A Visual Architectural Approach to Maintaining Web Applications

Ahmed E. Hassan and Richard C. Holt

School of Computer Science

University of Waterloo

200 University Avenue West

Waterloo, ON

N2L 3G1 Canada

aeehassa@plg.uwaterloo.ca

September 4, 2002

Abstract

Web applications are complex software systems which contain a rich structure with many relations between their components. Web developers are faced with many challenges when they need to gain a better understanding of these applications to maintain or evolve them. Current development tools focus primarily on implementation, with little support for the application's evolution. Web developers need tools to assist in the evolution and maintenance of web applications.

We present an approach to assist developers in understanding the structure of their web application. A set of parsers analyze the source code and pages of the web application to produce box-and-arrow architecture diagrams of the application. Using these diagrams developers can see the interactions between the various components in their application. They can also perform impact analysis studies on the application's architecture. The approach is flexible and can be re-targeted to analyze applications developed using the various current and future web technologies.

1 INTRODUCTION

Consider a fictitious startup company, *WebFlight*, which has developed a complex and dynamic software system whose functionality is delivered through the web. Users can surf to *WebFlight*'s web application where they can browse, buy, and hold airline tickets. All these activities are done using a web browser to follow links and complete order forms. Whenever the user clicks a link on a *WebFlight* web page, a request is generated and sent to *WebFlight*'s web server through the HTTP protocol. The web server receives the request and in turn invokes the appropriate actions to generate a response. For example, a new order object may be created to track the user's request. A database may be queried to retrieve relevant information to fulfill the request. The results are then transcribed to HTML and sent back to the browser, which displays them to the user. To meet a tight development schedule, the developers at *WebFlight* neglected many recommended software engineering practises such as documenting the architecture of their application.

Because of the success of *WebFlight*'s web site, the company has just hired another ten developers to work on the next release of the application. For the new release, the application needs to be migrated to a more scalable web server/application server. Furthermore, new features are to be added to meet customer's demand. The new release will support selling train tickets in addition to airline tickets.

Alas, no documentation exists to assist the recently hired developers, nor do they have any tools designed to assist in understanding the application. To aggravate matters, many of the senior developers are no longer with *WebFlight*[1)]. The other senior developers who are still with *WebFlight*, are too busy to answer the many questions that the new hires want to ask. The recently hired developers are using primitive tools such such as *grep* and text editors to analyze the existing application and to add new features to the application, while trying to meet near impossible deadlines.

This story about *WebFlight* is a typical scenario of the development of a web application. The web community is producing a huge amount of large scale, complex, and dynamic web applications that have little documentation to explain their internals and which are critical to the success of their organization. As these applications age, the knowledge of the application disappears when developers depart or move to new projects. New developers are faced with a legacy system which they must modify with little knowledge of its internals. There are many tools to assist developers during the development process, but few to assist them in evolving their application to accommodate new requirements, to run on new platforms, or simply to fix bugs.

The primary business of software is no longer new development; instead it is maintenance [Glass 1992]. Web applications are the legacy systems of the future, and developers require tools to understand these application and help maintain them.

[1)] The web community has a high turnover rate: the average employment length is just over one year [Konrad 2000].

Software engineering researchers recognize the need to apply well-studied and tested principles to the development of web applications. Recent research [Hatzimanikatis *et al.* 1995; Brereton *et al.* 1998; Tilley 1999; Antoniol *et al.* 2000; Boldyreff 2000; Ceri *et al.* 2000; Ricca and Tonella 2000] recognizes the need to adapt traditional software engineering principles to assist in the development of web applications. Unfortunately, the web development community has generally not adopted these techniques [Pressman 2000]. The techniques used nowadays by web application developers are similar to the ad-hoc ones used by their predecessors in the 1960s and 1970s.

This paper presents an approach to assist developers in maintaining their web application by providing better means to analyze and understand these applications. The approach is based on a set of parsers which examine the source code of the application and output the relations between the various components of the web application. These relations are used to generate box-and-arrow diagrams to show the architecture of the web application. Developer can navigate these diagrams and perform impact analysis on them to better understand the application. They can ask questions such as "*Which web page writes data to this database table?*", or "*Which object reads data from this database table?*". Conveniently provided answers to such questions permit developers to more rapidly and effectively maintain their web applications.

1.1 Organization of Paper

The rest of this paper is organized as follows. Section 2 describes the various components in web applications and gives an overview of the data flow in web applications. Section 3 presents an example web application, and shows its visualized architecture. Section 4 shows how to use the generated diagrams to assist developers maintaining a large web application. Section 5 explains how the data to be visualized is gathered from the many types of software artifacts in web applications and shows how the diagrams are generated from the gathered data. Section 6 describes related work, and section 7 draws conclusions from our work.

2 THE COMPONENTS OF A WEB APPLICATION

In web applications, various components written in many different languages are linked together using scripting languages. The use of scripting languages speeds up the development of the application as it provides flexibility in combining components written in different languages with mismatching interfaces. On the other hand, the use of scripting languages increases the difficulty of maintaining the application as the scripts tend to be undocumented and scattered throughout the application.

A database is an essential part of a web application as it is the primary communication interface between the many components of the web application. Unfortunately, current visualization tools for web applications focus primarily on displaying the hyperlinks between the static pages. As the web was originally developed as a document-sharing platform, these tools approach the problem of visualizing and maintaining web application as a document maintenance problem rather than a software engineering problem. They neglect the dynamic structure of the application. They fail to show the interactions between the databases, the distributed objects, and the web pages that form a web application [Tilley and Huang 2001].

