	fetch [flags] [parameters]
General Description	Retrieve a resource from the Web
flags	-rx: Do a recursive fetch, limited to X levels. The recursive flag is active only when the retrieved resource is of text/html type. -i: Retrieve only internal links. i.e., that share the same domain of the original URL -e : Retrieve only external links -zip: Compress all retrieved resources and return an html link to the user, which can be used to download the compressed zip file. -html: Retrieve only resources of type text/html, the default with -r flag. This flag is useful when passing a set of URLs to the fetch command. -csv : Retrieve only resources of type text/csv. -pdf : Retrieve only resources of type application/pdf. -img : Retrieve only resources of super type image.
parameters	Fetch command can take any string as its input parameter. If the string contains a valid URL, the command will retrieve it. Otherwise the first part of the string (up to 10 tokens) is used to perform a Google search and the first result URL is used.
Pipe Input	The Web command accepts input either through a pipe or as parameter list. Input can be any set of valid URLs, strings.
Default Output	For all text types, the system will write any string less than 120k directly to the output stream. For larger resources or non-text types, the system will write a link that refers to the location of the resource on the Web.
MIME types	Fetch command support generally supports all retrievable Web resources. At this point it works best to retrieve all text and image types plus: - application/pdf - application/zip - application/soap+xml

Table 6-1: Fetch Web command summary

Example1:

```
1   fetch "http://www.google.com";
```

The script above will simply retrieve the given URL and display it on the result frame. A flag `-r2` can be used, this indicate that the script should retrieve this page and all pages it has links for, recursively, up to two levels.

Example 2:

```
1   getLinks "http://themiddleclass.org/csv" |
2   fetch -csv |
3   parseCSV -i2 -i3 -h1 |
4   extract "Congressperson (House)";
```

The script above parses the given URL, extracts all links in the page, and fetches only the links that point to CSV files. The `parseCSV` Web command then parses the files specifying to ignore lines two and three, and using line one as the metadata header line. Finally, the extract Web command returns the list of names under the column “Congressperson (House)”.

6.7.2 Submit

The submit Web command takes a page URL, the name (id) of a form in the page and the values the user needs to assign for its fields in order, then submits the form based on its method and action returning the response to the user.

Web command name	submit
Synopsis	submit [flags] “URL” “form id” [parameters]
General Description	Fills and submit a Web form
flags	-http : displays the http request used to submit the form, and the http response.
parameters	First parameter is a valid URL Second parameter is the form name or id followed by a set of form fields’ values in order.
Pipe Input	Both the page URL and the form id have to be written as parameters, while fields’ values can be read from the parameters list or from a pipeline.
Default Output	A string representing the form submission response.
MIME types	Only supports working with text/html resources. Any URL pointing to other MIME types will return an error.

Table 6-2: Submit Web command summary

6.8 HTML Parsing

A set of commands is provided for dealing specifically with the MIME type `text/html`. Most basic HTML parsing functionalities are provided through the use of a java-based Parser for HTML (Oswald 2006).

6.8.1 GetLinks

This Web command takes any HTML structured string as a parameter and extracts all anchor tags from it. It outputs all extracted links to the output stream.

Web command name	getLinks
Synopsis	getLinks [flags] [parameters]
General Description	Parses HTML structured string and extract a list of all anchor tags in it.
flags	-http : output only http links. -ftp : output only ftp links -mail : output only email links
parameters	Takes one parameter, a String representing HTML formatted text
Pipe Input	a String representing HTML formatted text, this command is usually used after a fetch command
Default Output	A set of anchor tags (links)
MIME types	Only supports text/html resources. Any URL pointing to other MIME types will return an error.

Table 6-3: GetLinks Web command summary

6.8.2 GetTags

This Web command parses HTML Web pages and outputs only tag nodes. It can take parameters specifying certain tag names to extract.

Web command name	getTags
Synopsis	getLinks [parameters]
General Description	Parses HTML structured string and extract a list of all tags in it.
flags	none

parameters	Zero or more strings representing valid HTML tag names, no parameters indicate to retrieve all tags in the text.
Pipe Input	a String representing HTML formatted text, this command is usually used after a fetch command
Default Output	A set of HTML tags
MIME types	Only supports only text/html resources. Any URL pointing to other MIME types will return an error.

Table 6-4: GetTags Web command summary

6.8.3 GetTagByID

This command extracts a certain HTML tag by its id or name. It is useful after a `fetch` command to extract a certain part of the Web page for display or extra processing.

Web command name	getTagByID
Synopsis	getTagByID [parameters]
General Description	Parses HTML structured string , searches for a tag with a given id or name attribute
flags	none
parameters	One parameter specifying the id value to search for
Pipe Input	a String representing HTML formatted text, this command is usually used after a fetch command
Default Output	HTML tag
MIME types	Only supports text/html resources. Any URL pointing to other MIME types will return an error.

Table 6-5: GetTagByID Web command summary

6.9 Text-Processing

This set of commands offers useful tools for parsing and manipulating any text-based resources on the Web. It employs the powerful Java regular expression capabilities enabling end users to gain even more flexibility in processing text resources.

6.9.1 Extract

This command extracts a line or a section that matches a given regular expression. It is useful after any command that produce text-based type.

Web command name	extract
Synopsis	Extract [flags] [parameters]
General Description	Parses any string , searches for a line or section with a given regular expression match
flags	-l : Default, Outputs only the line where a given regular expression is matched. -s : Outputs a section where a given regular expression is matched. The user needs to enter a regular expression to specify the start and end of a section as parameters.
parameters	One parameter, it specifies the regular expression to match. With the –s flag the user needs to add another two parameters specifying the section possible matching regular expression start and end.
Pipe Input	a text-based resource
Default Output	Plain text
MIME types	Supports any text resources. Any URL pointing to other MIME types will return an error.

Table 6-6: Extract Web command summary

6.9.2 Sort

This command sorts a piped list of strings alphabetically and writes them to the standard output

Web command name	sort
Synopsis	sort [flags]
General Description	Sorts alphabetically a list of strings
flags	-a : Default, ascending sort -d : descending sort
parameters	none
Pipe Input	List of strings
Default Output	Plain text
MIME types	Supports any text resources. Any URL pointing to other MIME types will return an error.

Table 6-7: Sort Web command summary

6.9.3 Replace

This command processes the piped input, searches it for a given regular expression and replaces it with a given string.

Web command name	replace
Synopsis	replace [flags] [parameters]
General Description	Replace first matched string in piped text with a given string
flags	-a : replace all matched strings
parameters	Takes two parameters, the first is the regular expression to match and replace, the second is the new string.
Pipe Input	List of strings
Default Output	Plain text
MIME types	Supports any text resources. Any URL pointing to other MIME types will return an error.

Table 6-8: Replace Web command summary

6.9.4 Count

This Web command receives a list of items through a pipe, and outputs the number of piped items.

Web command name	count
Synopsis	Count [flags]
General Description	Count the number of piped items
flags	-l: counts the number of lines for a text-based resource -w : count number of words for a text-based resource
parameters	none
Pipe Input	Any list of items
Default Output	Number
MIME types	Supports any resource type. Will simply return the number of resources piped to it.

Table 6-9: Count Web command summary

6.10 Mathematical

6.10.1 Calculate

This Web command is similar to a simple calculator. It takes any simple mathematical equation as a string, parses it and writes the result to the standard output

Web command name	Calculate
Synopsis	calculate [parameters]
General Description	Takes a mathematical equation as a string, parses it and writes the result to the standard output
flags	none
parameters	One parameter representing a mathematical equation as a string.
Pipe Input	none
Default Output	Number

MIME types	none

Table 6-10: Calculate Web command summary

6.11 Running External Programs and scripts

Web2Sh offers a straightforward approach to utilise many other available services over the Web, or even services offered by the platform it runs on. This was tested by running some of the UNIX shell tools.

For example:

```
1   fetch "url" | txt | unix.grep 'Web2Sh';
```

This pipeline will retrieve the given URL, and then extract only the text content eliminating any HTML tags. Finally, the result will be passed to the UNIX `grep` text search utility.

6.12 Error Handling

The first possible error that concerns Web commands is the use of a wrong command name, flags, or parameters. This is not detected by the Parser, as it only knows of the general command syntax rules, i.e. an identifier followed by optional flags followed by optional parameters. If such an error occurs, then it will be detected by the Interpreter. This will terminate the script execution and return an error message to the user.

The second type of possible error is when a Web command tries to retrieve online resources and fails for a group of reasons such as the famous "404 page not found error". In the case of an error, the Web command returns to the user an error message reporting the error.

If a script is to return a set of items as a result, and a few of them failed to be retrieved, then the shell prints out the results feedback to the user's result area and reports the errors that occurred at the beginning.

For Example:

```
1   fetch
2   "http://www.cs.nott.ac.uk/~eoe/test.html"
3   "Essam Eliwa";
```

This script tries to retrieve the first URL, and search Google for the second, extract the first result URL, and retrieve it.

As the first URL refers to a non-existent file, it will generate an error, while the second parameter works fine. The shell will aggregate the error feedback followed by the retrieved Web page and send them to the user's result area.

6.13 Summary

This chapter presented the main Web command as the main building block that end users can use to create simple and sophisticated Web automation tools. Web commands combined with the WSh scripting language provide users with the ability to write, customise, and run scripts that save them both time and effort.

The Web command Syntax was explained and a set of examples was provided. Furthermore, the capability of utilising external tools, such as UNIX shell tools, was introduced. Finally, error handling at the command level was explained.

Chapter Seven: **WEB2SH APPLICATIONS**

7.1 Introduction

Many end users may invest time into designing and writing scripts to automate some or all of the tasks they carry out on the Web. Other users may prefer to do these tasks manually, yet if they find ready-made Web commands to accomplish certain tasks they may very well opt to use them. Moreover, for many programmers the challenge of building such scripts will be a motive in itself, and the fact that time would be saved in the future every time they use the same script, in addition to that the created scripts can be used by other users if it was shared as a Web command.

Web2Sh aims to provide an easy-to-use framework that enables end users to create and execute Web commands. It offers the end-user a command line for automating Web tasks for/on the Web. Having a script that takes some parameters (if needed), executes the required Web activity/task, automates any repetitive work, and displays a customised page with the results of the activity would be very useful to Web end users. The eventual aim is to support the Web as a platform concept.

This chapter presents some of the possible uses of Web2Sh; it demonstrates the usefulness of the framework through the implementation of various possible Web related activities. There are many scenarios where there is a strong need for the development of a specialised tool. The following sections demonstrate some of the observed challenges that Web2Sh can be used to automate.

7.2 Automation Tools for the Research Community

7.2.1 Building Arabic Corpus

Web2sh was used in the process of building an Arabic corpus for research in the Arabic printer character recognition (AbdelRaouf 2010). The corpus is a set of collected words of a given language. The corpus included linguistic and statistical analysis of the Arabic language. Such corpora are used for researches in the field of optical character recognition, speech recognition, automatic translation, and natural language processing.

The role of Web2sh in the process of generating the Arabic corpus was as follows:

- To harvest text from the Web;
- To filter text selected from the Web to include only the Arabic text.

The `charset` Web command was implemented to filter URLs based on the ISO character set used in the resource. For the Arabic text, the command will need to look for the value of "iso-8859-1", "utf-8", or "Windows 1256" in the content attribute of a Meta tag. The `txt` Web command was used to filter out HTML tags leaving only the text content of the page, and then another Web command was created to filter only Arabic characters based on their Unicode representations.

The following web2sh script was applied to make the needed collection and filtering of the text.

```
1  getlinks -r3 "URL" | charset "utf-8" | fetch
2  | txt =>> "arabicText.dat";
```

The `getlinks -r3 "URL"` Web command will make recursive retrieval (up to 3 levels) of Web pages links starting with the given base URL, then `charset` Web command will only output the URLs that have `utf-8` based content. The list of URLs is the passed to the fetch command for retrieval of its contents, and then the `txt` Web command performs the required text filtering. Finally, the result is appended to the file `arabicText.dat`, and a link to download the file will be returned to the end-user when the pipeline finishes.

Each Web command is a Java class that was written and appended to the Web2Sh core commands library. Building such core Web commands would require significant programming skill and some time to build and test. However, once implemented each command can be customised, and commands be connected together using the Web2Sh embedded scripting language. This will enable Web2Sh end users from quickly putting together scripts that automate complicated tasks.

7.2.2 Organizing PDF files

The research process often includes the download and categorization of papers in PDF format. Many of the downloaded papers may have an unrecognized name that does not express their contents. A researcher can upload his collection of PDF files, compressed as a zip file, into Web2Sh, or any Web space that is available online.

A Web command (`pdfRenamer`) was created as part of this study to check each PDF file, it automatically collects data on each file using the Google Scholar service, and then it renames each file with the year followed by the title of the paper. It places the file inside a folder named after the author name. Finally, all files are compressed again and a link is passed to the user in order to download it.

The following web2sh script was applied to achieve this task.

```
1  fetch "zipFileURL" | unzip |
2  pdfRenamer | zip;
```

7.3 Automating Repetitive Web Tasks

As mentioned in Chapter 6, Web2Sh provides end users with an easy alternative (i.e. the Web commands) to automate complex tasks that usually require advanced programming language knowledge. The Web2Sh Web commands can automate both simple and sophisticated tasks that users may attempt on the Web, such as link navigation, form filling and submitting forms, in addition to many other useful commands that enable users to extract specific content from the page and return it to user, or pipe it to other Web2Sh commands for further processing. Users can combine these commands in Web2Sh pipelines to solve complex tasks, using, for instance, a similar model to UNIX pipes where users can chain the result of one command onto another command.

7.3.1 Finding the Top 10 UK Movies on the BBC Web Site

An end-user may create a command to get him the top 10 UK movies from BBC Web site like:

```
1  new top10UkMovies() {
2   fetch http://www.bbc.co.uk/movies/ |
3   find -a "Top UK Movies" |
4   getLinks -n 10;
5  }end;
```

This command will fetch http://www.bbc.co.uk/movies/ page, search for "Top UK Movies," -a flag instruct the command to return all the content after the first occurrence for the string it is searching for, then the getLinks command gets the first 10 links found in the string that was piped to it.

7.3.2 Filtering and Retrieving Links

Web2Sh offers various Web commands to retrieve Web resources and filter them based on given criteria. In the following script, a set of links that has the word "mobile" in its text is extracted from the BBC Web page, using the `find` command filters the links based on a given parameter. The links are then directed one by one to the `fetch` command, which gets the content of the pages, then adds it one by one to standard output file. This process happens when a *Web command* return a list of results and it is piped to another command it will be executed for each item in the list. The final output is the content of the Web pages that its links were retrieved and filtered from the BBC home page.

```
1  getLinks "http://www.bbc.co.uk" | find
2  "Mobile" | fetch;
```

7.4 User Accessibility

Web sites are usually designed for a graphical mode of interaction. Sighted users quickly spot relevant information in Web pages. By contrast, individuals with visual disabilities need to use screen-readers to browse the Web, it is more difficult for them to extract the pieces of data they need directly. Web2Sh offers new approaches for building Web commands according to the users' needs to extract a text representation of the data they need. Web developers and Web researchers can use such tools to improve accessibility. W3C HTML techniques for Web content accessibility guidelines (Chisholm, Vanderheiden et al. 2000) lays out some rules that developers should follow.

Web2Sh can be used to enable Web developers to assess their Web sites accessibility. For example, they can use Web2Sh scripts to follow the links in a Web site and extract a certain value from it to evaluate, in order to improve their site accessibility.

```
1   fetch –r2 "URL" | getImageTags |
2   echo –url-t-alt;
```

This script will make a two level recursive fetch on the given URL, and then for each returned html page it extracts the image tags in the page. Finally, it will print each found image `URL` followed by a tab followed by the value of its `alt` attribute.

7.5 Building Web Mashups

A Mashup is a Web application that combines data from more than one source into a single integrated tool; there is an essential transformation that is taking place on the Web based on information composition through Mashups (Anant 2006).

Web2Sh offers a novel approach for end users to build such Mashups. One of the challenges in the development of Mashups has been the need for usable design tools to facilitate their building by a larger number of end users. As an example, Web2Sh commands could be used to extract information about locations of cinemas from a

Web page and feed that information to another command to obtain a map for each extracted location.

Another set of commands could be used to extract items from various Web pages, such as film reviews, and to build a new page of combined information to be displayed to the user.

For example, a user can retrieve his friend list from Facebook using the `getFbFriends` command then execute a Google search on each friend name. This returns to the user each friend name, followed by the top five links related to him on Google search.

```
1   getFbFriends “user”, “password” | tee $out |
2   google –t5 |
3   echo $out,”\n”,$STDIN;
```

The final `echo $out,”\n”,$STDIN;` Web command instructs the shell to print first the URL in the `$out` followed by a new line then followed by the items (links) in the command input stream.

7.5.1 Aggregate Search Results with Semantic Tags

Web2Sh has a useful set of commands to enable users to extract a list of any HTML tags within the page, such as `getLinks`, `getHeaders`, `getForms`, `getTables`, and `getlists`. Users can combine, using pipes, any of these commands with the `find` Web command, which can search within the returned lists for specific patterns or specific attribute values depending on the flags that users assign to the command.

A useful example is for a user searching for a “Java RMI Tutorial“. This user can simply search for the term on Google, then he wants to check every link he got against one of the social bookmarking sites to see if anyone has found this useful before, and he may also find a more useful description for the various links.

Instead of having multiple browser tabs / windows opened, with repeating copying and pasting of the titles or the links, the following command can do the job and return the results combined on one page.

```
1  Google –n 20 “Java RMI Tutorial“ |
2  getHeaders –h2c “r” | txt | delicious;
```

This command will retrieve the top 20 results for the given search term on Google. Then it will extract the headers where the `<h3 class=”r”>` tag was used (the title of the pages that were indicated by the Google search). Then it will extract the text inside, and then it will pass a list of the search terms to the `delicious` command, which will do the search on the `http://del.icio.us/` Web site and return the results to the user.

7.6 Extracting Data from Web Pages

A Web page that is fetched using Web2Sh can be parsed as a string of characters allowing users to search for any specific patterns on the page to extract data they require. End users may choose to search for specific HTML tags to search within their content or to ignore the HTML tags and search directly within the text; in this case, they may not need to examine the page source HTML code.

Users may also use regular expressions to search for different patterns with many Web commands, e.g. `extract` and `grep`, by using these commands Web2Sh enables end users to search within Web pages content for specific patterns, this includes searching the HTML tags, which can be useful in many cases. For example, more experienced end users, with basic regular expression knowledge, can use these Web commands to search for a pattern that matches all HTML table cells with specific attributes.

A simple example is for extracting data using Web2Sh. If users required retrieving all links on a Web page, and sorting them according to which URL they refer to, they can use a set of commands like:

```
1   fetch bbc.co.uk | getlinks | sort;
```

Another example; If users needed to strip all HTML tags, using the `txt` filter command, and process only the text content of the page to search for a certain string, and then display all lines that this string appeared in :

```
1   fetch www.bbc.co.uk | txt | find "Egypt";
```

Other filter commands are available through Web2Sh such as title, URL, HTML, RSS and css. These filters help users to extract certain parts of the page.

7.7 Composite Search Tasks

End users often need to build composite searches, where the result of one search is the input for another. For example, in academia a researcher may want to search for papers about some topic then get the keywords used in a paper or the list of authors, then use them to build another search, this usually involves opening multiple browser windows and spending time and effort copying and pasting keywords.

To take a more specific example; a user may want to search on the current top movies on IMDB site then find the nearest cinema that displays each movie. The following script will accomplish this task. It will return each movie name followed by the link from Google specifying the nearby cinemas.

```
1   fetch
2   "http://www.imdb.com/boxoffice/?region=uk" |
3    getLinks | find "href=\"/title/" | txt |
4    tee $out |
5    google -t1 " city name" | echo $out $stdin;
```

Such a script demonstrates the usefulness and the effectiveness of using Web2Sh to accomplish complex tasks using minimal lines of script and minimal effort.

7.8 Vendor-Specific Web Commands

Web2Sh provides a set of commands that makes use of various well-known Web applications like major Search engines, answer.com, amazon.com, and Google. Consider a command `define` that takes a term and get its definition and basic data about it using `www.answers.com`, the user may use the following command:

```
1   define "Web 2.0";
```

When the user executes the command, the result page from `www.answers.com` will be returned to him. This simple case of redirection has the following benefits and potential uses:

- It saves the user significant time of browsing to get the required data;
- It offers a faster way to be used on devices like PDAs or mobile phones to send a command;
- It can be implemented as voice commands for speed and ease of use.