The visualizations generated by our approach help remedy these problems. Reviewing studies in program maintenance and system understanding that were conducted on the development of traditional software systems [Lethbridge and Anquetil 1997; Sim 1998; Sim *et al.* 1998; Sim *et al.* 1999], we determined a set of useful relations and components that are helpful to web developers in their maintenance tasks.

Our visualizations are at a high level. We do not show diagrams of the "internals" of each components; instead our visualizations are at the component level. We show the interaction between the various components. For example, we would show that `component_1 updates databaseTable_1`, instead of showing that `func_1 updates column_1`. Our generated architecture diagrams for web applications show the main components of the application that are glued together to implement large sophisticated applications.

Furthermore, we ignore the internal architecture of the web browsers (used by the clients), web servers, or application servers as they would add complexity to the visualized system without contributing to the overall understanding of the system. The web server, application server, and the browser represent software infrastructure that is similar to the operating system and the windowing system, whose architectures are not shown when visualizing traditional software systems. We use techniques including containment and information hiding to reduce the complexity of the generated diagrams.

The following are the components that are shown in our generated diagrams:

Static pages. These contain HTML code and executable code (JavaScript) that runs on the web browser. They are served by the web server and do not need to be preprocessed by the application server.

Active pages. These include Active Server Pages and Java Server Pages. They contain a mixture of HTML tags and executable code (written in languages such as JavaScript, JScript, or VBScript). When an active page is requested, the application server preprocesses it and integrates data from various resources such as web objects or databases, to generate the final HTML web page sent to the browser.

Web objects. These are pieces of compiled code which provide a service to the rest of the software system through a defined interface. They are supported by CORBA, EJB and DCOM. They are not objects in the sense of source code object-oriented programming objects such as those defined in C++ or Java.

Multimedia objects. These include images and videos.

Databases. These are used to store data that is shared among the various components.

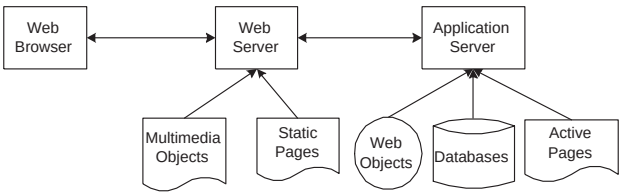

Figure 1: Dataflow Between the Components of a Web Application.

Figure 1 shows the data flow between the various components of a web application. By clicking on links or filling forms in the web browser, the user interacts with the web application. These requests are sent using the HTTP protocol to the web server which determines if it can can fulfill the request directly or if the request should be redirected to the application server to generate a response. The web server directly serves static HTML pages and multimedia content such as images, videos, or audio files. The application server processes active pages and returns the result to the web server as static HTML pages. Once the response is generated, the web server returns it as an HTML page back to the web browser, which displays it to the user. Our approach provides a tool to show these interactions to developers and assist them in understanding their applications.

3 WEBFLIGHT: A WEB APPLICATION

Our approach has been used to visualize the architecture of several large commercial and experimental web applications. These applications had over 200 distributed web objects and over 15 databases per application. In this paper, we present a simplified version of a commercial application. We have renamed the application to *WebFlight* to protect the intellectual property of the owning corporation, and we have reduced the functionality to meet the paper's space restrictions.

WebFlight is an online discount airline ticket agent. It offers discounted airline tickets for various destinations worldwide. Users can browse for tickets from various airlines. Once the user finds an appropriate ticket, the user can either buy the ticket immediately, or hold it for 24 hours. If a held ticket is not purchased within 24 hours, the ticket is cancelled.

To hold a ticket, the user must have a *WebFlight* account. To purchase a ticket, the user must have a *WebFlight* account, must provide a credit card number to charge, and can optionally provide an airline frequent flyer account number. The *WebFlight* web application was developed on the Microsoft Windows platform and contains components written in HTML, VBScript, Visual Basic, JScript, and C++. Figure 2

Figure 2: Main Page for the *WebFlight* Application.

shows a mockup of the main page of *WebFlight*, where the user can either browse, buy, or hold tickets.

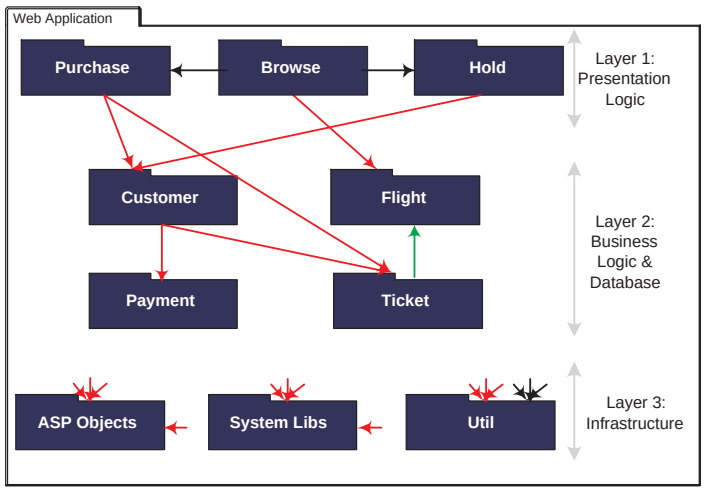

Figure 3: The Recovered Architecture of *Web Flight*.

When new developers begin working on the *WebFlight* application, they are faced with a large complex software system which they need to understand and modify. Many of them have experience developing web applications but none of them have experience working on the *WebFlight* application. They need a good understanding of the architecture of the application, of the internals of its various subsystems and the interactions between them. Ideally, they would consult the system's documentation, but no such documentation exists. Instead most of the system knowledge lives in the heads of the senior developers whose time is extremely constrained.

Using our approach we recovered the architecture of the *WebFlight* application, as shown in Figure 3.

The architecture was recovered by parsing the source code of the system using automated tools and by asking one of the senior developers for their assistance.

The architecture recovery process took approximately five hours. Most of that time was spent in getting assistance from the senior developer to layout the architecture diagrams and reduce their complexity; the automated analysis of the source code was essentially instantaneous. Once the initial recovery has been performed, the process can be repeated without requiring assistance from the senior developer. The automated recovery process can be repeated daily so that the architecture diagrams are always up-to-date.

Once the architecture is recovered, a developer can navigate the architecture using a specialized viewer. The viewer shows the various components of the web application and the interactions between them. The various components in a web application are shown by the viewer using different colors and icon shapes to assist the user in recognizing them. Blue folders represent subsystems, blue ovals represent web objects, grey pages represent active and static pages, blue boxes represent DLLs[2)], and green cylinders represent database tables. Furthermore, the viewer shows many (currently fourteen) types of relations (arrows) between the components of a web applications. To reduce the number of colors used in the diagrams for the arrows, the relations are arranged into three categories of dependencies: a black arrow indicates a hyperlink dependency, a red arrow indicates a control dependency, and a green arrow indicates a data dependency.

A textual description can be attached to each component; each time the mouse cursor is placed above such component the textual description is displayed in a pop-up window. The figures shown in the paper are edited screenshots to improve their readability.

From Figure 3, the developer can recognize that the application has a 3-layer architecture:

- The highest layer consists of the Presentation Logic which provides the web interface to the various functions of the applications: purchasing, browsing, and holding tickets. This layer contains the active pages, and static pages that form the application's interface. Current visualization tools focus on visualization components in this layer. Our approach shows more types of components and a richer set of relations between them.
- The next layer encapsulates the business rules in the application. The main concepts in the application domain reside in this layer (such as Customer, Flight, Ticket, and Payment). Furthermore, it contains the persistent storage (Databases) for the various subsystems of the application.
- The lowest layer provides the infrastructure and support for the upper layers. It provides basic support for activities such as user input verification, string manipulation, consistent web page layout, and programming language support. For an application developed on the Microsoft Windows platform, the *Infrastructure* layer contains Microsoft Windows specific components such as Active Server Pages ob-

[2)] A Dynamic Link Library (DLL) is a shared library on the Microsoft Windows operating system.

jects, Windows libraries, and general utilities. In Figure 3, the arrows to the *Infrastructure* subsystems have been truncated to simplify the diagram, as these subsystems are used extensively by the rest of the application.

4 SCENARIO: MODIFYING A SUBSYSTEM

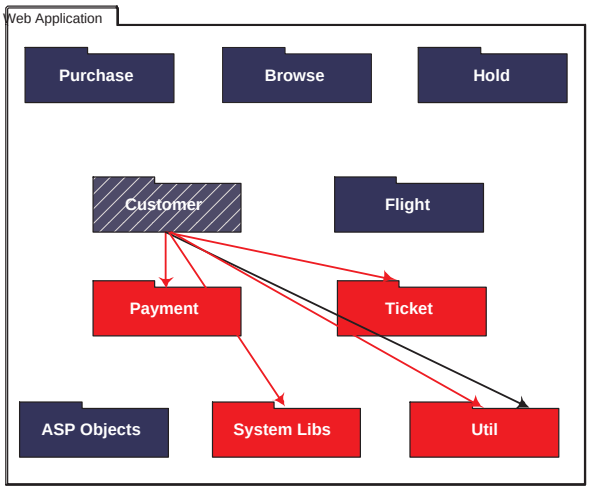

Figure 4: Subsystems Used by the *Customer* Subsystem.

We now consider this scenario for the next release, *WebFlight* needs to add support for purchasing, browsing, and holding train tickets as well as airline tickets. Joe, a new developer who just joined *WebFlight*, is given the task of performing all the changes needed in the *Customer* subsystem to support this new feature. To perform such task, Joe needs to first understand what are the various components that the *Customer* subsystem depends on so he can co-ordinate his changes with the other developers working on these subsystems. He starts up the architecture viewer and chooses the *Customer* subsystem. Then he asks the viewer to show him the subsystems that *Customer* uses, as seen in Figure 4. Then he uses the viewer to show him the subsystems that use the *Customer* subsystem, as in Figure 5. Looking at the diagram in Figure 5 Joe knows that he would need to make sure that his changes don't break the *Purchase* and *Hold* subsystems which depend on the *Customer* subsystem. He should contact the developers working on these two subsystems to ensure that they are aware of his changes.