7.9 Retrieving Financial and Stock Market Indicators

Users may need to extract specific information from a Web page, for example to extract Stock Market Indicators data from the CNN home page, users can write a WSh script to achieve this. One way of doing it might be:

```
1   Fetch "cnn" | findLi "market-" | txt
```

This command will retrieve the CNN page, and then find `li` tags where the class value has the string `"market-",` extract the text contained by the div tag, which is the name of each indicator followed by its value.

```
<li class="market-1 closed">
    <div class="marketInfo-left"> <!-- left side -->
        <span class="marketName"><a href="http://money.cnn.com/data/markets/dow" target="new">Dow</a></span>
        <span class="marketIndex">13,784.17</span>
    </div>
    <div class="marketNums-right down">
                <span class="closedMarket">Closed</span>
        <div class="percentDiff">
            <span>(<span class="plusMinus">-</span>1.55&#37;)</span>
        </div>
        <div class="numDiff">
            <span><span class="plusMinus">-</span>216.4</span>
        </div>
    </div>
</li>
```

Figure 7-1 excerpt from cnn.com Web page source

Such a script requires the user to be familiar with HTML well enough to examine the page source, as in Figure 7-1, to find the best approach for extracting the required information. Like mashups, this script will return an error when CNN updates its page code. Hence, Web2Sh and its scripting language offers a quick and easy platform for building similar mashups, which enable end users to quickly build and customise scripts as they would need to be tweaked each time a data source is changed.

7.10 Summary

This chapter introduced some of the tested automation script applications on Web2Sh. The usefulness and extensibility of the system is evident, the framework enables the core commands to be extended by developers to add new functionalities. Furthermore, end users can use the WSh language to customise and combine Web commands to achieve innovative automation tools that fit their needs.

Chapters three to seven discussed the model implemented in Web2Sh, focusing on the usefulness of end-user programming for Web automation. The approach followed was to integrate a UNIX shell-like model for the Web as a platform, and to provide a carefully designed set of Web commands that execute both simple and sophisticated Web tasks. In addition, a scripting language focused on enabling end-user Web automation.

The next chapter describes the evaluation of the Web2Sh system and the command line based approach for achieving Web automation. The evaluation is done through an experiment that includes using both Web2Sh and Yahoo pipes systems and gathering participants' feedback through a set of questionnaires.

Chapter Eight: **EVALUATION**

8.1 Introduction

This chapter describes and presents the evaluation of the experiment that was performed to examine the hypothesis of this thesis: the development of an interactive end-user programming UNIX-like model for the Web, as a platform in its own terms, is feasible and useful. The evaluation was carried out as a user trial that involved twenty-four participants. The user trial involved the use of both Yahoo pipes and Web2sh to produce a set of modules/Web commands, with the objective of automating information integration between Google products and Amazon Web services.

Before the main user trial, a pilot of the experiment was carried out with the aim of examining and validating the experiment's workflow and procedure. Based on the feedback from the pilot run, changes were made to the instruction text and to the questions (i.e. the narrative of the user tasks) to make them clearer to the participants and to avoid confusion. One of the main changes was to add to the text more details explaining the concepts of composite Web tasks and Web Automation.

The experiment was carried out over a period of three weeks. It was performed online by providing a detailed guide for the study flow and online links to Yahoo pipes and Web2Sh. Participants were asked to complete four online questionnaires using "surveymonkey.net". Participants were recruited from people who work in the Information Technology field; however, they varied in their detailed job description (i.e. not all of them work as developers). The participants had varying programming experience, which ranged from less than one year to more than four years.

The experiment procedure was published on the Web and volunteer participants were given a link to a Web page that contained the instructions and the workflow of the experiment. Each participant was asked to give their informed consent after a brief explanation of the experiment and the activities they would be required to carry out.

Each user was asked to fulfil five tasks that complemented each other. The overall objective of the five tasks was to integrate data from Google product search and

Amazon Web services in order to create a mashup of a set of dynamic links that contain an associate id of an amazon affiliate. The basic idea of affiliate marketing was explained to participants and they were asked to implement it through five smaller tasks that build on top of each other.

8.2 Specific Hypotheses

The Web2Sh framework and Yahoo pipes system were tested in an online user trial in order to evaluate the following hypotheses:

- H1. The development of an interactive end-user automation framework for the Web, as a platform in its own terms, is useful.
- H2. A Command line approach is easier to learn and use, by end users with programming experience.

8.3 Experiment Design

In order to carry out the experiment, a case study was chosen on the area of affiliate marketing (Hoffman and Novak 2000). The experiment consisted of five tasks, encompassing:

1. Introduction task, which mainly aims at getting the participant started with each system.
2. A task to create an automation tool that uses Google search to retrieve a result set.
3. A task to create an automation tool that uses the Google product search API to return a list of products.
4. A task to create an automation tool that searches the Amazon Web site for a specific product.
5. A task to integrate the tools created in steps 3 and 4 to create a mashup of top returned Google product results, based on a random keyword of the participant's choice. Each result is to be displayed as the title of the product, followed by an affiliate-encoded link to this product on Amazon.

Affiliate marketing is a type of performance-based marketing in which a business rewards one or more affiliates for each visitor or customer brought about by the affiliate's own marketing efforts. One of the pioneers of affiliate marketing is Amazon, which offers an affiliate program when a visitor clicks from an associate's website to Amazon and purchases a product, the associate receives a commission.

Many websites and online businesses generate additional income by becoming marketing affiliates. The main approach is to provide links on your Web site that lead to products or services offered by third parties. Depending on the scheme, an affiliate (i.e. the marketer) earns a commission whenever visitors to the affiliate's website click on a link that leads to a third party's website and/or makes a purchase.

In affiliate marketing, the third party (usually a company that sells a product) is able to identify the referring affiliate by means of an affiliate's id, which is assigned by the third party. The affiliate's id must be encoded in the referring URL. The following is an example of a URL that has an Amazon affiliate's id in it:

```
http://www.amazon.co.uk/?_encoding=UTF8
&tag=3ssam-21&linkCode=sb1&camp=2378&creative=8438
```

The affiliate's id in this case is the value of the tag parameter, which, in this example, is "3ssam-21". The affiliates can encode the URLs for the products they choose to advertise on their sites or it can be automatically generated for them.

Automating the task of generating Amazon's affiliate encoded URLs was chosen as the final task of the case study for the following reasons:

- Affiliate marketing has become a major strategic consideration for all e-commerce companies (Amit, Zott et al. 2002).
- The concept has certain technological complexities that can be made simpler with ready-made and easy to configure Web automation tools.
- Amazon was one of the first online businesses to implement an affiliate marketing program (Hoffman and Novak 2000, Solonchev 2003).
- Amazon offer the ability to generate an affiliate encoded URL as a service accessible through both its Web interface and its Web services.

The users were divided into two groups. The first group (group A) was asked to fulfil the five tasks using Yahoo Pipes, and then use Web2Sh. The second group (group B) was asked to fulfil the same tasks but in the opposite order (i.e. use Web2Sh first, and then using Yahoo Pipes). Each participant was instructed to keep a record of the time it took to complete each task. Detailed instructions were given for the first task to get the participant started, while notes that gradually become more general were given to participants in the remaining tasks.

8.3.1 Task One Summary

The participants were given a list of RSS feeds URLs, and they were instructed to retrieve and display its contents. Then they were asked to filter the results based on a given keyword. Participants were given the names of the Web commands/modules they need to use to implement the tasks and online support was offered to answer any questions they had.

A Web2Sh script that can be used to implement the task:

```
Fetch
"http://sports.espn.go.com/espn/rss/news"
"http://rss.cnn.com/rss/si_topstories.rss"
"http://feeds.feedburner.com/foxsports/RSS/headlines"
"http://www.telegraph.co.uk/sport/othersports/americanfootball/rss"
"http://feeds.bbci.co.uk/sport/0/football/rss.xml?edition=uk"| RssFilter –o "football" "Striker";
```

Another approach can be to save a list of the URLs on a Web page and the following script:

```
getLinks "http://www.web2sh.net/rssFeeds.html" | fetch
| RssFilter –o "football" "Striker";
```

Participants were requested to implement the same task on Yahoo pipes. An example of a Yahoo pipe that was used to implement the task is shown in Figure 8-1.

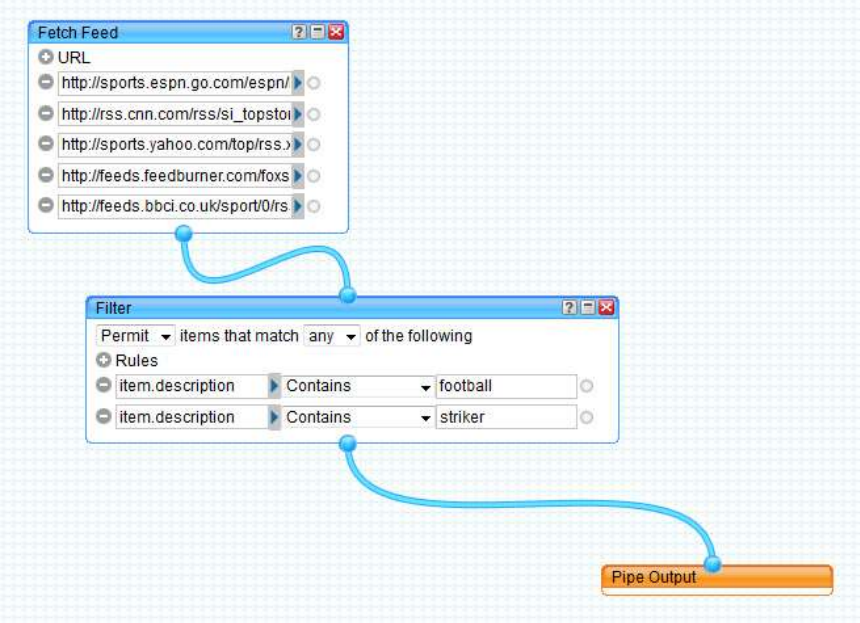

Figure 8-1 Sample solution for task one in Yahoo pipes

8.3.2 Task Two Description

This task was to create a script/module to search the Web using the Google search API and to return the results as RSS feed. Participants were given the names of the Web commands/modules they need to use to implement the tasks and online support was offered to answer any questions they had.

In Web2Sh, this task can be implemented using the `Google` Web Command to retrieve the search results for a given keyword:

```
Google "laptop";
```

As Yahoo pipes does not offer a built-in module for searching Google, the participant needed to construct the URL that needed to be used in order to retrieve the result, and then rename the returned XML tags to enable the result to be displayed correctly on the final output (see Figure 8-2).

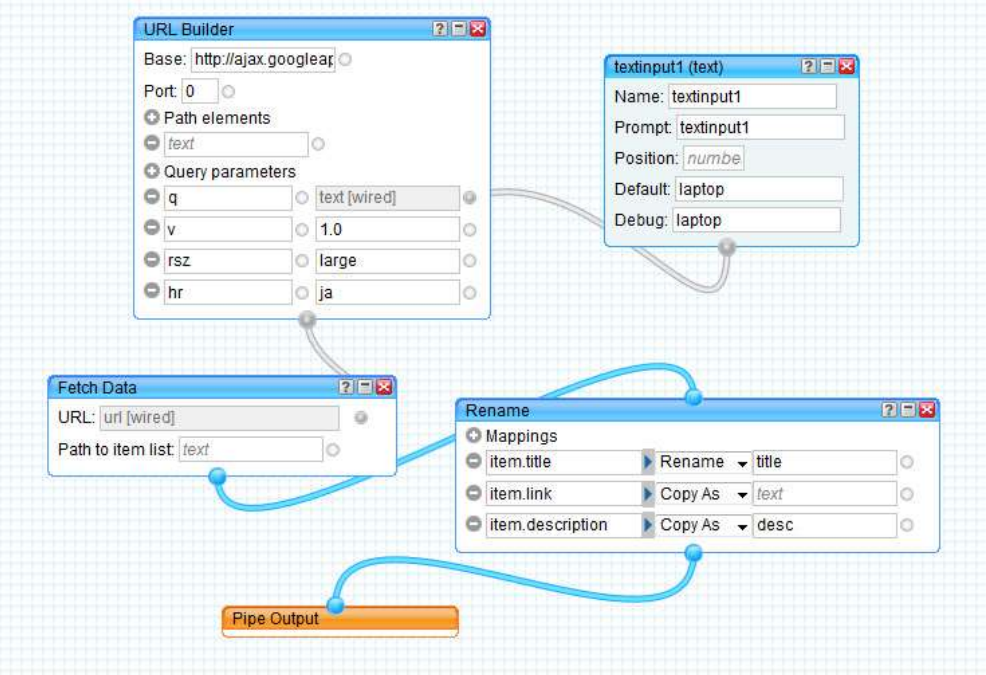

Figure 8-2 Sample solution for task two in Yahoo pipes

8.3.3 Task Three Description

This task was to create a script/module to use Google product search API to return a specified number of results. The participants were requested to acquire an access key to use the Google API. Participants were given the names of the Web commands/modules they needed to use in order to implement the tasks, and online support was offered to answer any questions they had.

In Web2Sh, this task can be implemented using the `product` Web Command, which was created specifically to connect to Google product API to retrieve the search results for a given keyword. The flag `-nX` is used to set a maximum number of returned results:

```
Products -n5 "AIzaSyD0It8nmpgq……" "laptop";
```

The participant could also use the basic `fetch` Web command to retrieve the results, and then then use the `Truncate` Web command to display a specified maximum number of results as in the following script:

```
Fetch
"https://www.googleapis.com/shopping/search/v1/public/products?key=AIzaSyD0It8nmpgqF01UHo7hJoJzFDKbFfIboUM&country=US&q=laptop&alt=atom" | Truncate –n5;
```

Similar to task two, Yahoo pipes does not offer a built in module for using the Google product API. The participant needed to construct the required URL to be used to retrieve the result, and then rename the returned XML tags to enable the result to be displayed correctly on the final output (see Figure 8-3).

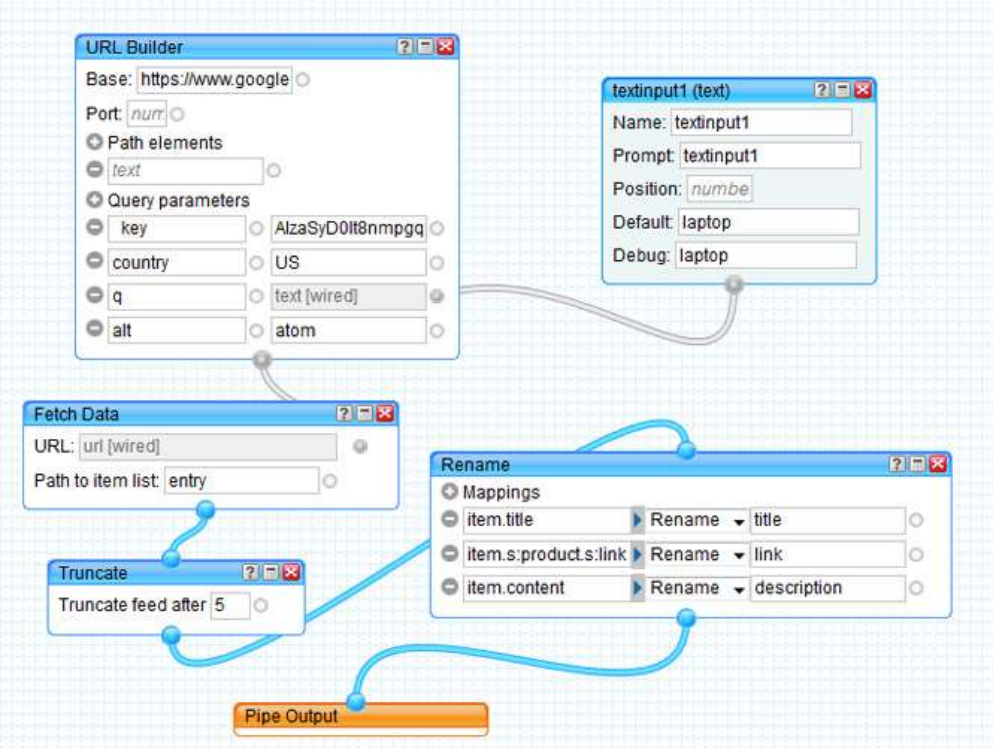

Figure 8-3 Sample solution for task three in Yahoo pipes

8.3.4 Task Four Description

This task was to create a script/module to use the Amazon product search API to search for a keyword and retrieve list of Amazon products details. The participants were offered instruction to create their own Amazon API key and associate tag (affiliate id). As obtaining these values could take a significant time with some

participants, they were offered the option to contact the researcher to obtain a working key and tag pair if they were not interested in creating their own.

In Web2Sh, this task can be implemented using the `AmazonSearch` and `AmazonViewer` Web Commands, which was created specifically to connect to Google product API to retrieve the search results for a given keyword. The flag `–nX` is used to set a maximum number of returned results.

```
AmazonSearch “AWS_ACCESS_KEY_ID” “AWS_SECRET_KEY”
 “Associate_ID” “Keyword” | AmazonViewer;
```

It is also possible to use the `echo` Web command to pass one or more keywords to the `AmazonSearch` Web command as in the following script.

```
echo "Keyword" |
AmazonSearch “AWS_ACCESS_KEY_ID” “AWS_SECRET_KEY”
 “Associate_ID” “Keyword” | AmazonViewer;
```

In Yahoo pipes, the participants were instructed to search for other users’ shared modules and use them as starting point to investigate the use of the YQL module (see Figure 8-4).

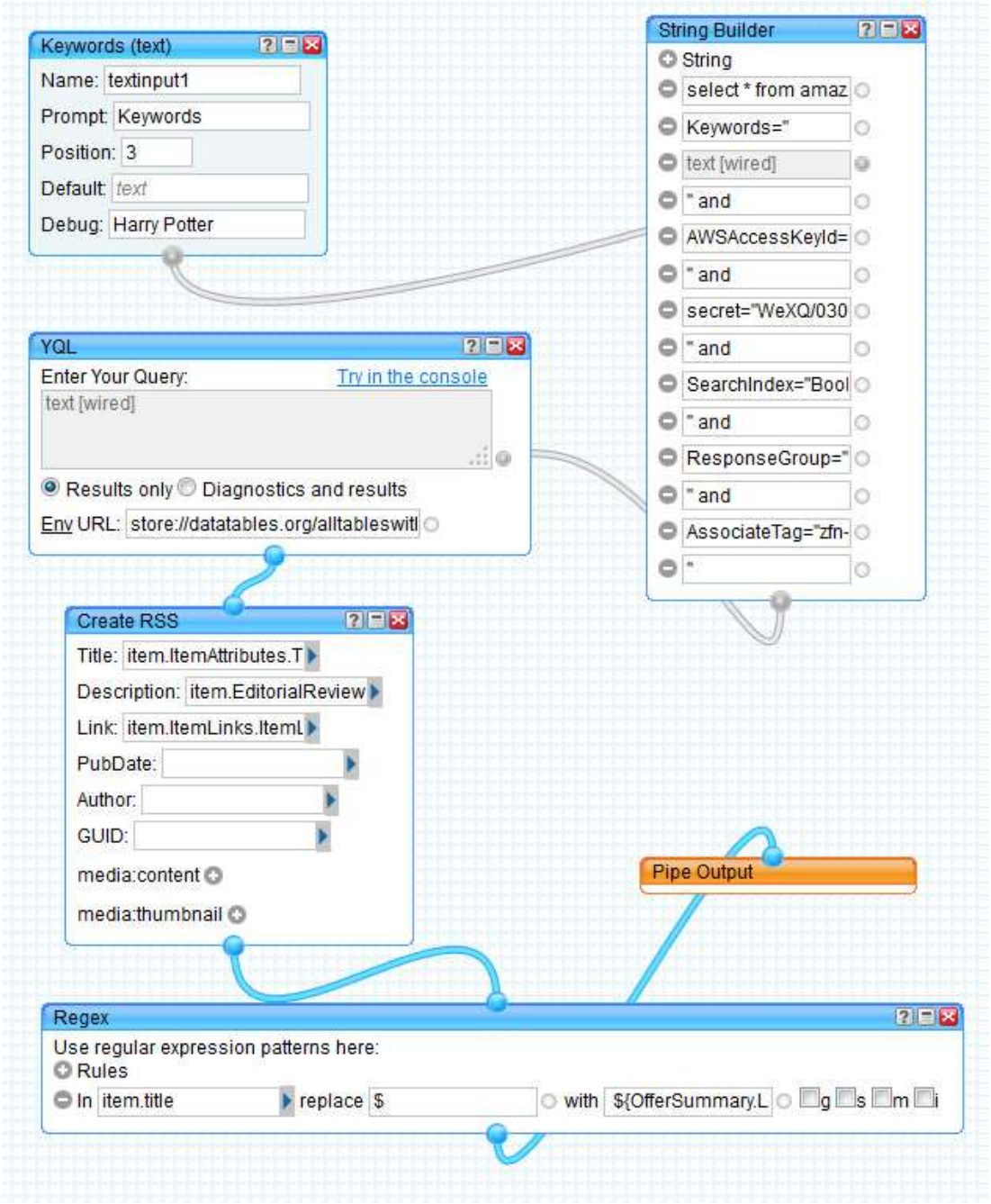

Figure 8-4 Sample solution for task four in Yahoo pipes

8.3.5 Task Five Description

This task was to integrate the output of task three and four in order to use a Google product search to retrieve a set of products based on a given keyword, then to generate a list that have Amazon links to these products, its price and title. Each link should have an associate id encoded in the URL.

In Web2Sh, this task can be implemented as in the following script:

```
products -n5 "GOOGLE_API_KEY""laptop" | getrsstitle |
AmazonSearch "AWS_ACCESS_KEY_ID" "AWS_SECRET_KEY"
"Associate_ID"|
AmazonViewer;
```

In Yahoo pipes, the participants were instructed to connect the modules they created in task three and task four. An example of one solution is shown in Figure 8-5.

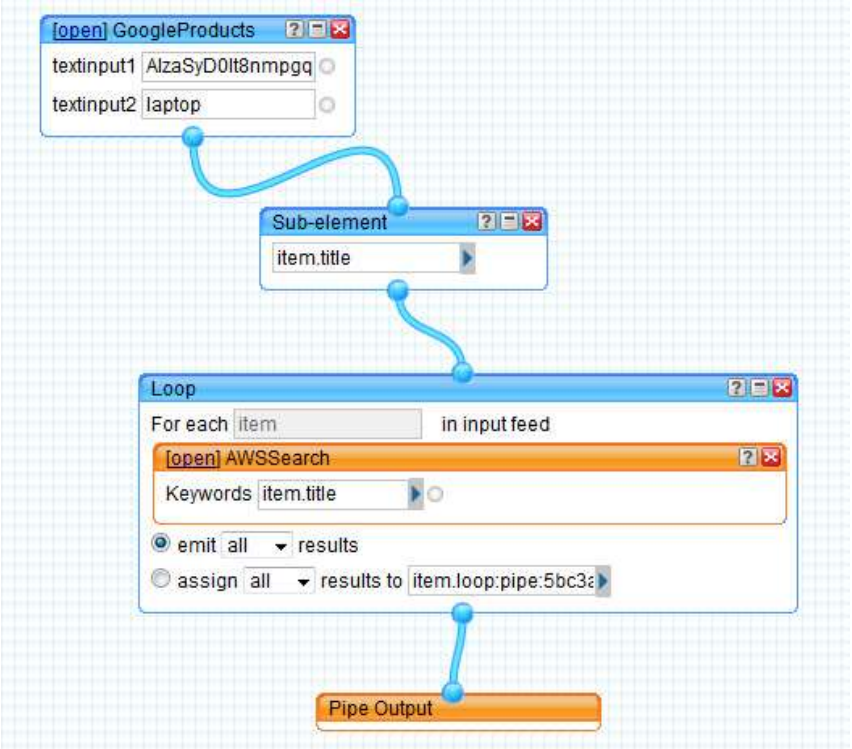

Figure 8-5 Sample solution for task five in Yahoo pipes

8.4 Participants' Background

All participants were postgraduates with varying programming experience (different years of experience and using different programming languages). Thirteen of the twenty-four participants had programming experience of more than four years. Ten participants had experience between one and three years while only one participant had less than one year as shown in Figure 8-6. When asked about how confident they were in their programming skills, seven answered very confident; fourteen answered confident; two answered neutral; and one answered less confident as shown in Figure 8-7.

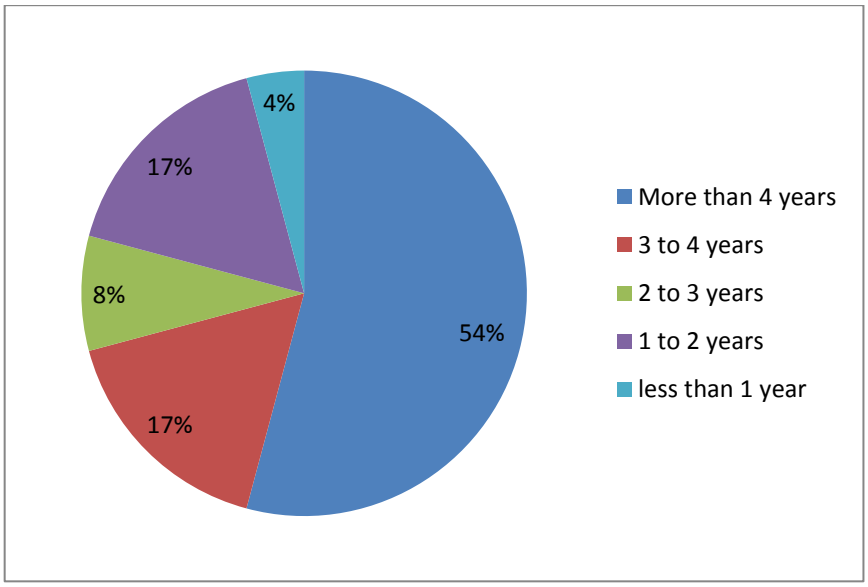

Figure 8-6 Participants' programming experience

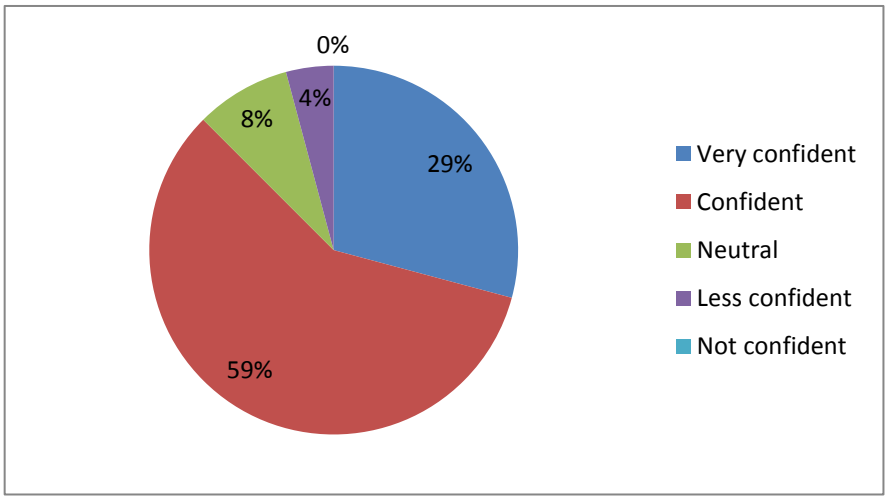

Figure 8-7 Participants' self-evaluation of their programming skills

All participants have used various programming languages; the most used languages were Java, HTML, C++, C, C#, JavaScript, and XML. When asked about their preferred language, Java was the most preferred one, by seven users, as shown in Figure 8-8.

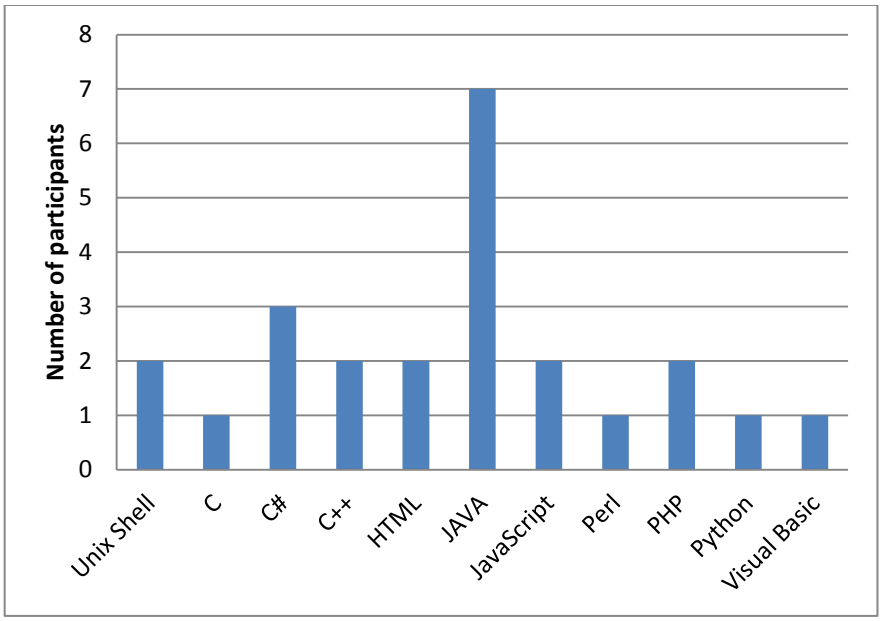

Figure 8-8 Participants' preferred languages

All participants use the Web to conduct a variety of tasks; they vary in terms of the time they spend on the Web as demonstrated in Figure 8-9. Seventeen participants said they execute composite Web tasks from a few times a month to more than once a day as shown in Figure 8-10, and seven participants answered that they never execute composite Web tasks repeatedly. A composite Web task was defined as a task such that accomplishing it involves interacting with multiple Web resources.

In answer to a given question, twenty-three participants agreed on the given sentence that *an easy to use approach to automate Web tasks is useful*, and one participant was neutral. The overall positive feedback of the participants using both Web2Sh and Yahoo pipes (Figure 8-13 and Figure 8-16) indicates that even for the users who do not require repetitive execution of composite Web tasks, they still considered Web automation useful for them.

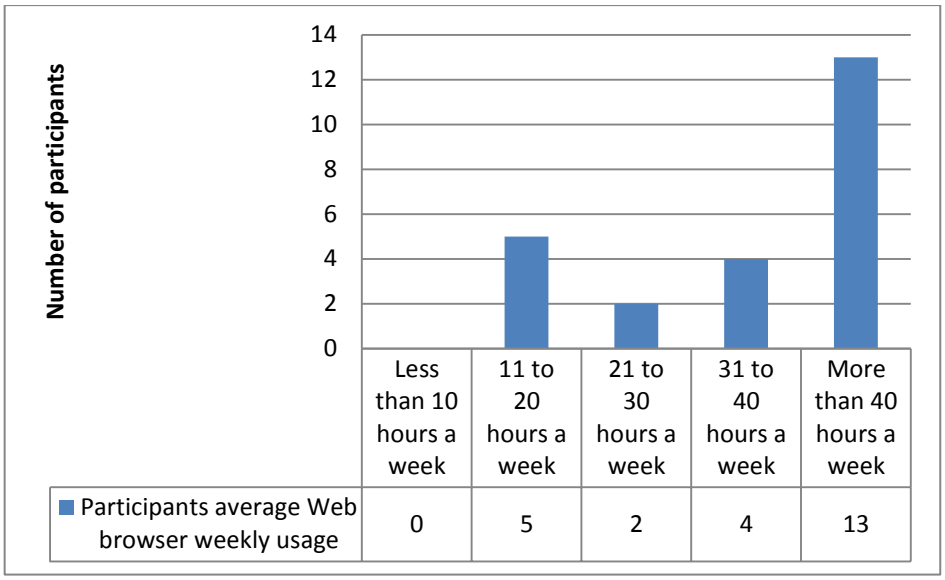

Figure 8-9 Average number of hours using a Web browser weekly

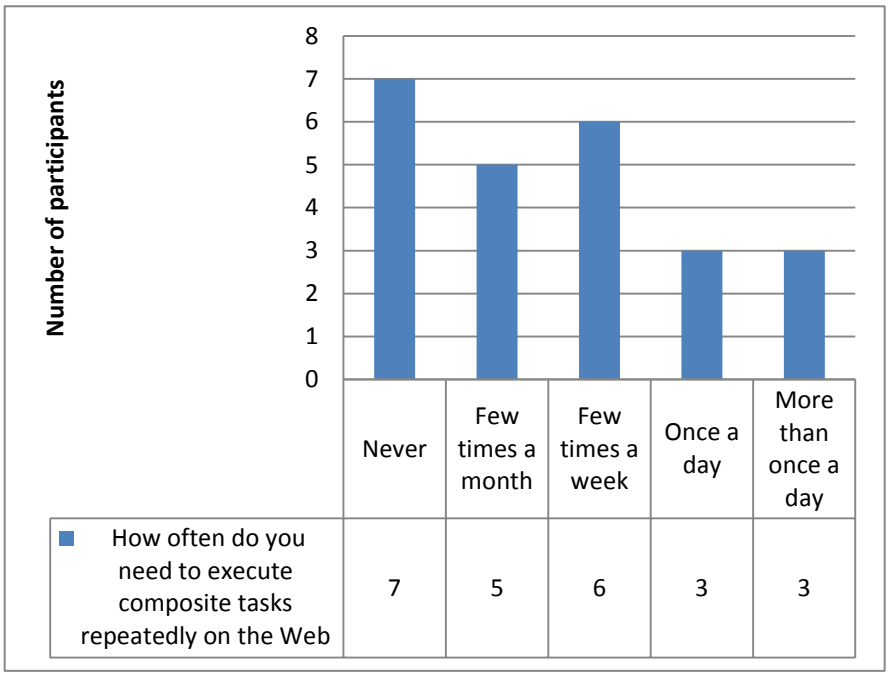

Figure 8-10 Frequency of executing repetitive composite Web tasks

Only one participant specified that he used a tool (WebZip[23]) to automate the task of retrieving and saving Web content. However, seven participants answered that they

[23] http://download.cnet.com/WebZIP/3000-2377_4-10011036.html

have built programs to automate Web tasks. Examples of the tasks they specified are as follows:

- An online marketing tool, using the Facebook Query Language (FQL) and Facebook APIs, to get the participants' own friends list on Facebook in order to send them a private message or post on their wall.
- An application, using VB.NET, to search multiple Web resources for jobs according to a given criteria in the term of the companies and skills of the job seeker.
- RSS news feeds aggregator using PHP, MySQL and JavaScript.
- A C++ application to extract and save Arabic words from various Web sources.
- Using JavaScript and Google charts API to draw a graph using data retrieved from the Web.

The level of skill of the participants and their Web experience makes them very suitable candidates for the experiment. Moreover, the fact that most of them did not use Web automation tools before means that there is no prior bias to any system before starting the experiment.

8.5 Data Analysis

8.5.1 Qualitative Findings

In order to investigate which approach was more favourable to the participants, all participants were asked to specify which approach they preferred using for Web task automation and why. They were offered the choice between:

- Command line approach (as in Web2Sh)
- Graphical (GUI) approach (as in Yahoo pipes)

Two thirds of the participants chose the “Command line approach” as their preferred approach. The answers of the participants regarding why they chose a specific system indicate the following observations:

The command line approach was found to be more preferable and easier to master for users who had programming experience. However, this is can be affected by the type of activities they practise on a daily basis. Most participants who selected We2Sh were found to believe that command line interfaces are easier for them to use, as they are used to it in their work. For example, the following quotes where extracted from participants answers:

- Participant A102: “command line is more preferable to me as it can implement many tasks easily.”
- Participant B103: “command line easier, maybe this is because that I got used to it during my daily tasks”.
- Participant A113: “I love scripting and this is what I found in web2sh”

Most of the participants who selected the GUI approach said that they found it more user-friendly and more entertaining to work with. For some participants they found it was easier to learn. For example, the following quotes where extracted from participants answers:

- Participant A121: “I felt it was more friendly”
- Participant B111: “More user friendly”
- Participant B102: “I prefer using GUI interface as it was easier to know all the commands and their attributes. GUI is usually easy to debug. in addition it is easy to make pipes and use it once more”

Some of the users' comments indicated the need for better documentation, more informative errors and more usable debugging methods in both systems. The following quotes were extracted from participants' answers commenting on Web2Sh:

- Participant A120: "Task 5 needs more clarification as for me it is the first time to address the issue of Web automation. So, I would be happy if the task is described in more detail and the Web commands were better described"
- Participant B104: "The difficulty came from a lack of information regarding the usage and parameters of the commands"
- Participant B104 also commented: "Overall Web2Sh can be seen as a mashup between "Curl for Dummies with a dash of command line". It is quick and easy, very intuitive for a UNIX user. It has potential"
- Participant A113: "Having a live demo would be more likely to let me understand more what and how to do it. Regardless of that I would recommend using it but after including better documentation"

When asked about suggestions to increase the usefulness of the Web2Sh system many useful ideas was presented that will be considered in the future work. Many participants asked for the removing of the need to use a "`;`" at the end of pipeline. Many users indicated the need for better messages and some other interface related features. The following quotes where extracted from participants answers commenting on Web2Sh:

- Participant A120: "Using Web 2.0 technologies (ontologies such as RDF and SPARQL) will make web2sh a more intelligent tool in mapping and linking the different content from different pages"
- Participant A120 also suggested: "Spreading the idea in scientific and developers communities will be so beneficial. A set of tools that allow the

creation of a mash-up that integrate the reported fixes of a specific bug in one page from multiple resources would be very useful"

- Participant B103: "If a command (or set of piped commands) failed then it would be nice to have more informative error messages"
- Participant B107: "The manual is not very clear. More specific example for each command can make the system more useful."

When asked to comment about Yahoo pipes system, the participants' answers varied between who found it easier and fun to use and who struggled to use it. Some of the participants' comments were:

- Participant A102: "Dealing with Yahoo pipes is not an easy process which needs a lot of trials. I finished two tasks with the help of the researcher, but failed to finish the third task and decided not to complete the remaining tasks because it is very hard tool to use"
- Participant A113: "task 1 and 2 was easy to do, while tasks 3, 4, and 5 I wasn't capable of completing them successfully so I need to have a demo in order to be able to do it"
- Participant A105: "I had fun using it in the first two tasks"

Participants also suggested more detailed documentation and better debugging capabilities as participant B102 commented, "A better documentation and a good debug system to track the errors would be useful". Participant B105 also said, "Better documentation and notes are needed". Both participants preferred the Yahoo pipes approach for achieving Web automation.

Participant B115 was one of the few who finished most of the tasks on Yahoo pipes. He reported a good feedback about his experience doing all the tasks. The feedback indicates that having the ability to integrate various Web automation tools

approaches, such as including JavaScript code in Yahoo pipes, would enable better and faster utilisation of Web resources.

- Participant B115: "Task one was easy. Task two and three were relatively easy, but I did not know at the time that Yahoo pipes use the name "item" as a magic word to refer to the parsed items. Task four took too long and required a lot of learning. The most difficult part was to generate Amazon signature in the URL I was frustrated because while Yahoo pipes are straightforward to use, I could not for example add a simple JavaScript that would call a Google crypto API to generate the required signature. Task five looks easy but I had problems in achieving it, although I think I'm few steps away from doing it correctly."

8.5.2 Quantitative Findings

An analysis of the time taken to implement the assigned tasks revealed that Web2Sh was faster to implement all the tasks in comparison to Yahoo pipes (for both groups: A and B). Figure 8-11 shows the mean time taken by participants to complete each task. Task 4 was the most difficult to do and took most of the time in both systems.

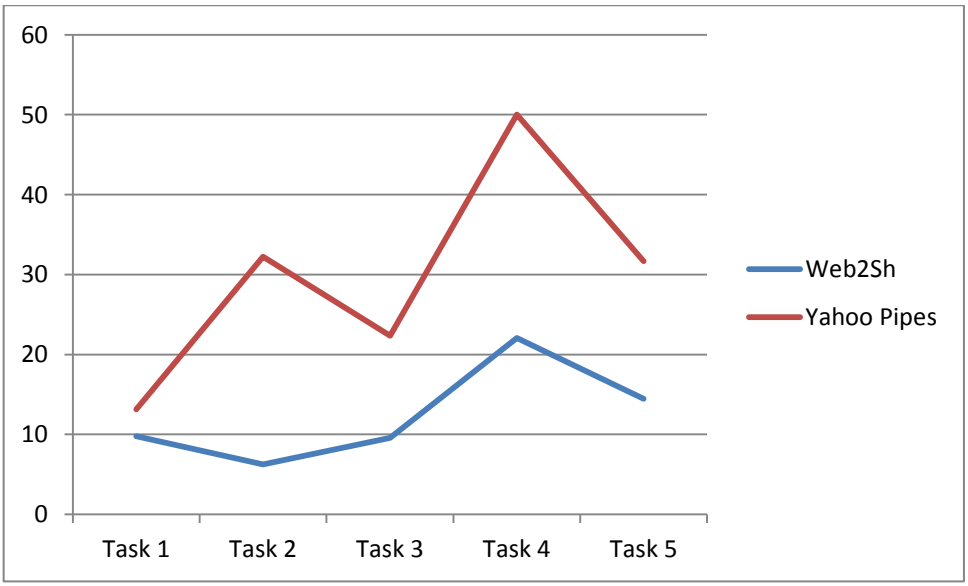

Figure 8-11 Mean value for time taken per task

The analysis of the number of completed tasks for each participant in both systems revealed that ten participants completed only one out of the five tasks on Yahoo pipes, and that the majority of the participants completed less than four tasks. Participants were instructed to check other user-shared modules that may assist them to accomplish their tasks, however only a few of them managed to make good use of other users shared modules. This can be attributed to two reasons: (1) the existence of a large number of user-created modules, which made the participants get a bit lost when trying to search for useful ones to use from them; and (2) many of those modules are out of date, rendering them dysfunctional and useless.