Now that he knows all the suppliers and users of the *Customer* subsystem, he is ready to add the new feature to it. But he does not yet know the internal design of the *Customer* subsystem. Next, Joe launches the architecture viewer and double clicks on the *Customer* subsystem in Figure 3. As a result, the viewer shows the internals of the *Customer* subsystem, as seen in Figure 6. The Figure shows that there are three

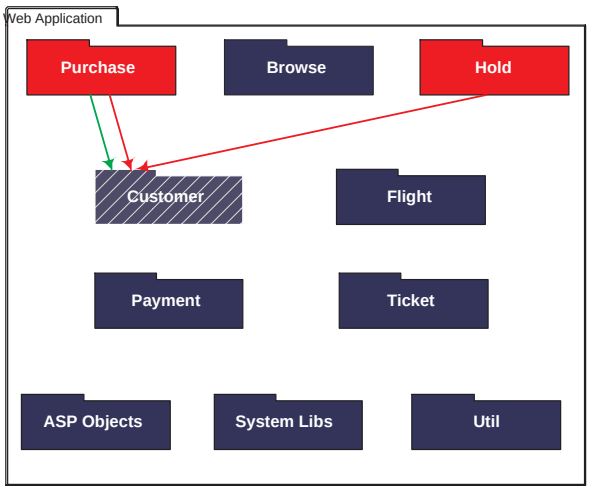

Figure 5: Subsystems Which Use the *Customer* Subsystem.

`DLL`s in the *Customer* subsystem:

- a façade `CUSTOMER.DLL` which accepts all the request from the users of the *Customer* subsystem and directs them to the appropriate component.
- a `CUSTOMER_DB.DLL` which handles all access to the `CUSTOMER` database table, and
- a `PURCHASE_ORDER.DLL` which encapsulates all the business rules for purchasing of a ticket (such as pricing details, discounts, and special offers).

Also, there is an `ITINERARY` database table to store the customer's itineraries.

Using these diagrams, Joe now has a good idea of the structure of the software system. He is now ready to start planning the addition of the new feature into the *Customer* subsystem. At any point Joe can perform more queries to understand better the interactions between the various components inside the subsystem, he is modifying, or any other subsystem. As Joe starts coding his changes, he and the other members in the team can see the effects of their changes on the architecture of the system.

Jenny, *WebFlight*'s architect, may choose to monitor the progress of the development using our viewer. She may intervene if she notices that Joe or other developers are introducing any un-wanted dependency that would complicate the architecture of the system, or make it harder to maintain. Using the visualizer, the architect and the development team are always up-to-date on the state of the software system.

5 VISUALIZING A WEB APPLICATION

In the previous section, we have shown diagrams that are recovered from the source code and pages of a web application. In this section, we present an overview of our architecture recovery process. Our

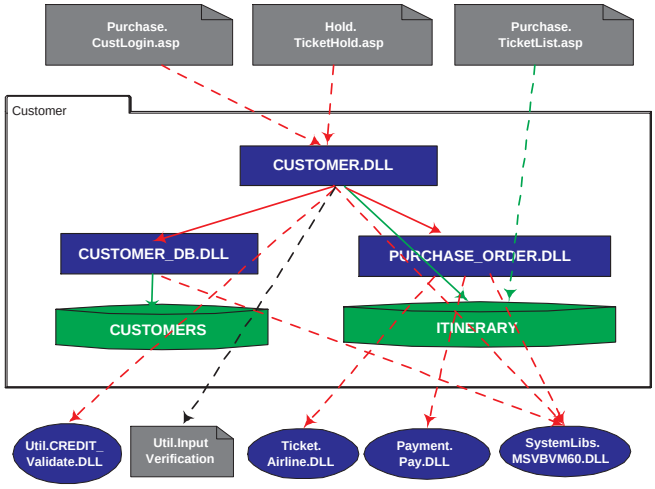

Figure 6: The Internals of the *Customer* Subsystem.

architecture visualization is a semi-automated process. It is an adaptation of a similar approach used by the Portable BookShelf (PBS) [Penny 1992; Finnigan *et al.* 1997; PBS 1998] environment to recover the architecture of traditional software systems. The PBS environment incorporates knowledge and techniques developed over the last decade in program understanding and architecture recovery. The process is broken into three main phases:

1. Extracting facts from the application's source code using a set of extractors.
2. Abstracting and merging the multi-language facts.
3. Finally, generating the architecture diagrams.

5.1 Extracting the Facts

The extraction of facts from the source code is an automated phase. A set of parsers process the various components of a web application and emit facts about these components. These facts are later used to generate the architecture diagrams.

Web applications are developed using various programming languages. To deal with web applications, we developed five types of extractors: an HTML extractor, a Server Script extractor, a DB Access extractor, a Source Code extractor, and a Binary Code extractor. Figure 7 shows an overview of the various extractors and their input and the type of facts generated by them. Each extractor parses a component or a section within a component and generates the corresponding facts. For example, the DB Access extractor analyzes an Active page and emits relations such as (`file_1` SQLInsert `DBTable_1`), meaning `file_1` contains a code segment which makes an `SQL` insert into `DBTable_1`. The Binary Code extractor reads a binary and extracts