In Web2Sh, thirteen participants completed all the five tasks; all participants except one completed more than two tasks. The nature of the task-specific Web commands combined with the WSh scripting language offered by Web2Sh allowed the participants to complete the tasks in much less time and made them more successful in creating their customised automation scripts. It is important for Web2Sh, once it is made available to a larger community, to put a regular maintenance mechanism in place that would continuously validate any user-shared scripts/tools (see Figure 8-12).

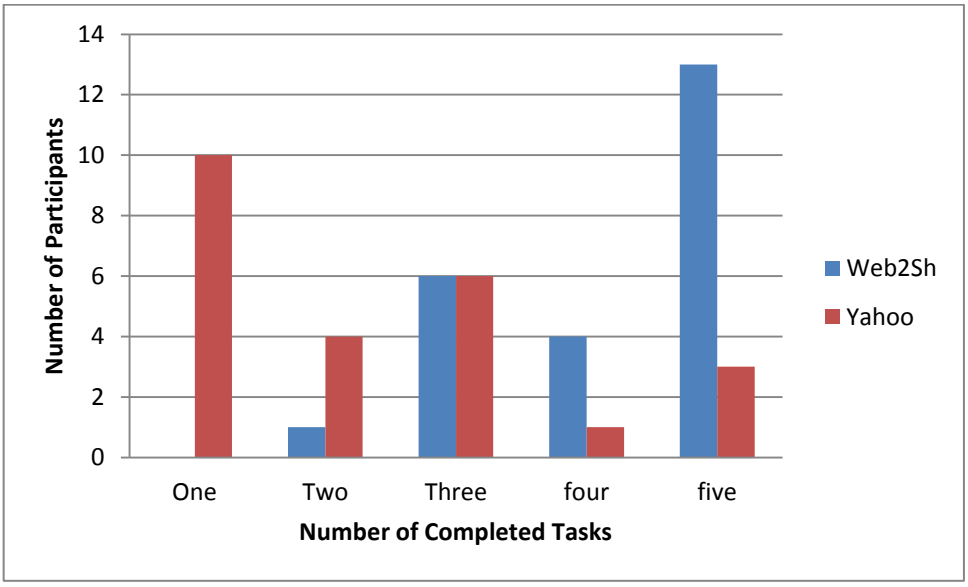

Figure 8-12 Number of tasks completed by all participants

The analysis of the final participant's general rating for both systems revealed that the average Web2Sh rating was four, while the average rating for Yahoo pipes was three, where one represents the lowest rating and five represents the highest rating

(see Figure 8-13). The majority of the participants agreed that Web automation in general is useful for them (see Figure 8-16). Two thirds of the participants answered that they prefer the Web2Sh approach to achieve Web Automation. It was observed that a number of factors might justify liking one approach over the other, including: personal preferences, type of daily activities, and prior experience. For example, six of the seven users who practised programming using a UNIX shell scripting language preferred using Web2Sh, other users who don't practise programming much or tend to use a Visual development environment (e.g. visual studio) mostly preferred Yahoo pipes (see Figure 8-14).

Another factor that seemed to affect the participants' choice of their preferred approach, is how familiar they are with the tasks. The participants had very little or no information about Affiliate marketing, so there was a learning experience they gained when using the first system based on their assigned group. Figure 8-15 shows that for "group A" participants who started the experiment by using the Yahoo pipes system ended to prefer Web2Sh. While for "group B" the participant were divided in half between the two systems.

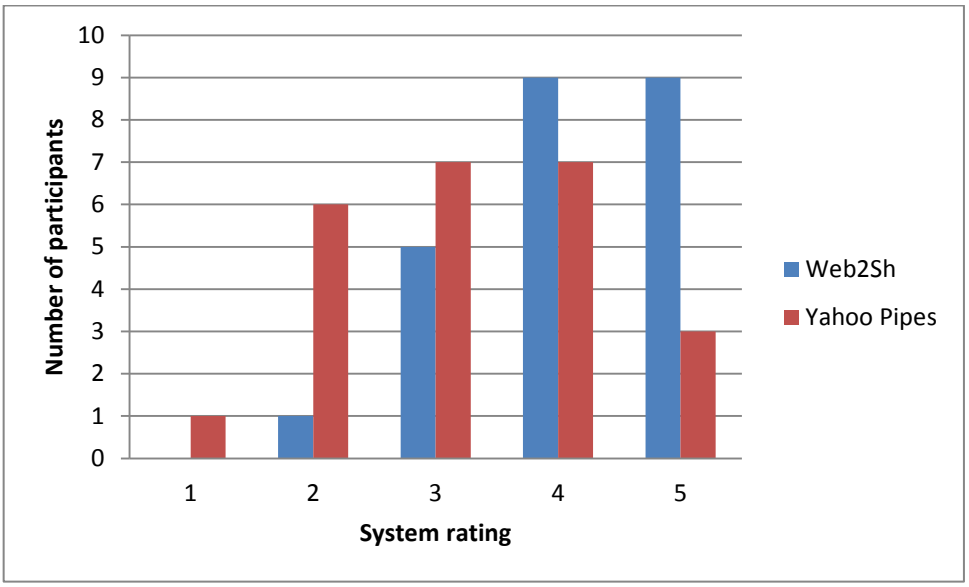

Figure 8-13 Final participants overall rating

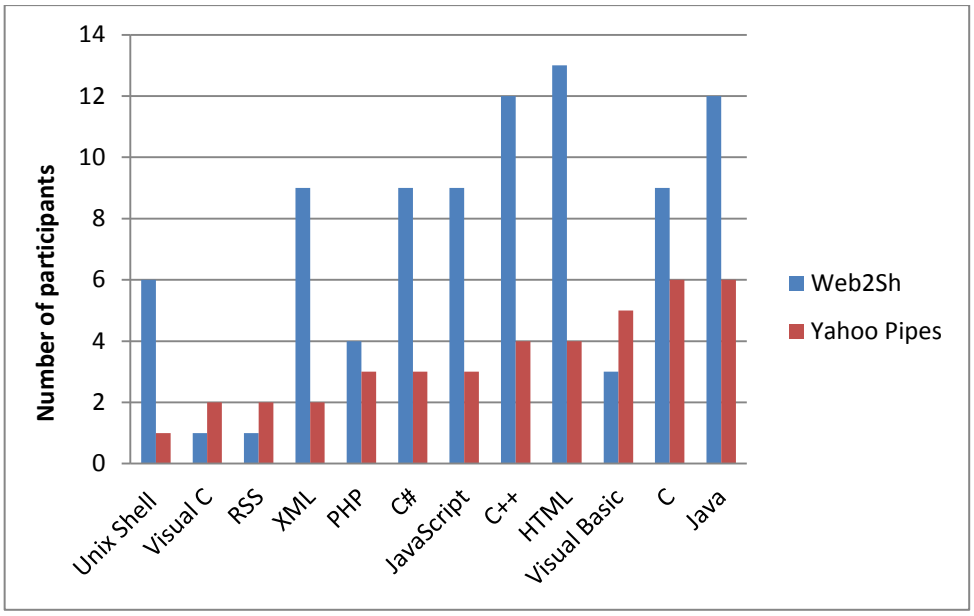

Figure 8-14 Preferred system by each language users

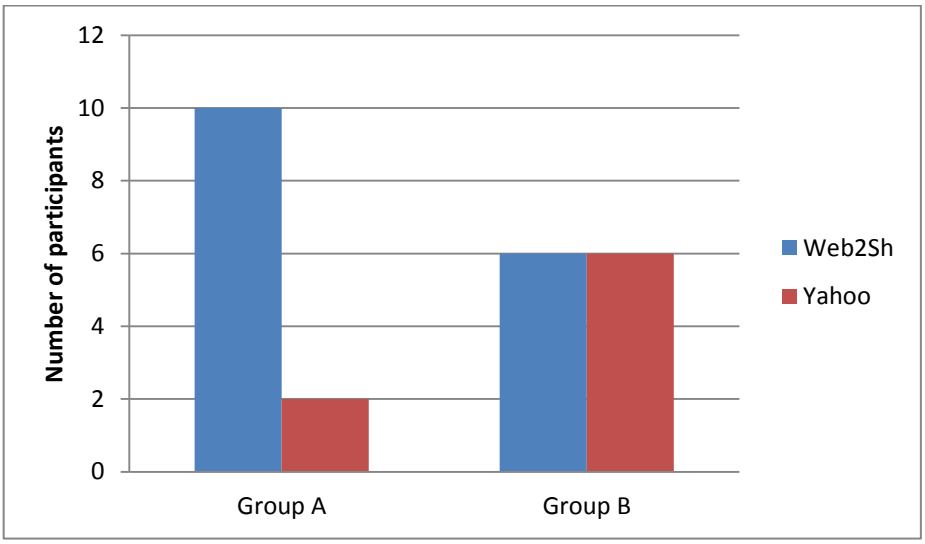

Figure 8-15 Preferred approach by group

A total of eighteen participants strongly agreed with the statement "An easy to use approach to automate Web tasks is useful", five selected "somewhat agree" and one was neutral. When asked whether you agree or disagree about the "Web2Sh is useful" statement, eight participants selected "strongly agree", eleven selected "somewhat agree", and five were neutral. None selected "Somewhat Disagree" or "Strongly Disagree". When asked whether they agree or disagree about the "Yahoo pipes is useful" statement, five participants selected "strongly agreed", twelve

selected “somewhat agree”, five were neutral, and two selected “somewhat disagree” as illustrated in Figure 8-16.

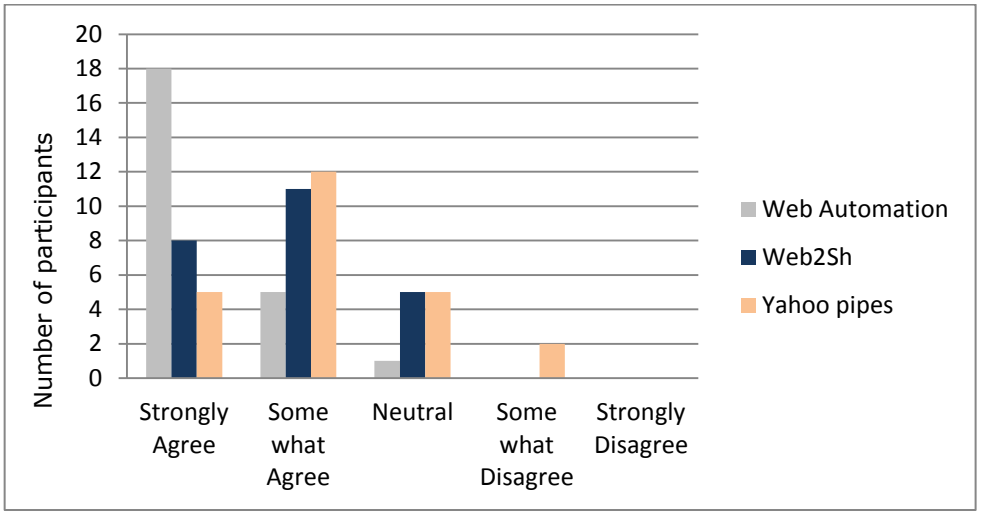

Figure 8-16 A system to automate Web tasks is useful

8.6 Summary of Findings

Of the twenty-four participants, all but one agreed that a Web automation system is useful, one participant was neutral and none disagreed. The answers were slightly different when asked about the usefulness of each tested system (i.e. Web2Sh and Yahoo pipes) as shown in Figure 8-16. This sample shows the development of an interactive end-user system for Web is useful for end users with programming background. However, qualitative feedback indicates that Web commands of more personal interest for each user would be much more useful. This was most clearly demonstrated when participant B104 (Software Developer) commented, “The particular commands were of limited personal interest and thus have low usefulness. The approach when used to generate commands of more personal interest would yield results of much more usefulness”. Participant A111 (a system administrator with UNIX shell experience) requested to include more UNIX familiar commands.

Participant A104 (postgraduate student) requested tools to interact with Wikipedia. Participant A120 (postgraduate student) commented about the Web2Sh system saying that “The idea is very interesting for broad range of users and I think that people who depend mostly on their smart phones will be the greatest beneficiaries of such system”

These responses show that the general idea of a Web automation system would be useful for a wide range of end users. However, it will be preferred to create a domain-specific language for each group with shared interests. This would require surveying the tasks they require the most and what would be the best naming strategy to make the Web commands names more familiar and easy to use for them.

A major drawback in Yahoo pipes was that the majority of the user-shared modules were not functional, usually due to a change in the structure or availability of the Web resources they were based on. The large number of non-functional modules makes it harder for a user to utilise tools shared by other users. The number of non-functional modules in the Yahoo pipes system is an indication of a general drawback when using any mashup or automation tool, which is that it will stop working as soon as any change occurs in its data sources.

This problem is intensified when users share their Web automation tools while not maintaining them. This was clear in Yahoo pipes. While many users share useful modules, they are hard to find due to the numerous number shared modules and the fact that many work only for a limited time span. Many of the modules on Yahoo pipes fail to return any output due to this problem. For example, a book titled "Mashup Case Studies with Yahoo! Pipes" (Loton 2008) used a pipe created by another user to perform an Amazon items search[24], however due to a change in Amazon's Web service WSDL this module is not working anymore. This means that all other modules (as in the mentioned book) that use it also fail to work. This pipe was created in 2007 and the book was printed on 2008. Other newer modules offer valid solutions that worked with the new structure; however locating them is not easy.

For example, in the evaluation reported in Chapter 8, the fourth task was a modified version of the Amazon marketing case study in (Loton 2008). The task was only completed by four participants in Yahoo pipes, even though many other participants tried to complete them (for time ranging from thirty minutes up to three hours). The participants were instructed to explore modules created and shared by other users to solve this task, as some of the shared modules can facilitate the task implementation. However, only a few participants managed to locate and use those modules due to

[24] http://pipes.yahoo.com/pipes/pipe.info?_id=aAHDUL193BG8LFnqiXrL0A

the long list shared modules (mostly non-functional). Even when a user executes a search using a keyword such as “Amazon API search” the list of returned modules is too long making it much harder to locate useful and working ones. Many of the modules was working when they were created initially, yet as the Amazon search Web service was changed, all older modules stopped working.

This indicates a necessity to validate any public shared modules and downgrade them to a non-working status once they fail to complete successfully. Such functionality can be automated on Web2Sh by running a batch to test users’ shared scripts and flagging a script if an error was returned upon execution. Constant maintenance of the core system Web commands is also essential, in addition, enabling an advanced search and rating system for users shared scripts must be considered, as it will be very useful for end users.

The support for more graphical elements in the Web2Sh command line interface to enhance its usability should be considered in the future. For example, some participants liked the idea of having a list of modules displayed in the left side pane in Yahoo pipes. Other participants found it confusing. Hence, adding any graphical elements should be examined based on the targeted users, and users should be able to enable and disable them based on their preferences.

To conclude, the following points summarise the findings of the evaluation:

- The majority of participants agreed that the notion of Web automation is useful to them.
- The command line approach to building and automating Web tasks is favourable to the majority of participants.
- As a development environment, participants were found to need less time to get tasks done using the command line approach in comparison with the Graphical (GUI) approach.
- The majority of users agreed that Web2Sh was more useful to them than Yahoo Pipes.

- The majority of users agreed that Web2Sh was easier to use than Yahoo Pipes
- Web2Sh was faster to implement the task on, in comparison to Yahoo pipes.
- More participants finished a larger number of tasks on Web2Sh in comparison to Yahoo pipes.

8.7 Summary

The evaluation, presented in this chapter, examined the usefulness of Web2Sh via a user study. Twenty-four participants completed the experiment, all of which had some degree of programming experience. Eighteen of the participants strongly agreed with the statement "An easy to use approach to automate Web tasks is useful", five selected "somewhat agree" and one was neutral. When asked whether you agree or disagree about the "Web2Sh is useful" statement, eight participants selected "strongly agree", eleven selected "somewhat agree", and five were neutral. None selected "Somewhat Disagree" or "Strongly Disagree". When asked whether they agree or disagree about the "Yahoo pipes is useful" statement, five participants selected "strongly agreed", twelve selected "somewhat agree", five were neutral, and two selected "somewhat disagree" as illustrated in Figure 8-16.

This chapter covered the evaluation of this thesis hypothesis, which is the development of an interactive end-user programming UNIX-like model for the Web, as a platform in its own terms, is feasible and useful. Section 8.5 demonstrates the findings of the evaluation, which supports the thesis hypotheses.

The next chapter describes the foundations and arguments for the design and implementation decision throughout the development of Web2Sh framework, lays out the conclusions of this research, and suggests future work.

Chapter Nine: **CONCLUSION AND FUTURE WORK**

9.1 Introduction

This research introduced Web2Sh (Web 2.0 Shell): a framework encompassing the WSh (Web Shell) scripting language, a Web command line interface, and a repository of Web commands. The framework demonstrated the emerging need for tools suitable for automating and customising the user experience on the Web. The aim of Web2Sh is to provide the required infrastructure for Web automation in order to give Web users more control over their Web experience while moving towards the vision of the Web as a platform.

This thesis is based on the hypothesis that the development of an interactive end-user programming UNIX-like model for the Web, as a platform in its own terms, is feasible and useful. The presented Web2Sh model builds on the experiences gained from the study of other existing systems and tools of Web automation. It also uses modern enabling Web technologies to take the concept of Web automation a step forward.

An important design goal of the Web2Sh framework is to find an appropriate balance between efficiency and usefulness on the one hand and ease of use on the other. It was also important to integrate into the framework a Web-focused language (WSh) to support the customisation and integration of Web commands to serve variety of domains. The current implementation of the Web2Sh system offers a set of Web commands that stand out as a proof-of-concept. This set of commands serves to demonstrate the usefulness and the extensibility of the Web2Sh model.

The implementation of the *polymorphic MIME types*, *(*section *6.3*), and other design features allow for basic decision-making at the level of Web commands and the Interpreter. Compared to other tools offered by other vendors and researchers, Web2Sh has a significant advantage because of its unique approach that combines the following elements:

- The use of the browser as a thin client, and using the server-side to run and execute Web commands.

- The provision of a practical Web-oriented scripting language, that utilise the flexible UNIX piping concept
- The provision of a library of Web automation tools in the form of Web commands.
- A command line interface that enables fast implementation of WSh scripts.

This unique approach enables users to harness the power of Java and server-side processing through a simple Web-based interface using a simple UNIX-like scripting language.

The evaluation experiment, described in chapter 8, examined the usefulness of Web2Sh. All of the participants had previous programming experience. Eighteen of the participants strongly agreed with the statement "An easy-to-use approach to automate Web tasks is useful", five selected "somewhat agree" and one was neutral (Figure 8-16). These responses combined with the open-ended comments of the participants, indicated that end users, who have programming background, find an easy to use system for Web automation to be useful for them. It also indicated that creating a set of focused domain-specified commands to match users from specific domains would be even more usefulness.

Most of the tools that are currently available do not offer as comprehensive an approach as Web2Sh. Most of the other tools are too specific to a certain domain, with no adequate ability to extend their coverage to more domains (as in the case of Accessmonkey). Moreover, they are totally based on the client side and tightly coupled with the browser (as in the cases of Chickenfoot, Greasemonkey, Mozilla Ubiquity, and iMacros).

The only tool that offers a server side-based system is Yahoo Pipes, yet it is very dependent on its graphical user interface for the creation of automation tools, which renders it less efficient and more time consuming for experienced end users (those who have programming background). While Yahoo pipes modules are efficient when dealing with RSS feeds, it is much more difficult to access and extract information from other types of Web resources, such as HTML pages.

To conclude, the features of Web2Sh demonstrate a closer relationship to the nature of resources available on the Web, offers many useful functions with a minimum amount of coding required, and simplify the specifications that a user needs to learn to start using it. The following Web2Sh characteristics give it many advantages as a framework intended for Web automation:

- A command line Web based interface that can be used from any Web browser.
- A powerful back-end, harnessing the potential of the Java language.
- Easy to write Web commands that make use of most of the available Web resource types.
- Web command's Polymorphic MIME types feature enabling different execution paths based on the MIME type of the resource passed to it.
- A UNIX-like scripting language offering a simple grammar for complex tasks when compared to other tools.
- Users can access a set of powerful UNIX commands such as the `grep` command.
- An extensible framework backend allowing the fast and easy creation of sophisticated, domain-specific, and customisable Web commands.

9.2 The Need for a New Scripting Language

At the early stages of this work, other available scripting languages (e.g. JavaScript) were considered for the implementation of Web2Sh framework. However, the study of other tools (reviewed in section 2.3) showed that the approach of customising an existing language such as JavaScript in Accessmonkey (Jeffrey and Richard 2007) was less user-friendly and less effective compared to the tools that have their own focused scripting language such as in Chickenfoot (Bolin, Webber et al. 2005b).

The initial requirements for the framework, its objectives, and the profile of the potential users implied the need for implementing a focused task-oriented scripting language that would enable easy and effective Web automation. The need for the ability to connect Web commands to achieve sophisticated tasks combined with the software tools philosophy and its success in the UNIX shell scripting languages model, that presented a good model for a platform-oriented automation, implied that a similar model would be feasible and useful for end-user task automation on the Web as a platform.

The implemented WSh scripting language is targeted at the Web. It merges the GUI nature of the Web with the power of the command line interface. It also enables the use of Web commands in pipelines to achieve complex tasks as demonstrated in Chapters 5, 7, and 8.

9.3 Is Yahoo Pipes Enough?

Yahoo is investing a lot of effort to improve its Pipes product. It has gained considerable popularity since its launch in 2007. So far, the focus of Yahoo Pipes has been in dealing with RSS-based feeds. It provides a minimal set of modules to retrieve other types of resources such as regular HTML based Web pages, offering limited functionality to parse them once they have been retrieved.

Moreover, syndicate feeds used by Yahoo Pipes usually provide only excerpts of the page. The user needs to click on the title of any item to get the whole content. This means that in many cases Yahoo Pipes can only access the parts that are available in the feed.

Tim O'Reilly stated that Yahoo Pipes was a first step towards creating a “programmable Web for everyone” (O'Reilly 2007b), arguing that Yahoo Pipes could be considered as a good start based on the Unix shell piping approach for creating a Mashup.

The Yahoo Pipes GUI interface makes it easy for novice users to create and run basic modules. Yet, the graphical user interface and the limitation of the other MIME types hinder the Yahoo Pipes tool. For casual programmers, advanced users, and

programmers it would be easier, faster, and more efficient to have a text-based editor where users can write code rather than using the ‘drag and drop’ GUI interface.

Many Web sites still do not provide syndicate feeds. For example, the IMDB web site offers a list of Top 250 movies as voted by their users[25] but it would be very difficult to deal with this list in Yahoo Pipes, as there is no available feed for it yet.

Furthermore, there is a study showing that a command line-based interface, which utilises a natural language is usually easier to learn, faster to use, and more effective than GUI based interfaces (Lovis, Chapko et al. 2001). Another study indicated that combining the power of character-based user interfaces (CUI) with some graphical component support makes such systems more user friendly (Kressin, Berger et al. 1997).

Another study shows that graphic-based interfaces have no significant advantage over text-based interfaces for expert users, although novice users prefer the former. For novice users there is a learning transfer effect from text-based to graphical user interface. The mapping between the user interface and tasks is important to be reflected in user-friendly aspects (Chen, DDS et al. 2007).

The Web is an ideal platform to employ a combination of graphical and text-based interfaces. The idea of implementing a UNIX-like shell for the Web, for creating automation tools and customised information retrieval, is very useful and effective. It provides the ability to use a command line interface implemented as a simple HTML page. This will integrate the GUI nature of HTML and the power of CUI command line. With the rise of reach internet applications (RIA), many useful GUI components can be added to such an interface to offer support for more novice end users.

Tim O'Reilly wrote in Unix Power Tools in 1993:

> “Most of the non-graphical utility programs that have been running under UNIX since the beginning share the same user interface. It is a minimal interface, to be sure, but one that allows programs to be strung together in pipelines to do any job that no single stand-alone program could do. Now, most operating systems, including modern UNIX and Linux systems, have

[25] http://www.imdb.com/chart/top

> graphical interfaces that are powerful and easy to use, yet none of them is as powerful or exciting to use as classic pipes and filters offered by UNIX shells" (O'Reilly 2007b).

This indicates the need for a framework that includes a character user interface (CUI), which can provide similar functionality to Yahoo Pipes. It would be desirable for a framework to provide the support for a graphical interface or make use of already available interfaces, such as Yahoo Pipes, to be integrated in the framework.

Using a text-based command line is fast and easy to use when creating sophisticated scripts is required. The framework introduced in this research (Web2Sh) aims to provide this interface for end users. Most Web2Sh commands can produce an RSS feed by adding a pipe to the RSS filter command at the end of any pipeline. The resulting feed can then be called from Yahoo! Pipes or any other similar tool as any other data source. This way Yahoo Pipes can act as a GUI interface for many Web commands offered by Web2Sh.

9.4 End-user Web Automation

This research introduced and focused on the idea of "*end-user Web Automation*", which is defined in this thesis as *the study of theoretical and practical techniques for applying an end-user programming model to enable the automation of Web tasks, activities, and interactions*. The term *Web automation* in this context refers to the notion of enabling the creation of effective tools to achieve the automation of various Web tasks. End-user Web automation is a promising and novel research field that allows and adopts the vision of the Web as a platform. The study of this field connects various computer science fields.

The research also introduced Web2Sh, a framework encompassing the new WSh scripting language, a Web command line interface, and a repository of Web commands. It demonstrated the emerging need for tools suitable for automating and customising the user experience on the Web. Web2Sh provides the required infrastructure for Web automation to give end users more control over their Web experience while moving towards the vision of the Web as a platform.

The study discussed the design and implementation of the Web2Sh framework based on a simple scripting language (WSh) that facilitates writing automation scripts. Web2Sh aims to provide a basic set of commands in different domains. Users have the power to create, use, and share their own commands. WSh also enables the extension and customisation of Web commands.

In fact, Web2Sh introduces a new approach for creating and sharing Web automation tools among end users through the development and re-use of Web commands. Thus, it can evolve into a powerful framework that encompasses a large set of commands in different application areas.

9.5 Meeting the Objectives

This section revisits the specific objectives set out in the first chapter of this thesis:

29. Investigate a new model for end-user Web automation that would offer an easy to use approach to automate Web tasks.

30. Design, implement, and evaluate a new simple and effective scripting language intended for the development of Web automation tools.

31. Design, implement, and evaluate a Web-based framework to enable end users to create, run, and share Web automation tools.

32. Design, implement, and evaluate a repository of Web commands as a reusable, efficient, and customisable Web automation tool.

The next four sub-sections address each of these objectives linking them to the work achieved in this thesis.

9.5.1 New Model for End-user Web Automation

This thesis presented a novel model that combined end-user programming concepts with Web standards and enabling technologies. The model presented can be used to enable the automation of Web tasks, activities, and interactions. The thesis defines this area of research as *end-user Web automation.*

The model introduced built on the five following design elements:

- UNIX shell scripting, see section 2.4;
- SPHINX for creating personal, site-specific Web crawlers, see section 2.3.1;
- YubNub simple command line for the Web, see section 2.3.2;
- The software tools philosophy, see section 5.3;
- The shift towards the "Web as a platform" vision.

The model approach was to apply the software tools philosophy to the Web, using UNIX shell scripting as a proven successful model. The idea of site-specific and customisable Web crawlers in the SPHINX framework is reflected in the introduced concept of the Web command, at the same time considering the software tools philosophy. The YubNub application, simple as it is, inspired the idea of a simple command line for the Web. In order for this model to work, an important enabling technology was Asynchronous JavaScript and XML (AJAX), which allowed the communication through HTTP streams between the command line interface and Web2Sh backend server.

The Web2Sh framework implemented the introduced model to show its feasibility and usefulness as illustrated in Chapters 4-7. Chapter 8 described the evaluation of the Web2Sh framework that was carried out through an experiment in which end users used both Web2Sh and Yahoo pipes systems. The majority of the participants confirmed that the Web2Sh approach was easy to use for them; only one participant out of the twenty-four participants answered that the system was somewhat difficult to use.

9.5.2 WSh, a Scripting Language for Web Automation

All investigated systems, which aimed at achieving a degree of Web automation, have based their approach on the sole use of GUI-based programming, or have used an existing Web scripting language, most commonly JavaScript. The arguments presented in this thesis in section 2.3 and the evaluation in Chapter 8 demonstrate the desirability and usefulness of implementing a focused scripting language with the target of enabling easy and effective Web automation.

The success and power of the shell-scripting model in UNIX platforms inspired the design of WSh. However, WSh extended this into the dimension of Web automation. Hence, WSh offers a set of unique features, which enable it to provide extensive and sophisticated tools to achieve its goal of Web automation.

One of the aims of WSh design was to enable the creation of Web commands that are 'smart' enough to act in a logical way based on the MIME type, context, and content of the resource it process. In order for a smart Web command to provide certain functionality to its users, it must be able to detect the resource MIME type, the context within the pipeline, and attempt to examine the content itself when needed. It then determines what actions to execute. WSh design supports the creation of intelligent Web commands more easily. In addition, it supports the extension of core Web commands without the need to alter the framework.

WSh design takes into account the variety of Web formats, standards, and technologies. It implements a design whereby any Web resource's MIME type can be processed by the framework. According to this design (named *Polymorphic MIME types*), each Web command may support different functionalities for resources of different types and varying contents. The context of the Web command may also affect the actions it takes.

Chapter 5 presented the knowledge and experience gained by designing and implementing the WSh scripting language. It also described the key enabling computer science concepts and technologies that facilitated the development of the WSh scripting language.

The design of the WSh language makes the best of the new Web enabling technologies and tools. By providing a Web-based command line interface, Web2Sh users can simply create and use Web commands using a JavaScript-enabled Web browser.

WSh contributes to the research of end-user programming and programming language processors. Although it is not the focus of this research, it was necessary to investigate these research topics in order to gain the knowledge required to design and implement a scripting language that can be useful, effective, simple, and powerful.

9.5.3 Web2Sh, a Web Automation Framework

Web2Sh contributes to the research of interactive end-user Web automation by suggesting a new approach of automating Web tasks, activities, and interactions. This approach is both simple and effective. As the development of Web automation tools in general is a complex task for the majority of end users, Web2Sh can be of great benefit to end users with little or no programming experience (casual programmers) who want to automate their Web interactions simply, effectively, and rapidly.

Web2Sh is also well suited for users with a UNIX shell background, as the presented Web shell language (WSh) was inspired by the UNIX platform. Web2Sh also offers access to those users to some of the very powerful tools of the UNIX platform such as `sed` and `grep`. This is achieved by implementing a UNIX tools wrapper that acts as a Web command, passes the input from Web2Sh to the appropriate UNIX tool, and then returns the output back to Web2Sh. This approach brings the power of these tools directly to any Web-enabled device that has a Web browser supporting JavaScript. The evaluation presented in chapter 8 highlighted that out of seven participants, who practised programming using UNIX shell scripting, six preferred the Web2Sh approach in comparison to Yahoo pipes.

An additional target group of potential users is researchers. Web2Sh provides a set of tools that enable researchers to save time and effort by enabling them to automate some of the common research tasks, such as citing online references, processing `pdf` files, collecting raw data from the Web, and checking for word synonyms.

Web2Sh utilises the user-generated content approach by offering its users the possibility to create and share new commands. This approach allows the Web commands repository to grow further. It also enables all its users to utilise and share their experiences through the creation of new commands, which can bring about many new innovative Web commands that may have not been foreseen when developing the core Web commands.

9.5.4 Repository of Web Commands

This work introduced the *Web command* as a tool created with a specific goal to provide a solution for automating simple or sophisticated tasks in various domains related to end users' Web interactions. As discussed in Chapter 6, Web2Sh framework offers a repository of core Web commands with various functionalities as a proof of concept.

The generic approach of the Web2Sh framework, combined with the WSh scripting language, allows for connecting any set of Web commands using pipes. This provides general solutions for a larger set of problems in order to meet the Web users' needs. The design of the Web2Sh framework also allows for the easy extension of the core Web commands. Thereby, the developers can implement any functionality they require as a Java class and then plug it into the framework.

Participants' feedback after using both Web2Sh and Yahoo pipes, described in chapter 8, indicates that providing general Web commands, that is commands that can fit into any domain, is essential and enable more flexibility. However, for each specific domain of users a more focused and advanced Web commands need to be provided in order to offer easy and quick retrieval of data that can be of interest to them from specific Web sources. Hence, different version of WSh language and Web commands can be customised as a domain-specific language for each group of users based on their needs.

9.6 Future Work

The work presented in this thesis is a solid foundation for future research in this novel research topic. Various possible directions can be researched to continue this work. This section highlights the most interesting ones from the perspective of end-user Web automation.

Future work is envisaged as encompassing four directions. First, it involves conducting a study to examine the usability of the proposed scripting language, command line interface, and Web command repository for specific domains such as librarians. This should lead to extension of the set of Web commands to target domain-specific users. Secondly, additional functionality is to be added to the

command line interface to make it easier to use by non-expert users. Finally, there is an ambitious aim also to investigate the use of a p2p model for users to share Web commands and distribute the workload among peers' computers.

The most prioritised continuation of this work is the extension of the core Web commands repository to include:

- Web commands that can manipulate and process all MIME types;
- Further investigation regarding how to apply the knowledge from other Web research domains, such as information retrieval, to create sophisticated domain specific Web commands.
- The investigation of creating domain-specific versions of the WSh language to target different users' domains that can benefit the most of such system, e.g. researchers, e-marketing agents, librarians, and system administrators.
- Investigating the possibility and possible benefits of extending client side Web commands and enable them to be piped to other Web commands.

The performance of the system is an important issue; the current framework implements concurrency for running Web commands in parallel and the current Web2Sh framework can be hosted on multiple servers (server cluster) in order to handle an increasing numbers of users. However, a further investigation is needed in order to enable the use of distributed processing techniques on the Web command level, i.e. distributing the execution of a pipeline among several machines.

Furthermore, there is a need to address the security of the communication between the client interface and the server. This should be easily implemented by installing a public key certificate to the Apache Web server hosting the framework. However, the possibilities it opens up are immense, as the user may even use Web2Sh infrastructure to administer the server. Wrappers can be implemented to provide such Web commands that are related to the administration of the hosting server and even other systems residing on the same server. An investigation of the security and

usefulness of such approach could introduce new dimensions to the usage of the Web2Sh framework.

Another interesting investigation is to exploit the p2p approach to share and run the Web commands. This may be of great benefit to the system performance as the processing of the scripts can be divided among multiple machines that would host a shared copy of each Web command.

Further research could also aim to increase the usefulness of the end-user programming approach for enabling Web automation. Further investigations might be conducted in order to discover if an extension is required to the WSh scripting language, and implement such an extension if needed.

In the field of Human Computer Interaction (HCI), user experiments are needed to test the usability of the system. It can also include a usability study comparing the use of Web2Sh as a command line based interface approach with other GUI based systems such as Yahoo Pipes. Such a study could also investigate adding dynamic features to the command line interface using a JavaScript library to enable a more user-friendly environment for the process of editing scripts.

Possible integration of the command line approach with the GUI model as proposed by (Kressin, Berger et al. 1997) can be also considered. The proposed method is to enhance the user-friendliness of the command line-driven programs by having a correspondence with a series of GUI objects. This approach would be beneficial in empowering even more novice users to utilise the Web2Sh framework while preserving the power and speed of the command line approach.

The evaluation study in chapter 8 indicates that even for the expert users one third of the participants preferred to use the GUI as in Yahoo pipes. The integration of Yahoo pipes commands and Web2Sh is easily achieved through the passing of RSS feeds from one System to the other. Hence, end users can have the choice to use a Web2Sh command through calling it from Yahoo pipes, as long as it produces an RSS feed, and vice versa.

9.7 Closing Remarks

This research investigated the feasibility and usefulness of developing an end-user programming UNIX-like model for the Web that would enable the creation of more effective and accessible Web automation tools in the form of Web commands. This involved the development of a new scripting language dedicated to the goal of Web automation.

No other reported systems have followed this model before. Other notable systems have only achieved a partial level of Web automation. They also exhibited a number of drawbacks that were highlighted in Chapter 2.

An innovative design of the Web2Sh Framework was presented, allowing for the possibility of further future extensions. The use of the user-generated content model with the power of the WSh language enables an extension of the Web command repository by writing and sharing new automation scripts that glue together existing commands. The core commands can also be easily extended by other developers, since the design of Web2Sh follows the practices of OOP and design patterns to allow for such extension without affecting the core of the framework.

This thesis presents a novel model that solves the problem of the lack of easily customisable automation tools for the Web end users. It provides a general solution that utilises various types of Web resources, and works with all modern browsers. It has shown the feasibility of the proposed approach and the usefulness of the various applications areas and scenarios that can be implemented to automate various users' Web interactions, tasks, and activities.

BIBLIOGRAPHY

AbdelRaouf, A., et al. (2010). "Building a Multi-Modal Arabic Corpus (MMAC)." The International Journal of Document Analysis and Recognition (IJDAR) 13(4): 285-302.

Alan, S. (1986). "Encapsulation and inheritance in object-oriented programming languages." SIGPLAN Not. 21(11): 38-45.

Alexa. (2012). "Alexa Top 500 Global Sites." from http://www.alexa.com/topsites.

Alexander, C., S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King and S. Angel (1977). A Pattern Language. New York, Oxford University Press.

Amit, R., C. Zott and E.-A. Center (2002). "Value drivers of e-commerce business models." Creating value: Winners in the new business environment: 15-47.

Anant, J. (2006). Enterprise information mashups: integrating information, simply. Proceedings of the 32nd international conference on Very large data bases, Seoul, Korea, VLDB Endowment

Anderson, N. (2006). "Tim Berners-Lee on Web 2.0: "nobody even knows what it means"." from http://arstechnica.com/news.ars/post/20060901-7650.html.

Anderson, N. (2007). "Tim Berners-Lee on Web 2.0: "nobody even knows what it means"." Retrieved 29 January 2011, from http://arstechnica.com/news.ars/post/20060901-7650.html.

Ankolekar, A., M. Krötzsch, T. Tran and D. Vrandecic (2008). "The two cultures: Mashing up Web 2.0 and the Semantic Web." Web Semantics: Science, Services and Agents on the World Wide Web 6(1): 70-75.

Appel, A. W. (1998). Modern compiler implementation in Java, Cambridge University Press.

Aquino, J. (2005). "yubnub." Retrieved 30 May 2011, from http://yubnub.org/.

Aquino, J. (2005). "YubNub, A Social Command line for the web." Retrieved 1 November 2005, from http://jonaquino.blogspot.com/2005/06/yubnub-my-entry-for-rails-day-24-hour.html.

Armbrust, M., A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica and M. Zaharia (2010). "A view of cloud computing." Commun. ACM 53(4): 50-58.

Atsushi, S. and K. Yoshiyuki (1998). Internet scrapbook: automating Web browsing tasks by demonstration. San Francisco, California, United States, ACM Press

Beard, C. (2008). "Introducing Ubiquity." Retrieved 20 March 2011, from https://mozillalabs.com/blog/2008/08/introducing-ubiquity/.

Beged-Dov, G., D. Brickley, R. Dornfest, I. Davis, L. Dodds, J. Eisenzopf, D. Galbraith, R. V. Guha, K. MacLeod, E. Miller, A. Swartz and E. v. d. Vlist. (2000). "RDF Site Summary (RSS) 1.0." Retrieved 15 March 2008, from http://web.resource.org/rss/1.0/spec.

Bergman, M. K. (2001). "The deep Web: Surfacing the hidden value." The Journal of Electronic Publishing 7(1).

Berners-Lee, T. (1989). "Information Management: A Proposal." Retrieved 26 June 2008, from http://www.w3.org/History/1989/proposal.html.

Berners-Lee, T. (1998, 14 October 1998). "Semantic Web Road map." Retrieved 20 July 2008, from http://www.w3.org/DesignIssues/Semantic.html.

Berners-Lee, T. (2002a). Foreword in: Spinning the Semantic Web: Bringing the World Wide Web to Its Full Potential, {The MIT Press}.