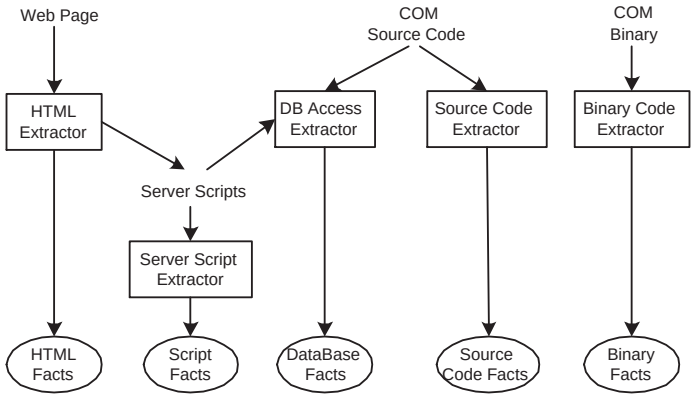

Figure 7: Conceptual Architecture of the Fact Extractors.

the function calls and the data accesses from the symbols table stored in the binary file. The Binary Code Extractor is used whenever we do not have a source code extractor for the language in which the component is written or whenever we do not have access to the source code of the component. Together these extractors emit facts for the entire web application. The extractors are designed to recover from parsing errors which are due to the different programming language dialects and extensions. They can heuristically parse for patterns in the source code, and perform sub-parsing limited to subsections of large files which are written in various programming languages. In [Hassan 2001; Hassan and Holt 2002], we present the design of the various extractors, the parsing techniques used, and the error recovery mechanisms invoked by the extractors.

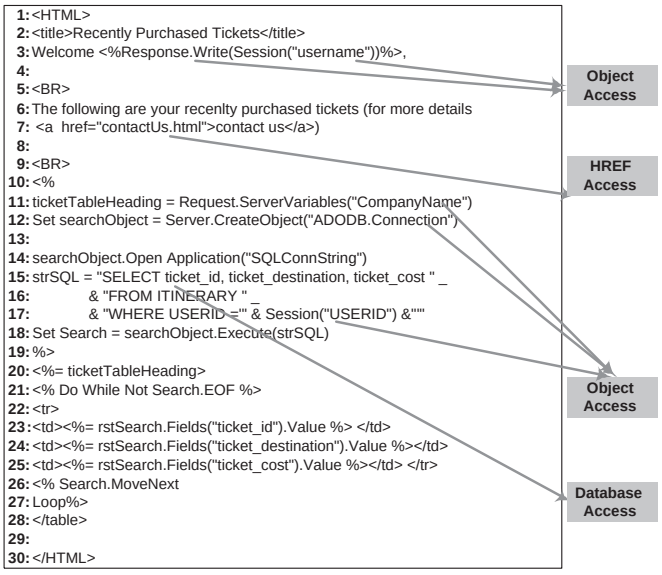

Figure 8: Code Listing of the *TicketList.asp* Page.

As an example of the recovery process, Figure 8 shows the file for the active page named *TicketList.asp*.

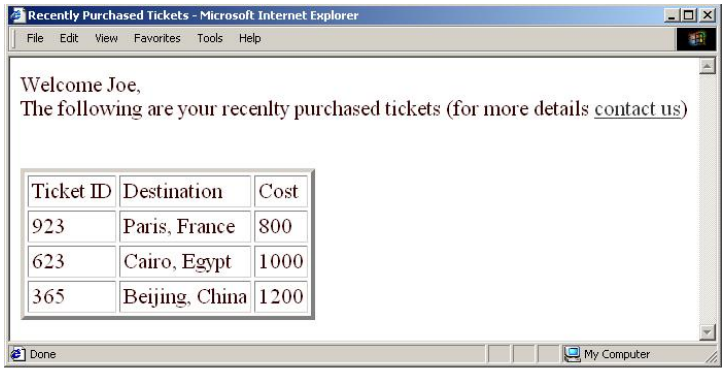

Figure 9: The *TicketList.asp* Page Viewed by the User.

Figure 9 shows the *TicketList.asp* page when viewed in the customer's web browser. The page presents a listing of tickets recently purchased by a customer. Looking at Figure 8, in lines 15-17 the page accesses the `ITINERARY` database table to retrieve the purchased ticket's information using the customer's Id. It gets the customer's Id from a Session object which is set when the customer first logs into the web application. Then in lines 21-27, the page loops through the results of the database access and prints a table.

Three types of extractors work cooperatively to extract data about the file:

- The DB Access Extractor recognizes that the *TicketList.asp* is `SELECT`ing data from a table (line 15 in Figure 8).
- The HTML extractor emits a fact indicating that *TicketList.asp* links to the *ContactUs.asp* (line 7 in Figure 8).
- The Server Script Extractor parses the VBScript code and emits the object accesses which occur in the file.

Each file in the application is processed using the extractors. Due to the large number of files and the use of various programming languages in a web application, we cannot easily visualize the application, in the following sections we detail the steps needed to visualize such large multi-language applications.

The various extractors are invoked by a shell script which crawls the directory tree of the source code for the web application. The script determines the type of the component and invokes the corresponding extractor. For example, if the script determines that a file is a binary file, the Binary Code extractor is invoked. Each extractor stores its generated facts in a file with the same name as the input file and the name of the extractor as the suffix. Later, another script crawls the directories and consolidates all the generated facts files into a single file called `THEFACTS`.