Berners-Lee, T. (2004). "How It All Started." Retrieved 26 june 2008, 2008, from http://www.w3.org/2004/Talks/w3c10-HowItAllStarted/?n=1.

Berners-Lee, T., W. Hall, J. A. Hendler, K. O'Hara, N. Shadbolt and D. J. Weitzner (2006). "A framework for web science." Found. Trends Web Sci. 1(1): 1-130.

Berners-Lee, T. and E. Miller (2002b). "The Semantic Web lifts off." ERCIM News, Number 51: 9-11.

Bishop, T. (2009). "Microsoft cuts Popfly mashup tool, citing economic 'refocus'." Retrieved 4 june 2011, from http://www.techflash.com/seattle/2009/07/Microsoft_cuts_Popfly_mashup_tool_citing_economic_refocus_50958577.html.

Bolin, M. (2005a). End-User Programming for the Web, Master Thesis, . MSc. Master, MIT.

Bolin, M., M. Webber, P. Rha, T. Wilson and R. C. Miller (2005b). Automation and customization of rendered web pages. Proceedings of the 18th annual ACM symposium on User interface software and technology. Seattle, WA, USA, ACM.

Bozzon, A., M. Brambilla, S. Ceri and P. Fraternali (2010). Liquid query: multi-domain exploratory search on the web. Proceedings of the 19th international conference on World wide web, ACM.

Brin, S. and L. Page (1998). "The anatomy of a large-scale hypertextual Web search engine." Computer Networks and ISDN Systems 30(1-7): 107-117.

Bush, V. (1945). As We May Think, The Atlantic Monthly, Electronic version reproduced in http://adammikeal.com/courses/chi/files/jan26.bush.pdf.

Chen, J.-W., DDS, MS, MSab and J. Zhang (2007). "Comparing Text-based and Graphic User Interfaces for Novice and Expert Users." AMIA Annu Symp Proc.: 125–129.

Chisholm, W., G. Vanderheiden and T. R. D. C. U. o. W.-.-. Madison;. (2000). "HTML Techniques for Web Content Accessibility Guidelines 1.0." Retrieved 22 May 2011, from http://www.w3.org/TR/2000/NOTE-WCAG10-HTML-TECHS-20001106/.

Cooley, R., B. Mobasher and J. Srivsatava. (1997). Web Mining: Information and Pattern Discovery on the Word Wide Web. the 9th IEEE International Conference on Tools with AI (ICTAI,97).

Delaglio, F. G., S. Vuister, G. W. Zhu, G. Pfeifer, J. Bax, A. (1995). "NMR Pipe: A multidimensional spectral processing system based on UNIX pipes." JOURNAL OF BIOMOLECULAR NMR Vol 6(3): 277-293.

Duboc, A. L. d. C. L. (2009). A Framework for the Characterization and Analysis of Software Systems Scalability. PhD, University College London.

Eser, K., H. Eben, B. Rob, C. Allen, M. Paul and Z. Haixia (2005). A1: end-user programming for web-based system administration. Proceedings of the 18th annual ACM symposium on User interface software and technology. Seattle, WA, USA, ACM.

Fagan, J. C. (2007). Mashing up Multiple Web Feeds Using yahoo! pipes. Computers in Libraries, Information Today Inc. 27: 10-17.

Foley, M. J. (2009). "Microsoft to shut down its Popfly mashup tool." Retrieved 4 June 2011, from http://www.zdnet.com/blog/microsoft/microsoft-to-shut-down-its-popfly-mashup-tool/3378?tag=content;search-results-rivers.

Freed, N. and N. Borenstein. (1996). "Multipurpose Internet Mail Extensions." Request for Comments: 2046; Retrieved 25th Jan, 2010 from http://www.ietf.org/rfc/rfc2046.txt.

Freed, N. and N. Borenstein (1996). Multipurpose Internet Mail Extensions Part One: Format of Internet Message Bodies, RFC 2046, Network Working Group, Internet Engineering Task Force (IETF).

Freeman, E., E. Freeman, B. Bates and K. Sierra (2004). Head First Design Patterns, O' Reilly & Associates, Inc.

Gamma, E., R. Helm, R. Johnson and J. Vlissides (1995). Design patterns: elements of reusable object-oriented software, Addison-Wesley Longman Publishing Co., Inc.

Garshol, L. M. (2003). "BNF and EBNF: What Are They And How Do They Work?" Retrieved 31 May 2011, from http://www.massey.ac.nz/~kahawick/159331/BNF+EBNF-Garshol.pdf.

Group, M. M. (2011). "World Internet Users and Population Stats." Retrieved 12 May 2011, 2008, from http://www.internetworldstats.com/stats.htm.

Hall, W., G. Hill and H. Davis (1993). The microcosm link service. Proceedings of the fifth ACM conference on Hypertext. Seattle, Washington, United States, ACM: 256-259.

Hammersley, B. (2005). Developing Feeds with Rss and Atom, O'Reilly.

Heydon, A. and M. Najork (1999). "Mercator: A scalable, extensible Web crawler." WORLD WIDE WEB 2: 219-230.

Hinchcliffe, D. (2006). "The State of Web 2.0." Retrieved 15 April 2008, 2008, from http://web2.wsj2.com/the_state_of_web_20.htm.

Hoffman, D. L. and T. P. Novak (2000). "How to acquire customers on the web." Harvard business review 78(3): 179-188.

Holovaty, A. (2008). "In memory of chicagocrime.org." Retrieved 31 March 2008, 2008, from http://www.holovaty.com/blog/archive/2008/01/31/0102.

Hégaret, P. L. (2005). "Document object model." Retrieved 17th Jan 2011, from http://www.w3.org/DOM.

Jackson, L. A. (1999). Using the Internet for training and development. Managing human resources in the 21st century : from core concepts to strategic choice. E. E. Kossek and R. N. Block. Cincinnati, Ohio, South-Western College Publ: 1-26.

Jacobs, I. and N. Walsh. (2004, 15 December 2004). "Architecture of the World Wide Web, Volume One." W3C Recommendation Retrieved 1 july 2008, 2008, from http://www.w3.org/TR/2004/REC-webarch-20041215/.

Jeffrey I. Cole, M. Suman, P. Schramm, R. Lunn, J.-S. Aquino, D. Firth, D. Fortier, P. Gussin, K. Hanson, W. Huang, R. Singh, M. Song, M. West and Y. Yamauchi (2003). The UCLA Internet Report - year three. Surveying the Digital Future, UCLA Center for Communication Policy.

Jeffrey, P. B. (2007). "Accessmonkey: enabling and sharing end user accessibility improvements." SIGACCESS Access. Comput.(89): 3-6.

Jeffrey, P. B. and E. L. Richard (2007). Accessmonkey: a collaborative scripting framework for web users and developers. Proceedings of the 2007 international cross-disciplinary conference on Web accessibility (W4A). Banff, Canada, ACM.

Jeffrey, P. B. and E. L. Richard (2007). Accessmonkey: a collaborative scripting framework for web users and developers. Banff, Canada, ACM.

Jeffrey, W. (2007). Marmite: Towards End-User Programming for the Web. Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing, IEEE Computer Society.

Jun, F., L. Aran, H. Kasper, amp and T. Yuzuru (2004). Clip, connect, clone: combining application elements to build custom interfaces for information access. Santa Fe, NM, USA, ACM Press

Kennon, M., S. Rodgers and Sheldon (2002). "An Improved Way to Characterize Internet Users." Journal of Advertising Research 42(5): 85-94.

Kodaganallur, V. (2004). "Incorporating Language Processing into Java Applications: A JavaCC Tutorial." IEEE SOFTWARE 21(4): 70-77.

Kressin, M., B. Berger and B. P. Smith (1997). Method for adding a graphical user interface to a command line application. US. 5617527.

Krulwich, B. (1997). "Automating the Internet: Agents as User Surrogates." IEEE Internet Computing 1(4): 34-38.

Lawler, J. M. (1998). The UNIX language family. Using Computers in Linguistics. H. A. Dry, Routledge.

Lawrence, E. (2008). Fiddler version (2.x) [Software]. Available from http://www.fiddlertool.com/fiddler/version.asp.

Levine, J., T. Mason and D. Brown (1992). Lex & Yacc, O'Reilly.

Loton, T. (2008). Mashup Case Studies with Yahoo! Pipes, Lulu. com.

Loui, R. P. (2008). "In Praise of Scripting: Real Programming Pragmatism." Computer published by IEEE Computer Society 41(7): 22-26.

Lovis, C., M. K. Chapko, D. P. Martin, T. H. Payne, R. H. Baud, P. J. Hoey and S. D. Fihn (2001). "Evaluation of a Command-line Parser-based Order Entry Pathway for the Department of Veterans Affairs Electronic Patient Record." J Am Med Inform Assoc. 8(5): 486–498.

Madria, S., S. Bhowmick, W. Ng and E. Lim (1999). "Research Issues in Web Data Mining." LECTURE NOTES IN COMPUTER SCIENCE(1676): 303-312.

Manola, F. and E. Miller. (2004). "RDF Primer, W3C Recommendation 10 February 2004." Retrieved 29 March 2008, 2008, from http://www.w3.org/TR/2004/REC-rdf-primer-20040210/.

Margaret, B., C. Curtis, P. Omkar, R. Gregg, S. Jay and W. Chris (2003). End-user software engineering with assertions in the spreadsheet paradigm. Proceedings of the 25th International Conference on Software Engineering. Portland, Oregon, IEEE Computer Society.

Marianna, O., G. David, B. Petter Bae and T. Manfred (2008). Design for creating, uploading and sharing user generated content. CHI '08 extended abstracts on Human factors in computing systems. Florence, Italy, ACM.

Markoff, J. (2008). "Mashups Are Breaking the Mold at Microsoft." Retrieved 29 May 2011, from http://www.nytimes.com/2008/02/10/business/10slipstream.html.

Meeyoung, C., K. Haewoon, R. Pablo, A. Yong-Yeol and M. Sue (2007). I tube, you tube, everybody tubes: analyzing the world's largest user generated content video system. Proceedings of the 7th ACM SIGCOMM conference on Internet measurement. San Diego, California, USA, ACM.

Miller, R. C. (2002). Lightweight Structure in Text. PhD, Carnegie Mellon University.

Miller, R. C. and K. Bharat (1998). "SPHINX: a framework for creating personal, site-specific Web crawlers." Computer Networks and ISDN Systems 30(1-7): 119-130.

Miller, R. C. and B. A. Myers (1999). Lightweight Structured Text Processing. USENIX 1999 Annual Technical Conference, Monterey, CA.

Miller, R. C. and B. A. Myers (2000). Integrating a Command Shell into a Web Browser. USENIX 2000 Annual Technical Conference.

Murugesan, S. (2007). "Understanding Web 2.0." IT Professional 9(4): 34-41.

Nardi, B. A. (1993). A Small Matter of Programming: Perspectives on End User Computing. Cambridge, MA, USA. , MIT Press.

Nelson, T. H. (1965). Complex information processing: a file structure for the complex, the changing and the indeterminate. Proceedings of the 1965 20th national conference. Cleveland, Ohio, United States, ACM.

Noda, T. and S. Helwig (2005). Rich Internet Applications, Best Practices Report from UW E-Business Consortium.

O'Reilly, T. (2005). "What Is Web 2.0." Retrieved 22 September 2007, from http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html.

O'Reilly, T. (2006a). "Web 2.0 Compact Definition: Trying Again." Retrieved 15 April 2008, from http://radar.oreilly.com/archives/2006/12/web-20-compact-definition-tryi.html.

O'Reilly, T. (2006b). "Levels of the Game: The Hierarchy of Web 2.0 Applications." Retrieved 30 May 2005, from http://radar.oreilly.com/2006/07/levels-of-the-game-the-hierarc.html.

O'Reilly, T. (2007a). "What Is Web 2.0: Design Patterns and Business Models for the Next Generation of Software." COMMUNICATIONS AND STRATEGIES(65): 17-38.

O'Reilly, T. (2007b). "Pipes and Filters for the Internet." oreilly radar Retrieved 29 March 2008, from http://radar.oreilly.com/archives/2007/02/pipes-and-filters-for-the-inte.html.

Oswald, D. (2006). "HTML Parser - Project Information." Retrieved 24 April 2008, from http://htmlparser.sourceforge.net/.

Ousterhout, J. K. (1998). "Scripting: higher level programming for the 21st Century." Computer 31(3): 23-30.

PC-World-Staff. (2008). "The 25 Most Innovative Products of the Year." 2008, from http://www.pcworld.com/article/140663-5/the_25_most_innovative_products_of_the_year.html.

Piero, F. (2010). Rich Internet Applications. R. Gustavo and S.-F. Fernando. 14: 9-12.

Pilgrim, M. (2005). Greasemonkey Hacks: Tips & Tools for Remixing the Web with Firefox, O'Reilly Media.

Powers, S. (2005). "What Are Syndication Feeds." O'Reilly Media, from http://www.drexel.edu/irt_new/rmcweb/What_Are_Syndication_Feeds_PDF.pdf.

Ramachandran, S. (2010). "Web metrics: Size and number of resources." Retrieved 15th Jan, 2011, from http://code.google.com/speed/articles/web-metrics.html.

Ramey, C. (1994). "Bash, the Bourne-Again Shell." In ROSE 94. The Romanian UNIX User Group.

Raymond, K. and B. Hendrik (2000). "Web mining research: a survey." SIGKDD Explor. Newsl. 2(1): 1-15.

Rob, B., P. M. Paul and C. K. Daniel (1997). How to personalize the Web. Proceedings of the SIGCHI conference on Human factors in computing systems. Atlanta, Georgia, United States, ACM.

Shian-Hua, L., S. Chi-Sheng, C. Meng Chang, H. Jan-Ming, K. Ming-Tat and H. Yueh-Ming (1998). Extracting classification knowledge of Internet documents with

mining term associations: a semantic approach. Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval. Melbourne, Australia, ACM.

Solonchev, A. (2003). Electronic commerce method and system utilizing integration server, Google Patents.

Srivastava, T., P. Desikan and V. Kumar (2005). Web Mining – Concepts, Applications and Research Directions. Foundations and Advances in Data Mining: 275-307.

Stones, R., N. Matthew and A. Cox (2000). Beginning Linux Programming.

Svatek, V., J. Kosek, M. Labsky, J. Braza, M. Kavalec, M. Vacura, V. Vavra and V. Snasel (2003). Rainbow - Multiway Semantic Analysis of Websites. 14th international Workshop on Database and Expert Systems Applications, Washington, DC, IEEE Computer Society.

Taivalsaari, A., Tommi Mikkonen, D. Ingalls and K. Palacz (2008). Web Browser as an Application Platform: The Lively Kernel Experience. Technical report, Sun Microsystems.

Tholomé, E. (2009). "Farewell to Mashup Editor." Retrieved 4 June 2011, from http://googlemashupeditor.blogspot.com/2009/07/farewell-to-mashup-editor.html.

Udell, J. (2000). Three Faces of XML in Zope. Services, Documents, Datastores. 8th International Python Conference.

Wall, L., T. Christiansen and J. Orwant (2000). Programming Perl, O'Reilly & Associates, Sebastopol, Calif.

Watt, D. A. and D. F. Brown (2000). Programming Language Processors in Java, Prentice Hall.

Weinstock, C. B. and J. B. Goodenough (2006). On Systems Scalability, Technical Note CMU/SEI-2006-TN-012,, Software Engineering Institute.

William, H. D., J. H. Ellen and G. Monica (2009). Oxford Internet survey 2009 report: the Internet in Britain, Oxford Internet Institute.

Williams, L. G. and C. U. Smith (2004). Web Application Scalability: A model-based approach. Technical report. Software Engineering Research and Performance Engineering Services.

Winer, D. (2003). "RSS 2.0 at Harvard Law." Retrieved 29 March 2008, 2008, from http://cyber.law.harvard.edu/rss/rss.html.

Yahoo. (2007). "Yahoo Pipes." Retrieved 10 September 2007, from http://pipes.yahoo.com/pipes/.

Yann, L. L. (2005). "From boom to bust and back again." Retrieved 26 june 2008, 2008, from http://news.zdnet.co.uk/internet/0,1000000097,39238833-1,00.htm.

Yuefeng, L. (2006). "Mining Ontology for Automatically Acquiring Web User Information Needs." IEEE Transactions on Knowledge and Data Engineering 18: 554-568.

Yun, W., M. Satya and R. Arkalgud (2007). "A classification of product comparison agents." Commun. ACM 50(8): 65-71.

Zaiane, O. R., X. Man and H. Jiawei (1998). Discovering Web access patterns and trends by applying OLAP and data mining technology on Web logs. Research and Technology Advances in Digital Libraries, 1998. ADL 98. Proceedings. IEEE International.

APPENDIX A: RESEARCH ETHICS APPLICATION

School of Computer Science

Research Ethics Review Checklist

To be completed by the researcher or student undertaking the study

- This checklist should be completed for every research project that involves human participants, use of personal data and/or biological material …
- … *before* potential participants are approached to take part in any research.
- The principal investigator, or the supervisor if the applicant is a student, is responsible for exercising appropriate professional judgement in this review.
- Email the completed and signed form, together with attachments, to cs-ethicsadmin@cs.nott.ac.uk The signature page may be scanned separately if required and/or email approval given.

SECTION I: Applicant Details

1. Name	Essam Mahmoud Mahmoud Omar Eliwa
2. Status	**Undergraduate Student / Postgraduate Student / Staff** (circle as appropriate)
3. Email address	psxee@nottingham.ac.uk
4. Date of application	10/01/2013
5. Is this a resubmission?	**Yes / No** (circle as appropriate) If Yes, please give details of changes in the description overleaf.

SECTION II: For UG & Postgraduate Students Only

1. Module name and number, or MA/MSc/MPhil/PhD course and department	PhD School of computer science
2. Supervisor's name	Colin Higgins, Peter Blanchfield
3. Student ID	4032979

Before completing this form, applicants should read the guidelines at www.cs.nott.ac.uk/ethics and ensure that they understand

- **what is defined as *personal data;***
- **what is required for *valid consent;***
- **the key requirements of the Data Protection Act**
- **the University of Nottingham Research Code of Conduct**

The signature at the end of this form confirms that this has been done.

SECTION III: Project Details

1. Project Title	A Framework for Interactive End-user Web Automation
2. Proposed Start Date and Period of Study	**16th Jan 2013 – 4 days**
3. Description of Project, including aims and objectives [Please include any information which may affect the consideration of the ethics involved, eg location of study, unusual circumstances, age range of participants.]	The purpose of this study is to create a datasets of Web users' feedback using Web automation systems. The research investigates the feasibility and usefulness of a Web-based model for end-user Web automation. The aim is to empower end users to automate their Web interactions. Web automation is defined here as the study of theoretical and practical techniques for applying an end-user programming model to enable the automation of Web tasks, activities, or interactions. **Study is going to done online, participants can complete all required tasks through their own personal computers.** **All participants will be adults with some programming experience.** The case study is performed using two web-based systems, Web2Sh and Yahoo pipes. Half of the participants will be requested to implement the tasks on Web2Sh first then Yahoo pipes, the other half will do the same tasks on Yahoo pipes then on Web2Sh. Hence, based on your group the order of using the systems will be decided. They are asked to fill the consent form and three questionnaires. The flow of the study will be in the

	following order: 1. Reading this instructions sheet and tasks requirement and consenting to participate in the study; 2. Complete a questionnaire collecting data about your background and your experience as a web user and/or programmer 3. Implement the tasks on the first system; 4. Complete a questionnaire collecting data about your feedback; 5. Implement the tasks on the Second system; 6. Complete a questionnaire collecting data about your feedback.
4. Will personal data or biological materials be collected, recorded and/or analysed? If Yes, please give details of the data or materials and the methods to be used. Please describe how safe storage will be maintained according to the Data Protection Act.	**Yes / No** [delete as applicable] Participant Name (optional) and email (optional) is collected in a separate contact info sheet. The email is used only for possible follow up for future research and informing interested participant of the outcome of the study. Raw data is stored initially on surveymonkey servers and is protected under their privacy policy. This is available at: http://www.surveymonkey.net/mp/policy/privacy-policy/ Data will be downloaded by the researcher using HTTPS protocol, and will be randomised before its used for analysis. Raw data will be deleted afterwards. Email contacts will be stored in a password encrypted format on the researcher University issued computer. All recorded data will be stored anonymously and confidentially. Where used in academic publications all data will be aggregated and no personally identifiable

	information will be published. Participation in this study is voluntary; a Participant may withdraw from the study at any time and for any reason without penalty.

SECTION IV: Research Checklist (Part 1) – for completion by the applicant

Please answer each question by ticking the appropriate box:		**Yes**	**No**
1.	Does the study involve participants who are particularly vulnerable or unable to give informed consent (ie children, people with learning disabilities, prisoners, your own students)?		NO
2.	Will the study require the co-operation of a gatekeeper for the initial access to the groups of individuals to be recruited (ie students at school, members of a self-help group, residents of a nursing home)?		NO
3.	Will it be necessary for participants to take part in the study without their knowledge and consent at the time (ie covert observation of people in non-public places)?		NO
4.	Will the study involve the discussion of sensitive topics (ie sexual activity, drug use)?		NO
5.	Will participants be asked to discuss anything or partake in any activity that they may find embarrassing or traumatic?		NO
6.	Is it likely that the study will cause offence to participants for reasons of ethnicity, religion, gender, sexual orientation or culture?		NO
7.	Are drugs, placebos or other substances (ie food substances, vitamins) to be administered to the study participants or will the study involve invasive,		NO

	intrusive or potentially harmful procedures of any kind?		
8.	Will body fluids or biological material samples be obtained from participants? (eg blood, tissue etc)		NO
9.	Is pain or more than mild discomfort likely to result from the study?		NO
10.	Could the study induce psychological stress or anxiety or cause harm or negative consequences beyond the risks encountered in normal life?		NO
11.	Will the study involve prolonged or repetitive testing for each participant?		NO
12.	Will financial inducement (other than reasonable expenses and compensation for time) be offered to participants?		NO
13.	Will the study involve the recruitment of patients, staff, tissue sample, records or other data through the NHS or involve NHS sites and other property? If yes, NHS REC and R&D approvals from the relevant Trusts must be sought prior to the research being undertaken.		NO

Research Checklist (Part 2) – for completion by the applicant

Please answer each question by ticking the appropriate box:		Yes	No	Not Applicable
14.	For research conducted in public, non-governmental and private organisations and institutions (such as schools, charities, companies and offices), will approval be gained in advance from the appropriate authorities?			**Not Applicable**
15.	If the research uses human participants, personal data or the use of biological material, will written consent be gained?	**Yes**		
16.	Will participants be informed of their right to withdraw from the study at any time, without giving explanation?	**Yes**		
17.	If data is being collected, will this data be anonymised?	**Yes**		
18.	Will participants be assured of the confidentiality of any data?	**Yes**		
19.	Will all data be stored in accordance with the Data Protection Act 1998	**Yes**		
20.	Will participants be informed about who will have access to the data?	**Yes**		
21.	If quotations from participants will be used, will participants be asked for consent?	**Yes**		
22.	If audio-visual media (voice recording, video, photographs etc) will be used, will participants be asked for consent?			**Not Applicable**

23.	If digital media (eg computer records, http traffic, location logs etc) will be used, will participants be asked for consent?	**Yes**		
24.	If the research involves contact with children, will the researchers have appropriate CRB checks?			**Not Applicable**

If you have answered 'no' to all questions in Part 1 and 'yes' to all relevant questions in Part 2	This project is deemed to have minima ethical risks - go to signature page.
If you have answered 'yes' to any of the questions in Part 1 or 'no' to any of the questions in Part 2	Please describe in Section V why this i necessary and how you plan to deal with the ethical issues raised.

Please note that it is the applicant's responsibility to follow the University of Nottingham's Code of Practice on Ethical Standards and any relevant academic or professional guidelines in the conduct of the study. **This includes providing appropriate information sheets and consent forms, and ensuring confidentiality in the storage and use of data.**

Any significant change in the questions, design or conduct over the course of the research should be notified to cs-ethicsadmin@cs.nott.ac.uk and may require a new application for ethics approval.

SECTION V: Further Information as required for paragraph 2 above

[to be completed by the applicant]

None

RESEARCH ETHICS REVIEW CHECKLIST – SIGNATURE PAGE

SECTION VI: Applicant Declaration

Please tick to confirm each of the following statements before signing the form:	
I confirm that I have read the University's Code of Practice	
I confirm that I have read the guidance documents listed on page 1	
I confirm that the information provided in this application is correct	
Signature of applicant: ……. Date: ……..	
Name of applicant: ……..……..	

SECTION VII: Supervisor Declaration for UG and PG Applications

Name of Supervisor		
Please tick to confirm each of the following has been approved before signing the form:		
The participant information sheet or leaflet is appropriate for this research project		
The procedures for recruiting and obtaining informed consent are appropriate		
The data collection and storage methods, where applicable, are in accordance with the Data Protection Act.		
Have you received training in research ethics?	**Yes /No** [delete as applicable]	
For UG and PG Taught only: [Initial the statement which is applicable]	This project involves minimal ethical risk and **DOES NOT REQUIRE** consideration by the Research Ethics Committee.	
	This project involves more than minimal risk and	

	DOES REQUIRE consideration by the Research Ethics Committee	
Signature		
Date		

SECTION VII: For completion by a School Research Ethics Committee Member

This approval is only required prior to the research when:

- The checklist reveals more than minimal risk for participants and/or personal data; and/or
- The research is being carried out by a member of staff or a PhD student.

Name of SREC member			
Comments or suggestions			
Decision (circle as appropriate)	**Approve**	**Revise**	**Reject**
Signature			
Date			

On completion, an email confirming the decision should be sent to the applicant and the supervisor/principal investigator. The completed form will be kept by the School Office.

APPENDIX B: WSH FORMAL SPECIFICATION DEFINITION FILES

```
(*
* This is the formal grammar of web2sh using EBNF
* Author: Essam Eliwa, esomar@gmail.com
*)
Script ::= 'start' , WhiteSpace , ['new' , WhiteSpace ,
WebCommandName]
CodeBlock 'end;' ;
WebCommandName ::= Idetifier ;
Identifier ::= Letter {LetterOrDigit | '_'} ;
Letter ::=
'a'|'b'|'c'|'d'|'e'|'f'|'g'|'h'|'i'|'j'|'k'|'l'|'m'|'n'|'o'|'p'|'q'|
'r'|'s'
|'t'|'u'|'v'|'w'|'x'|'y'|'z'|
'A'|'B'|'C'|'D'|'E'|'F'|'G'|'H'|'I'|'J'|'K'|'L'|'M'|'N'|'O'|'P'|'Q'|
'R'|'S'
|'T'|'U'|'V'|'W'|'X'|'Y'|'Z' ;
Digit ::= '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9' ;
LetterOrDigit ::= Letter | Digit ;
Number ::= ['-'] Digit {Digit} ['.'] {Digit};
WhiteSpace ::= ' ' , {' '} ;
CodeBlock ::= '{' {Statement} '}' ;
VariableName ::= '$' , Idetifier ;
UnOp ::= '!' ;
BinOp ::= '+' | '-' | '*' | '/' | '^' | '%' ;
LogOp ::= '<' | '<=' | '>' | '>=' | '==' | '!=' ;
EqOp ::= '=' ;
Eos ::= ';' ;
Character ::= ? All Unicode characters - '"' ? ;
String ::= '"' {Character} '"' ;
Redirection ::= '=>' , (VariableName | FilePath) |
'=>>' , (VariableName | FilePath) ;
FilePath ::= [Character {Character} '/'] , (FilePath | FileName) ;
FileName ::= Character {Character} ;
Expression ::= "null" | "false" | "true" | Number | String | Item |
VariableName | Expression BinOp Expression | UnOp Expression ;
TagName ::= '<' , Idetifier
Item ::= TagName, { WhiteSpace, Idetifier , EqOp, String } , ('/>' |
{Character | Item}, TagName, '/>')
LogExp ::= Expression , LogOp , Expression [ WhiteSpace 'and'
LogExp] [
WhiteSpace 'Or' LogExp] ;
WebCommand ::= WebCommandName , { WhiteSpace , Flag} , { WhiteSpace
,
Parameters} ;
Flag ::= "-" LetterOrDigit {LetterOrDigit} ;
Parameter ::= String | Number | VariableName ;
Pipe ::= '|' , WebCommand ;
Tee ::= 'tee' , ( WebCommand | FilePath ) , Pipe ;
Stmt ::= WebCommand , [ {Pipe} | Tee | Redirection ] , Eos |
VariableName , `=´ , Expression , Eos | ;
/**
* JavaCC basic web2sh parser grammar
* Author: Essam Eliwa, esomar@gmail.com
*/
```

```
//
options
{
JDK_VERSION = "1.6";
LOOKAHEAD= 1;
STATIC = false;
ERROR_REPORTING = true;
JAVA_UNICODE_ESCAPE = true;
DEBUG_PARSER = false;
DEBUG_LOOKAHEAD = false;
DEBUG_TOKEN_MANAGER = false;
COMMON_TOKEN_ACTION = true;
IGNORE_CASE = true;
VISITOR = true;
MULTI = true;
NODE_EXTENDS = "org.web2sh.nodes.MyNode";
TRACK_TOKENS = true;
/*TRACK_TOKENS Inserts jjtGetFirstToken(), jjtSetFirstToken(),
// getLastToken(), and jjtSetLastToken() methods in SimpleNode.
// The FirstToken is automatically set up on entry to a node scope;
// the LastToken is automatically set up on exit from a node scope*/
}
PARSER_BEGIN(web2shParser)
package org.web2sh.parser;
import java.io.*;
import java.util.*;
import org.web2sh.nodes.*;
import org.web2sh.interpreter.*;
import org.web2sh.io.streams.*;
public class web2shParser
{
/**
* send AST to Interpreter (the root node of the AST) after a
successful
parse.
* @return the void
*/
public void rootNode()
{
Web2shInterpreter interpreter = new
Web2shInterpreter(jjtree.rootNode());
// return jjtree.rootNode();
}
}
PARSER_END(web2shParser)
SKIP :
{
" "
| "\t"
| "\n"
| "\r"
}
TOKEN_MGR_DECLS :
{
private List < Comment > comments = new ArrayList < Comment > ();
//this arraylist hold all the program tokens to access any token
data
later on if needed,
//to be checked later
// public ArrayList < Token > tokens = new ArrayList < Token > ();
public List < Comment > getComments()
```

```
{
return comments;
}
public void clearComments()
{
comments.clear();
}
private void CommonTokenAction(Token token)
{
//tokens.add(token);
if (token.specialToken != null)
{
Token special = token.specialToken;
if (special.kind == SINGLE_LINE_COMMENT)
{
LineComment comment = new LineComment(special.beginLine,
special.beginColumn, special.endLine, special.endColumn,
special.image.substring(2));
comments.add(comment);
}
else if (special.kind == MULTI_LINE_COMMENT)
{
BlockComment comment = new BlockComment(special.beginLine,
special.beginColumn, special.endLine, special.endColumn,
special.image.substring(2, special.image.length() - 2));
comments.add(comment);
}
}
}
}
/* Handling Comments */
SPECIAL_TOKEN :
{
< SINGLE_LINE_COMMENT :
"//" (~[ "\n", "\r" ])*
(
"\n"
| "\r"
| "\r\n"
)? >
}
MORE :
{
< "/*" > : IN_MULTI_LINE_COMMENT
}
< IN_MULTI_LINE_COMMENT >
SPECIAL_TOKEN :
{
< MULTI_LINE_COMMENT : "*/" > : DEFAULT
}
< IN_MULTI_LINE_COMMENT >
MORE :
{
< ~[ ] >
}
/* RESERVED WORDS AND LITERALS */
TOKEN : /* KEY WORDS */
{
< EXIT : "exit" ";" >
{
System.out.println("Quitting...");
```

```
System.exit(0);
}
}
TOKEN : /* OPERATORS */
{
< PIPE : "|" >
| < TEE : "tee" >
}
TOKEN :
{
< VARNAME : "$" < IDENTIFIER > >
| < IDENTIFIER :
< LETTER >
(
< LETTER >
| < DIGIT >
)* >
| < FLAG :
"-"
(
< LETTER >
| < DIGIT >
) >
| < #LETTER : [ "_", "a"-"z", "A"-"Z" ] >
| < #DIGIT : [ "0"-"9" ] >
}
TOKEN :
{
< CHARACTER_LITERAL :
"'"
(
(~[ "'", "\\", "\n", "\r" ])
|
(
"\\"
(
[ "n", "t", "b", "r", "f", "\\", "'", "\"" ]
| [ "0"-"7" ] ([ "0"-"7" ])?
| [ "0"-"3" ] [ "0"-"7" ] [ "0"-"7" ]
)
)
| ("\\u"
[ "0"-"9", "A"-"F", "a"-"f" ]
[ "0"-"9", "A"-"F", "a"-"f" ]
[ "0"-"9", "A"-"F", "a"-"f" ]
[ "0"-"9", "A"-"F", "a"-"f" ]
)
)
"'"
>
|
< STRING_LITERAL :
"\""
(
(~[ "\"", "\\", "\n", "\r" ])
|
(
"\\"
(
[ "n", "t", "b", "r", "f", "\\", "'", "\"" ]
| [ "0"-"7" ] ([ "0"-"7" ])?
```

```
| [ "0"-"3" ] [ "0"-"7" ] [ "0"-"7" ]
)
)
| ("\\u"
[ "0"-"9", "A"-"F", "a"-"f" ]
[ "0"-"9", "A"-"F", "a"-"f" ]
[ "0"-"9", "A"-"F", "a"-"f" ]
[ "0"-"9", "A"-"F", "a"-"f" ]
)
)*
"\""
>
}
SimpleNode start() #Program :
{
ASTPipedCommand rootNode;
}
{
rootNode = pipedCom() ";"
{
jjtThis.setName("Program");
jjtThis.setImage(rootNode.getImage());
jjtThis.setData(rootNode.getData());
return jjtThis;
}
}
ASTPipedCommand pipedCom() #PipedCommand :
{
SimpleNode node;
SimpleNode connector;
jjtThis.setName("PipedCommand");
}
{
node = webCom() { jjtThis.setData(node.getData());
jjtThis.setImage(node.getImage());}
(
connector = pipe() node = webCom(){ jjtThis.addData(node.getData());
jjtThis.setImage(connector.getImage() +node.getImage()); }
| connector = tee() node = params() {
jjtThis.addData(node.getData());
jjtThis.setImage(connector.getImage() +node.getImage()); }
)*
{
return jjtThis;
}
}
ASTWebCommand webCom() #WebCommand :
{
Token t;
ASTParams params;
ASTFlags flags;
}
{
t = Identifier()
{
jjtThis.setImage(t.image);
jjtThis.setName(t.image);
}
(
flags = flags()
{
```

```
jjtThis.setImage(flags.getImage());
flags.setType("FLAG");
jjtThis.addData(0, flags);
}
)?
(
params = params()
{
jjtThis.setImage(params.getImage());
params.setType("PARAM");
jjtThis.addData(1, params);
}
)?
{
return jjtThis;
}
}
Token Identifier() #void :
{
Token t;
}
{
t = < IDENTIFIER >
{
return t;
}
}
ASTFlag flag() #Flag :
{
Token t;
jjtThis.setName("flag");
}
{
t = < FLAG >
{
jjtThis.setImage(t.image);
jjtThis.addData(t);
return jjtThis;
}
}
ASTFlags flags() #Flags :
{
ASTFlag flagNode;
jjtThis.setName("flags");
}
{
(
flagNode = flag()
{
jjtThis.setImage(flagNode.getImage());
jjtThis.addData(flagNode);
}
)+
{
return jjtThis;
}
}
ASTParam param() #Param :
{
Token t;
jjtThis.setName("param");
```

```
}
{
t = < VARNAME >
{
jjtThis.setImage(t.image);
jjtThis.setType("var");
return jjtThis;
}
|
t = < STRING_LITERAL >
{
jjtThis.addData(t);
jjtThis.setImage(t.image);
jjtThis.setType("String");
return jjtThis;
}
}
ASTParams params() #Params :
{
ASTParam paramNode;
jjtThis.setName("params");
}
{
(
paramNode = param()
{
jjtThis.setImage(paramNode.getImage());
jjtThis.addData(paramNode);
}
)+
{
return jjtThis;
}
}
ASTPIPE pipe() #PIPE :
{
jjtThis.setName("PIPE");
Token t;
}
{
t = < PIPE >
{
jjtThis.addData(t);
jjtThis.setImage(t.image);
return jjtThis;
}
}
ASTTEE tee() #TEE :
{
jjtThis.setName("TEE");
Token t;
}
{
t = < TEE >
{
jjtThis.addData(t);
jjtThis.setImage(t.image);
return jjtThis;
}
}
/*@bgen(jjtree) Generated By:JJTree: Do not edit this line.
```

```
web2shGrammar.jj */
/*@egen*/ /**
* JavaCC WSh parser grammar, generated by JJTree
*/
//
options
{
JDK_VERSION = "1.6";
LOOKAHEAD= 1;
STATIC = false;
ERROR_REPORTING = true;
JAVA_UNICODE_ESCAPE = true;
DEBUG_PARSER = false;
DEBUG_LOOKAHEAD = false;
DEBUG_TOKEN_MANAGER = false;
COMMON_TOKEN_ACTION = true;
IGNORE_CASE = true;
}
PARSER_BEGIN(web2shParser)
package org.web2sh.parser;
import java.io.*;
import java.util.*;
import org.web2sh.nodes.*;
import org.web2sh.interpreter.*;
import org.web2sh.io.streams.*;
public class web2shParser/*@bgen(jjtree)*/implements
web2shParserTreeConstants/*@egen*/
{/*@bgen(jjtree)*/
protected JJTweb2shParserState jjtree = new JJTweb2shParserState();
/*@egen*/
/**
* send AST to Interpreter (the root node of the AST) after a
successful
parse.
* @return the void
*/
public void rootNode()
{
Web2shInterpreter interpreter = new
Web2shInterpreter(jjtree.rootNode());
// return jjtree.rootNode();
}
}
PARSER_END(web2shParser)
SKIP :
{
" "
| "\t"
| "\n"
| "\r"
}
TOKEN_MGR_DECLS :
{
private List < Comment > comments = new ArrayList < Comment > ();
//this arraylist hold all the program tokens to access any token
data
later on if needed,
//to be checked later
// public ArrayList < Token > tokens = new ArrayList < Token > ();
public List < Comment > getComments()
{
```

```
return comments;
}
public void clearComments()
{
comments.clear();
}
private void CommonTokenAction(Token token)
{
//tokens.add(token);
if (token.specialToken != null)
{
Token special = token.specialToken;
if (special.kind == SINGLE_LINE_COMMENT)
{
LineComment comment = new LineComment(special.beginLine,
special.beginColumn, special.endLine, special.endColumn,
special.image.substring(2));
comments.add(comment);
}
else if (special.kind == MULTI_LINE_COMMENT)
{
BlockComment comment = new BlockComment(special.beginLine,
special.beginColumn, special.endLine, special.endColumn,
special.image.substring(2, special.image.length() - 2));
comments.add(comment);
}
}
}
}
/* Handling Comments */
SPECIAL_TOKEN :
{
< SINGLE_LINE_COMMENT :
"//" (~[ "\n", "\r" ])*
(
"\n"
| "\r"
| "\r\n"
)? >
}
MORE :
{
< "/*" > : IN_MULTI_LINE_COMMENT
}
< IN_MULTI_LINE_COMMENT >
SPECIAL_TOKEN :
{
< MULTI_LINE_COMMENT : "*/" > : DEFAULT
}
< IN_MULTI_LINE_COMMENT >
MORE :
{
< ~[ ] >
}
/* RESERVED WORDS AND LITERALS */
TOKEN : /* KEY WORDS */
{
< EXIT : "exit" ";" >
{
System.out.println("Quitting...");
System.exit(0);
```

```
}
}
TOKEN : /* OPERATORS */
{
< PIPE : "|" >
| < TEE : "tee" >
}
TOKEN :
{
< VARNAME : "$" < IDENTIFIER > >
| < IDENTIFIER :
< LETTER >
(
< LETTER >
| < DIGIT >
)* >
| < FLAG :
"_"
(
< LETTER >
| < DIGIT >
) >
| < #LETTER : [ "_", "a"-"z", "A"-"Z" ] >
| < #DIGIT : [ "0"-"9" ] >
}
TOKEN :
{
< CHARACTER_LITERAL :
"'"
(
(~[ "'", "\\", "\n", "\r" ])
|
(
"\\"
(
[ "n", "t", "b", "r", "f", "\\", "'", "\"" ]
| [ "0"-"7" ] ([ "0"-"7" ])?
| [ "0"-"3" ] [ "0"-"7" ] [ "0"-"7" ]
)
)
| ("\\u"
[ "0"-"9", "A"-"F", "a"-"f" ]
[ "0"-"9", "A"-"F", "a"-"f" ]
[ "0"-"9", "A"-"F", "a"-"f" ]
[ "0"-"9", "A"-"F", "a"-"f" ]
)
)
"'"
>
|
< STRING_LITERAL :
"\""
(
(~[ "\"", "\\", "\n", "\r" ])
|
(
"\\"
(
[ "n", "t", "b", "r", "f", "\\", "'", "\"" ]
| [ "0"-"7" ] ([ "0"-"7" ])?
| [ "0"-"3" ] [ "0"-"7" ] [ "0"-"7" ]
```

```
)
)
| ("\\u"
[ "0"-"9", "A"-"F", "a"-"f" ]
[ "0"-"9", "A"-"F", "a"-"f" ]
[ "0"-"9", "A"-"F", "a"-"f" ]
[ "0"-"9", "A"-"F", "a"-"f" ]
)
)*
"\""
>
}
SimpleNode start() :
{/*@bgen(jjtree) Program */
ASTProgram jjtn000 = new ASTProgram(JJTPROGRAM);
boolean jjtc000 = true;
jjtree.openNodeScope(jjtn000);
jjtn000.jjtSetFirstToken(getToken(1));
/*@egen*/
ASTPipedCommand rootNode;
}
{/*@bgen(jjtree) Program */
try {
/*@egen*/
rootNode = pipedCom() ";"/*@bgen(jjtree)*/
{
jjtree.closeNodeScope(jjtn000, true);
jjtc000 = false;
jjtn000.jjtSetLastToken(getToken(0));
}
/*@egen*/
{
jjtn000.setName("Program");
jjtn000.setImage(rootNode.getImage());
jjtn000.setData(rootNode.getData());
return jjtn000;
}/*@bgen(jjtree)*/
} catch (Throwable jjte000) {
if (jjtc000) {
jjtree.clearNodeScope(jjtn000);
jjtc000 = false;
} else {
jjtree.popNode();
}
if (jjte000 instanceof RuntimeException) {
throw (RuntimeException)jjte000;
}
if (jjte000 instanceof ParseException) {
throw (ParseException)jjte000;
}
throw (Error)jjte000;
} finally {
if (jjtc000) {
jjtree.closeNodeScope(jjtn000, true);
jjtn000.jjtSetLastToken(getToken(0));
}
}
/*@egen*/
}
ASTPipedCommand pipedCom() :
{/*@bgen(jjtree) PipedCommand */
```

```
ASTPipedCommand jjtn000 = new ASTPipedCommand(JJTPIPEDCOMMAND);
boolean jjtc000 = true;
jjtree.openNodeScope(jjtn000);
jjtn000.jjtSetFirstToken(getToken(1));
/*@egen*/
SimpleNode node;
SimpleNode connector;
jjtn000.setName("PipedCommand");
}
{/*@bgen(jjtree) PipedCommand */
try {
/*@egen*/
node = webCom() { jjtn000.setData(node.getData());
jjtn000.setImage(node.getImage());}
(
connector = pipe() node = webCom(){ jjtn000.addData(node.getData());
jjtn000.setImage(connector.getImage() +node.getImage()); }
| connector = tee() node = params() {
jjtn000.addData(node.getData());
jjtn000.setImage(connector.getImage() +node.getImage()); }
)*/*@bgen(jjtree)*/
{
jjtree.closeNodeScope(jjtn000, true);
jjtc000 = false;
jjtn000.jjtSetLastToken(getToken(0));
}
/*@egen*/
{
return jjtn000;
}/*@bgen(jjtree)*/
} catch (Throwable jjte000) {
if (jjtc000) {
jjtree.clearNodeScope(jjtn000);
jjtc000 = false;
} else {
jjtree.popNode();
}
if (jjte000 instanceof RuntimeException) {
throw (RuntimeException)jjte000;
}
if (jjte000 instanceof ParseException) {
throw (ParseException)jjte000;
}
throw (Error)jjte000;
} finally {
if (jjtc000) {
jjtree.closeNodeScope(jjtn000, true);
jjtn000.jjtSetLastToken(getToken(0));
}
}
/*@egen*/
}
ASTWebCommand webCom() :
{/*@bgen(jjtree) WebCommand */
ASTWebCommand jjtn000 = new ASTWebCommand(JJTWEBCOMMAND);
boolean jjtc000 = true;
jjtree.openNodeScope(jjtn000);
jjtn000.jjtSetFirstToken(getToken(1));
/*@egen*/
Token t;
ASTParams params;
```

```
ASTFlags flags;
}
{/*@bgen(jjtree) WebCommand */
try {
/*@egen*/
t = Identifier()
{
jjtn000.setImage(t.image);
jjtn000.setName(t.image);
}
(
flags = flags()
{
jjtn000.setImage(flags.getImage());
flags.setType("FLAG");
jjtn000.addData(0, flags);
}
)?
(
params = params()
{
jjtn000.setImage(params.getImage());
params.setType("PARAM");
jjtn000.addData(1, params);
}
)?/*@bgen(jjtree)*/
{
jjtree.closeNodeScope(jjtn000, true);
jjtc000 = false;
jjtn000.jjtSetLastToken(getToken(0));
}
/*@egen*/
{
return jjtn000;
}/*@bgen(jjtree)*/
} catch (Throwable jjte000) {
if (jjtc000) {
jjtree.clearNodeScope(jjtn000);
jjtc000 = false;
} else {
jjtree.popNode();
}
if (jjte000 instanceof RuntimeException) {
throw (RuntimeException)jjte000;
}
if (jjte000 instanceof ParseException) {
throw (ParseException)jjte000;
}
throw (Error)jjte000;
} finally {
if (jjtc000) {
jjtree.closeNodeScope(jjtn000, true);
jjtn000.jjtSetLastToken(getToken(0));
}
}
/*@egen*/
}
Token Identifier() :
{
Token t;
}
```

```
{
t = < IDENTIFIER >
{
return t;
}
}
ASTFlag flag() :
{/*@bgen(jjtree) Flag */
ASTFlag jjtn000 = new ASTFlag(JJTFLAG);
boolean jjtc000 = true;
jjtree.openNodeScope(jjtn000);
jjtn000.jjtSetFirstToken(getToken(1));
/*@egen*/
Token t;
jjtn000.setName("flag");
}
{/*@bgen(jjtree) Flag */
try {
/*@egen*/
t = < FLAG >/*@bgen(jjtree)*/
{
jjtree.closeNodeScope(jjtn000, true);
jjtc000 = false;
jjtn000.jjtSetLastToken(getToken(0));
}
/*@egen*/
{
jjtn000.setImage(t.image);
jjtn000.addData(t);
return jjtn000;
}/*@bgen(jjtree)*/
} finally {
if (jjtc000) {
jjtree.closeNodeScope(jjtn000, true);
jjtn000.jjtSetLastToken(getToken(0));
}
}
/*@egen*/
}
ASTFlags flags() :
{/*@bgen(jjtree) Flags */
ASTFlags jjtn000 = new ASTFlags(JJTFLAGS);
boolean jjtc000 = true;
jjtree.openNodeScope(jjtn000);
jjtn000.jjtSetFirstToken(getToken(1));
/*@egen*/
ASTFlag flagNode;
jjtn000.setName("flags");
}
{/*@bgen(jjtree) Flags */
try {
/*@egen*/
(
flagNode = flag()
{
jjtn000.setImage(flagNode.getImage());
jjtn000.addData(flagNode);
}
)+/*@bgen(jjtree)*/
{
jjtree.closeNodeScope(jjtn000, true);
```

```
jjtc000 = false;
jjtn000.jjtSetLastToken(getToken(0));
}
/*@egen*/
{
return jjtn000;
}/*@bgen(jjtree)*/
} catch (Throwable jjte000) {
if (jjtc000) {
jjtree.clearNodeScope(jjtn000);
jjtc000 = false;
} else {
jjtree.popNode();
}
if (jjte000 instanceof RuntimeException) {
throw (RuntimeException)jjte000;
}
if (jjte000 instanceof ParseException) {
throw (ParseException)jjte000;
}
throw (Error)jjte000;
} finally {
if (jjtc000) {
jjtree.closeNodeScope(jjtn000, true);
jjtn000.jjtSetLastToken(getToken(0));
}
}
/*@egen*/
}
ASTParam param() :
{/*@bgen(jjtree) Param */
ASTParam jjtn000 = new ASTParam(JJTPARAM);
boolean jjtc000 = true;
jjtree.openNodeScope(jjtn000);
jjtn000.jjtSetFirstToken(getToken(1));
/*@egen*/
Token t;
jjtn000.setName("param");
}
{/*@bgen(jjtree) Param */
try {
/*@egen*/
t = < VARNAME >/*@bgen(jjtree)*/
{
jjtree.closeNodeScope(jjtn000, true);
jjtc000 = false;
jjtn000.jjtSetLastToken(getToken(0));
}
/*@egen*/
{
jjtn000.setImage(t.image);
jjtn000.setType("var");
return jjtn000;
}
|
t = < STRING_LITERAL >/*@bgen(jjtree)*/
{
jjtree.closeNodeScope(jjtn000, true);
jjtc000 = false;
jjtn000.jjtSetLastToken(getToken(0));
}
```

```
/*@egen*/
{
jjtn000.addData(t);
jjtn000.setImage(t.image);
jjtn000.setType("String");
return jjtn000;
}/*@bgen(jjtree)*/
} finally {
if (jjtc000) {
jjtree.closeNodeScope(jjtn000, true);
jjtn000.jjtSetLastToken(getToken(0));
}
}
/*@egen*/
}
ASTParams params() :
{/*@bgen(jjtree) Params */
ASTParams jjtn000 = new ASTParams(JJTPARAMS);
boolean jjtc000 = true;
jjtree.openNodeScope(jjtn000);
jjtn000.jjtSetFirstToken(getToken(1));
/*@egen*/
ASTParam paramNode;
jjtn000.setName("params");
}
{/*@bgen(jjtree) Params */
try {
/*@egen*/
(
paramNode = param()
{
jjtn000.setImage(paramNode.getImage());
jjtn000.addData(paramNode);
}
)+/*@bgen(jjtree)*/
{
jjtree.closeNodeScope(jjtn000, true);
jjtc000 = false;
jjtn000.jjtSetLastToken(getToken(0));
}
/*@egen*/
{
return jjtn000;
}/*@bgen(jjtree)*/
} catch (Throwable jjte000) {
if (jjtc000) {
jjtree.clearNodeScope(jjtn000);
jjtc000 = false;
} else {
jjtree.popNode();
}
if (jjte000 instanceof RuntimeException) {
throw (RuntimeException)jjte000;
}
if (jjte000 instanceof ParseException) {
throw (ParseException)jjte000;
}
throw (Error)jjte000;
} finally {
if (jjtc000) {
jjtree.closeNodeScope(jjtn000, true);
```

```
jjtn000.jjtSetLastToken(getToken(0));
}
}
/*@egen*/
}
ASTPIPE pipe() :
{/*@bgen(jjtree) PIPE */
ASTPIPE jjtn000 = new ASTPIPE(JJTPIPE);
boolean jjtc000 = true;
jjtree.openNodeScope(jjtn000);
jjtn000.jjtSetFirstToken(getToken(1));
/*@egen*/
jjtn000.setName("PIPE");
Token t;
}
{/*@bgen(jjtree) PIPE */
try {
/*@egen*/
t = < PIPE >/*@bgen(jjtree)*/
{
jjtree.closeNodeScope(jjtn000, true);
jjtc000 = false;
jjtn000.jjtSetLastToken(getToken(0));
}
/*@egen*/
{
jjtn000.addData(t);
jjtn000.setImage(t.image);
return jjtn000;
}/*@bgen(jjtree)*/
} finally {
if (jjtc000) {
jjtree.closeNodeScope(jjtn000, true);
jjtn000.jjtSetLastToken(getToken(0));
}
}
/*@egen*/
}
ASTTEE tee() :
{/*@bgen(jjtree) TEE */
ASTTEE jjtn000 = new ASTTEE(JJTTEE);
boolean jjtc000 = true;
jjtree.openNodeScope(jjtn000);
jjtn000.jjtSetFirstToken(getToken(1));
/*@egen*/
jjtn000.setName("TEE");
Token t;
}
{/*@bgen(jjtree) TEE */
try {
/*@egen*/
t = < TEE >/*@bgen(jjtree)*/
{
jjtree.closeNodeScope(jjtn000, true);
jjtc000 = false;
jjtn000.jjtSetLastToken(getToken(0));
}
/*@egen*/
{
jjtn000.addData(t);
jjtn000.setImage(t.image);
```

```
return jjtn000;
}/*@bgen(jjtree)*/
} finally {
if (jjtc000) {
jjtree.closeNodeScope(jjtn000, true);
jjtn000.jjtSetLastToken(getToken(0));
}
}
/*@egen*/
}
```