Alternatively, a network crawler can be used instead of a directory based crawler. Each crawler has its benefits and drawbacks. The network crawler would crawl the application pages starting from the

application's main web page. It won't find any files that are not accessible from its starting page. This is not a concern for the directory based crawler because all the present files are examined regardless whether they are referenced by other files or not. The directory crawler permits the developer to determine dead files that are no longer referenced by other files and which can be removed from the code base. Such a file would show up in the generated architecture diagram with no relations from other files to it (no in-arrows). If links to other pages are generated dynamically then directory crawling won't extract the links but network crawling will be able to extract these links.

Some of the facts generated by a network crawler will not match the relations generated by a directory based extractor as the preprocessing performed by the web server may alter the HTML code. For example if a file includes another file, a network extractor will mistakenly assign all the relations from the included file to the including file. Furthermore, a network crawler will not generate any facts about the scripts or the source code of the various components as they are not accessible from the network. For example if a network crawler were used to analyze the *TicketList.asp* file, then only the `HREF` access to *ContactUs.html* will be shown by our viewer. All other accesses no longer exist in the code viewed by the network crawler.

5.2 Abstracting and Merging the Extracted Facts

Each extractor emits facts that are language dependent and technology dependent. For example, a VBScript extractor outputs a fact indicating that a COM object property is accessed by the processed file; whereas the JavaScript extractor emits a fact indicating that the processed file assigns a value to a field in an Enterprise Java Bean (EJB). These facts are technology dependent (COM vs EJB), and language dependent (JavaScript vs VBScript). Abstractly, both extractors are indicating a processed file is accessing a data field in an object. In one case a file is reading a data field, and in the other case a file is updating a data field. Once the various extractors have processed the source files of the application, the facts are combined and abstracted to a higher level that is programming language and technology independent.

To handle the various kinds of facts that are extracted (and then abstracted) we use a pyramid of schemas as illustrated in Figure 10. The bottom layer of the pyramid has a schema for each source language. The next layer abstracts up to either object-oriented or procedural languages. The next layer simplifies and abstracts up to higher level facts that are common to web applications. Finally, the top layer further simplifies and abstracts to the architectural level. The schemas at these various levels will now be discussed in more detail.

Each language extractor has a schema which specifies the various entities it generates and the relations that exist between these entities. For example, Figure 11 shows the schema for the JavaScript extractor [3)].

[3)] To improve the readability of the figure we removed the attributes associated with the entities such as the line number where an entity is defined or used.

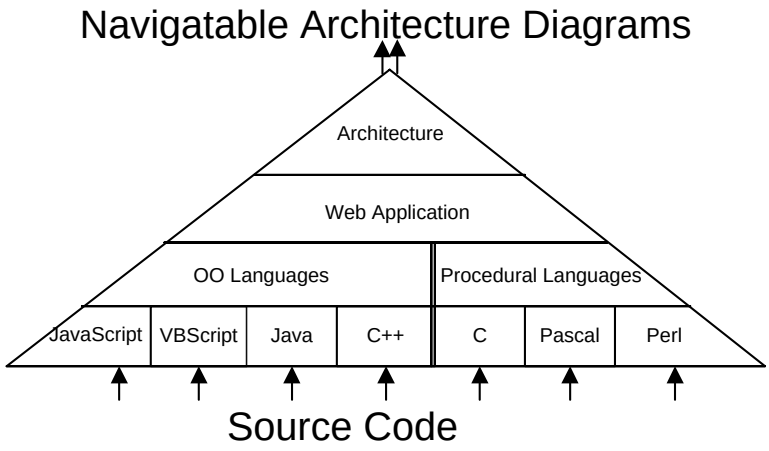

Figure 10: Pyramid of Schemas.

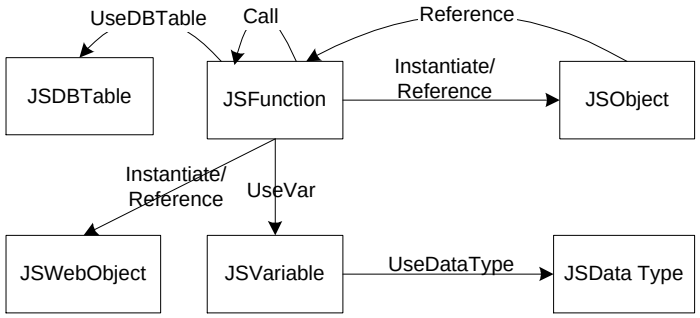

Figure 11: The JavaScript Schema.

For each extractor in our architecture recovery process, we need to provide a mapping from the schema of the extractor to the object-oriented or procedural schemas. For example, Figure 12 shows the object-oriented schema with the VBScript and JavaScript mappings. In the figure to indicate the various schemas, we prefix the VBScript entities with `VBS` and the JavaScript entities with `JS`. Using the abstracted object-oriented schema, we can now study the interaction between components written in different programming languages.

We must face the problem that the level of detail is too low and the amount of facts is too large at this level in the schema pyramid to permit us to reason about a large web application and visualize it. To resolve this problem, we introduce the next higher level schema - the Web Application Schema, which is shown in Figure 13. This schema consolidates and reduces a lot of the details in lower level schemas to single entities or relations. For example, all "`function call`" and "`data access`" edges between entities in the same component are removed. Furthermore, relations are raised to the level of a component, for example if `func_1` calls `func_2` and `func_1` is in `component_1` and `func_2` is in `component_2` then the relation is raised to the component level (`component_1` calls `component_2`) to reduce the level of detail in the generated diagrams.