APPENDIX C: WEB2SH EXECUTE SERVLET

```
package org.web2sh.ajax;
import java.io.*;
import java.util.ArrayList;
import javax.servlet.*;
import javax.servlet.http.*;
import org.web2sh.cmds.AbstractWebCommand;
import org.web2sh.cmds.WebFinish;
import org.web2sh.parser.ParseException;
import org.web2sh.parser.SimpleNode;
import org.web2sh.parser.TokenMgrError;
import org.web2sh.parser.web2shParser;
import org.web2sh.parser.web2shParserVisitor;
import org.web2sh.parser.visitor.InterpreterVisitor;
/**
* ExecuteServlet is responsible for receiving the user commands via HTTP servlet
request
* and process the request then returning
* the result over HTTP using AJAX technology
*
* @author Essam Eliwa
* @version 1.0
*/
public class ExecuteServlet extends HttpServlet {
private static final long serialVersionUID = 1L;
private ServletContext context;
/**
* Reference to the command interface.
*/
AbstractWebCommand finish;
/**
* Reference to Reader class.
*/
org.web2sh.cmds.Reader reader;
/**
* @see javax.servlet.GenericServlet#init(javax.servlet.ServletConfig)
*/
public void init(ServletConfig config) throws ServletException {
this.context = config.getServletContext();
// System.out.println("******INFO****** Servlet context is: " +
// context.getContextPath());
}
/**
* read the request data, write the response data.
* @see javax.servlet.http.HttpServlet#doGet(javax.servlet.http.HttpServletRequest,
javax.servlet.http.HttpServletResponse)
*/
public void doGet(HttpServletRequest request, HttpServletResponse response)
```

```
throws IOException, ServletException {
String code = request.getParameter("command");
// Extract the comName which is a new command name , it will be empty
// if user was not creating new command
// target will hold the code to run.
// String newComName = request.getParameter("comName");
// have value only when create command used
// // Commented, used for for testing
// WebComParser comParser=new WebComParser(code ,newComName);
// comParser.executeStmt(code);
web2shParser parser = new web2shParser(new StringReader(code));
SimpleNode rootNode;
String result;
try {
// here traces one entry (pipedcommand/program/line)
ArrayList<String> myTracerData = new ArrayList<String>();
rootNode = parser.start();
System.out.println("rootNode string: " + rootNode.toString());
// rootNode.jjtAccept((web2shParserVisitor)new DumpPrintVistor(),
// null);
// rootNode.dump(">> ");
result = "<br> for code :" + code;
// result="<br>output is : " +
// rootNode.jjtAccept((web2shParserVisitor)new DumpPrintVistor(),
// null);//comParser.getResult();
finish = (AbstractWebCommand) rootNode.jjtAccept(
(web2shParserVisitor) new InterpreterVisitor(),
myTracerData);
// reader = new
org.web2sh.cmds.Reader(finish.getSTDIN_Next_com());
response.setContentType("text/xml");
response.setHeader("Cache-Control", "no-cache");
finish.setWebPrintWriter(response.getWriter());
finish.writeToWeb();
/*
* boolean success = false; int counter = 0; StringBuffer item = new
* StringBuffer(); int read = -1; while (!success && counter < 5) {
*
* try { read = reader.readItem(item); //
* System.out.println(item.toString()); success = true; } catch
* (IOException e) { counter++; success = false;
* System.out.print("Finish --> Reading Fail, Wait and retry\n");
* Thread.sleep(1000); e.printStackTrace(); }
*
* }
*
* result += "<br>output is : " + item.toString()
* +"---aaa";//comParser.getResult();
*
* System.out.println("res string: " + item.toString());
```

```
*/
} catch (ParseException e) {
System.err.println("***INFO*** Web2sh Parse Exception: \n"
+ e.getMessage());
parser.ReInit(System.in);
result = "***INFO*** Web2sh Parse Exception: \n" + e.getMessage()
+ "<br> code is:" + code;
} catch (TokenMgrError e) {
System.err.println("***INFO*** Web2sh TokenMgrError: \n"
+ e.getMessage());
parser.ReInit(System.in);
result = "***INFO*** Web2sh TokenMgrError: \n" + e.getMessage();
} catch (Exception e) {
System.err.println("***INFO*** Web2sh Exception: \n"
+ e.getMessage());
parser.ReInit(System.in);
result = "***INFO*** Web2sh Exception: \n" + e.getMessage();
e.printStackTrace();
}
// code to call ShInterpreter class to run java code
// ShInterpreter intprtr= new ShInterpreter(target);
// String result=intprtr.getResult();
// /Writes the result to a temp file to be used to open in a new window
witeResultToFile(result);
// Send back either "<valid>true</valid>" or "<valid>false</valid>"
// XML message depending on the validity of the data that was entered.
// Note that the content type is "text/xml".
//
// response.setContentType("text/xml");
// response.setHeader("Cache-Control", "no-cache");
// response.getWriter().write("<result>"+Integer.toString(len)+"</result>");
// response.getWriter().write(result);
System.out.println("********** End servlet**********");
}
/**
* @see javax.servlet.http.HttpServlet#doPost(javax.servlet.http.HttpServletRequest,
javax.servlet.http.HttpServletResponse)
*/
public void doPost(HttpServletRequest request, HttpServletResponse response)
throws IOException, ServletException {
/*
* String targetId = request.getParameter("id"); if ((targetId != null)
* && !accounts.containsKey(targetId.trim())) {
* accounts.put(targetId.trim(), "account data");
* request.setAttribute("targetId", targetId);
* context.getRequestDispatcher("/success.jsp").forward(request,
* response); } else {
* context.getRequestDispatcher("/error.jsp").forward(request,
* response); }
*/
```

```
doGet(request, response);
}
/**
* get the results and write to a temp file to open in a new window.
* @param result <code> String </code> contains the result for user command
*/
public void witeResultToFile(String result) {
// System.out.println("***Test Data, line sep is:
"+System.getProperty("line.separator"));
try {
// to do, change this path
File f = new File("resultWin.htm");
// System.out.println("***Test Data, AbsolutePath is:
"+f.getAbsolutePath());
// System.out.println("***Test Data, Path is: "+f.getPath());
PrintWriter out = new PrintWriter(new FileWriter(f));
out.print(result);
out.close();
} catch (IOException ex) {
ex.printStackTrace();
}
}
}
```

APPENDIX D: ABSTRACT WEB COMMAND INTERFACE

```
package org.web2sh.cmds;
import java.io.IOException;
import java.io.PipedInputStream;
import java.io.PrintWriter;
/**
* The command interface for all web commands. (Command design pattern)
* @author Essam Eliwa
* @version 1.0
*/
public interface AbstractWebCommand {
public void execute();
/**
* returns a reference to an input stream of the following command in a pipe sequence
* if no other command follows it returns the final output stream to return the result to
the user,s browser
* @return a PipedInputStream
*/
public PipedInputStream getSTDIN_Next_com();
/**
* used to assign a PipedInputStream as the standard input stream of a command
* @param stdin
*/
public void setSTDIN(PipedInputStream stdin);
/**
* write the command output to the output stream given an array of bytes
* writing should always be done by a thread and reading by another thread so no
lock happens
* use writer class to write and reader class to read
* @param b
* @return <code> boolean </code>
* @throws IOException
*/
public boolean writeStdout(byte[] b) throws IOException;
/**
* write the command output to the output stream given a string
* writing should always be done by a thread and reading by another thread so no
lock happens
* use writer class to write and reader class to read
* @param str <code> String </code>
* @return <code> boolean </code>
* @throws IOException
*/
public boolean writeStdout(String str) throws IOException;
/**
* return true if there is data over the input stream and false if there is no data
```

```
available or the stream is closed.
* @return <code> boolean </code>
*/
public boolean isInputAvailable();
/**
* web command names such as: fetch, echo etc..
* @return <code>String</code> the name of a web command
*/
public String getComName();
/**
* sets the WebPrintWriter to a PrintWriter that is used to write the
* command output
*
* @param pw PrintWriter
*/
public void setWebPrintWriter(PrintWriter pw);
/**
* used by webFinish command at the end of a pipe sequence to
* output data over http to the client browser
*/
public void writeToWeb();
}
```