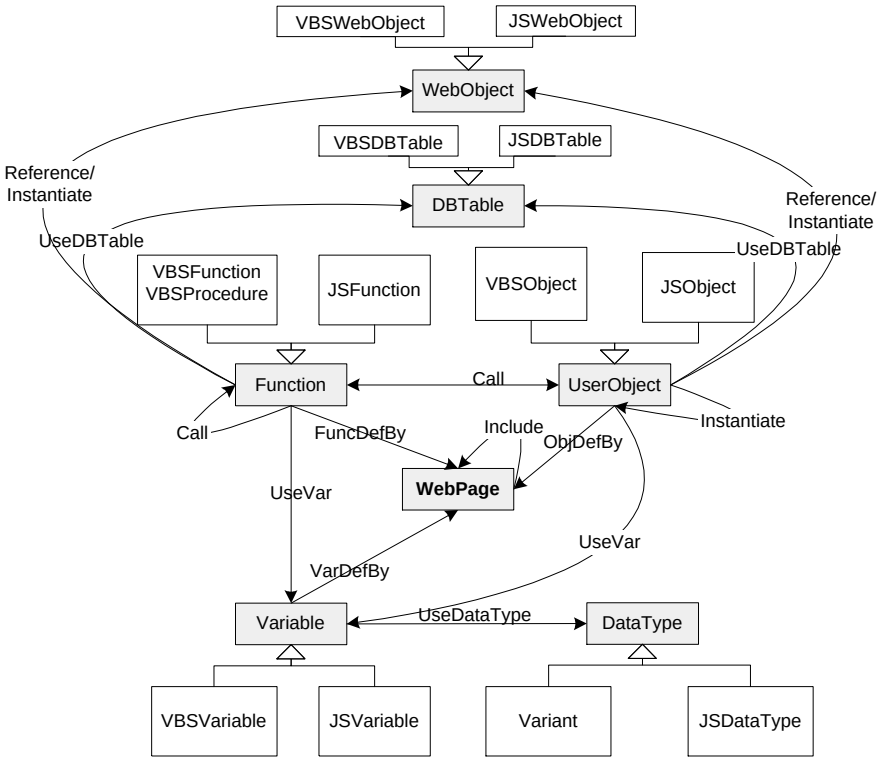

Figure 12: Mapping the JavasScript and VBScript Schemas to the OO Schema.

We use the Web Application schema as a basis for visualizing and studying simple web applications.

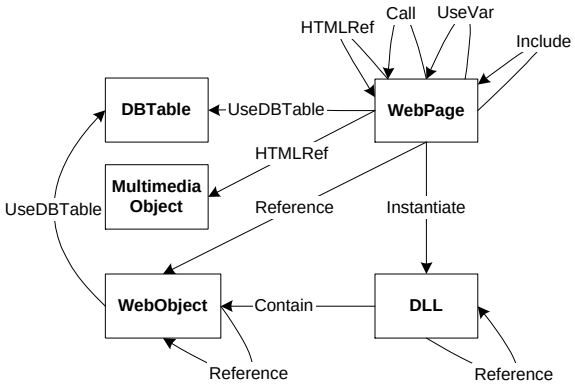

Figure 13: The Web Application Schema.

To study large web applications such as the one presented in this paper, we introduce yet a higher schema layer - the Architecture Schema. The Architecture Schema is technology and language independent. Figure 14 shows this schema. In the following section we explain how this schema reduces the complexity of the generated visualizations.

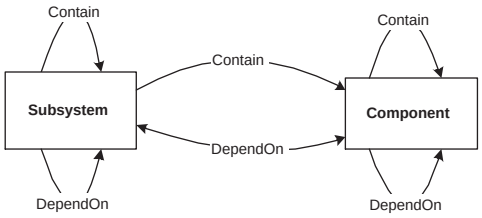

Figure 14: The Architecture Schema.

5.3 Generating the Architecture Diagrams

In this final phase, the extracted facts along with developer's or architect's input are used to produce the diagrams such as the one shown in Figure 3. Figure 15 shows the steps involved in producing the architecture diagrams.

If we were to directly use the facts at the Web Application Schema level to generate diagrams, we would get excessively complicated diagrams due to the large amount of extracted relations and components. Instead of showing all the extracted relations and artifacts in a single diagram, we decompose the artifacts of the software system into smaller meaningful subsystems. This decomposition reduces the number of artifacts shown in each diagram and improves the readability of the generated diagrams especially for large software systems.

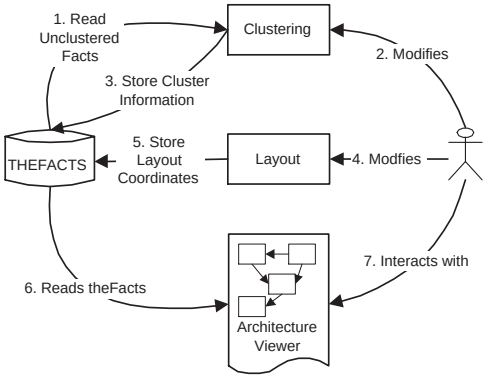

Figure 15: Generating Architecture Diagrams from the Facts.

A clustering tool reads the facts from the THEFACTS file and proposes decompositions based on heuristics such file naming conventions, development team structure, directory structure, or software metrics [Bowman 1999; Tzerpos and Holt 1996; Tzerpos and Holt 1998]. The developer then manually refines the automatically proposed clustering using their domain knowledge and available system documentation. The decomposition information along with the extracted facts is stored back into the THEFACTS file so other tools can access it.

An automatic layout tool reads the stored facts and the clustering information to generate diagrams such as the one shown in Figure 3. The layout tool attempts to minimize the line crossing in the generated architecture diagrams [Sugiyama and Misue 1991; Sugiyama *et al.* 1981]. Again, developer manual intervention is supported to improve these diagrams. By automating as much as possible this process, we are able to dramatically reduce the recovery time of large software systems.

6 RELATED WORK

Many researchers have recognized the need to adapt software engineering methodologies to the development and understanding of web applications. The work of Hatzimanikatis *et al.* in 1995 is the earliest to adapt traditional software engineering metrics to the development of web applications [Hatzimanikatis *et al.* 1995]. Currently, there are two major areas of active research in assisting developers in understanding their web applications and maintaining them:

Forward Engineering: which focuses on documenting web applications using specialized specification languages.

Reverse Engineering: which focuses on recovering the structure of web applications from their source code.

In this section, we compare our work to other research which focuses on assisting developers understand their web applications by adapting well studied software engineering techniques.

6.1 Forward Engineering

In [Ceri *et al.* 2000], Ceri *et al*. present the Web Modeling Language (WebML). WebML provides constructs to specify the high level concepts of a web application, enabling developers to specify at a high level their application before they start developing it. In [Conallen 1999], Conallen presents the Web Application Extension (WAE) for the Unified modeling language (UML). In WAE, each web page is modeled as a UML component and each web page has two aspects. A server-side aspect and a client-side one. The server-side aspect is very similar to our work. It shows the web page's interactions with the components that reside on the server. Whereas, the client-side aspect focuses on the page's interaction with the objects and applets that reside on the client's machine. This work is of great value for the maintainers of the application, if the initial developers of the application specified their application using these specification languages. Unfortunately, this is not the case as we saw in the example web application *WebFlight*. Our work provides a tool for

developers to recover the application's specification which is buried deep in the application's source code. Once the architecture is recovered using our approach, it can be written down using UML or WebML.

6.2 Reverse Engineering

Other researchers have recognized the need to assist developers in understanding existing application with no documentation and a large code base. In [Brereton *et al.* 1998], Brereton *et al.* demonstrate a tool which can track the evolution of a web site. Also in [Ricca and Tonella 2000], Ricca and Tonella present a similar tool. Both tools are based on a network crawler which crawls the pages on the web site periodically over a period of time and reports the changes in the pages and the web site. As previously mentioned, our approach uses a directory crawler instead of a network crawler. This technique enables us to track changes in the application's source code, even if these changes are not reflected in the pages viewed by the user. For example, in an earlier version of *WebFlight* users' itineraries were stored in a flat file. In a later version, the itineraries may be stored in an SQL database. In either versions when users view a listing of their itinerary, they would not notice any changes. Clearly, the architecture of the web application has changed. Our approach analyzes the source of the components of a web application. Using this approach, we can study more sophisticated dynamic web applications. Stated differently, we use a white box reverse engineering approach and they use a black box approach.

Alternatively, Antoniol *et al.* suggest a non-automated technique to recover the architecture [Antoniol *et al.* 2000]. The technique is founded on the Relation Management Methodology (RMM), which in turn is based on the Entity Relationship model. Using RMM, the application's domain is described in terms of entity types, attributes and relationships. For example, for *WebFlight* would have entities such as customers, flights, and airlines; and relations such as "buys", and "offers". Unfortunately, this technique is time consuming and can only recover the high level structure of the web application. It focuses more on recovering the main concepts in the design and does not recover the implementation details.

7 CONCLUSION

Maintaining web application is not a trivial task. Developers need tools to assist them in understanding complex web applications. Unfortunately, current tools are implementation focused and current web applications tend to have little documentation.

In this paper, we have shown an approach that can recover the architecture of a web application and show the interactions between its various components. The approach is based on a set of extractors

that co-operate to parse the source code of the application and gather data which is later processed and visualized.

Developers can use these visualizations to gain a better understanding of their application before they embark onto modifying it to add new functionality or fix bugs.

ACKNOWLEDGEMENTS

To validate our approach, we used web applications provided by Microsoft Inc. and Sun Microsystems of Canada Inc. In particular, we would like to thank Wai-Ming Wong from Sun for his assistance in our analysis of the various web applications contributed by Sun Microsystems.

REFERENCES

Antoniol, G., G. Canfora, G. Casazza, and A. D. Lucia (2000), "Web Site Reenginnering using RMM," In *Proceedings of euroREF: 7th Reengineering Forum*, Zurich, Switzerland.

Boldyreff, C. (2000), "Web Evolution: Theory and Practice," Available online at: `http://www.dur.ac.uk/cornelia.boldyreff/lect-1.ppt`.

Bowman, I. T. (1999), "Architecture Recovery for Object Oriented Systems," Master's thesis, University of Waterloo.

Brereton, P., D. Budgen, and G. Hamilton (1998), "Hypertext: The Next Maintenance Mountain," *Computer 31*, 12, 49–55.

Ceri, S., P. Fraternali, and A. Bongio (2000), "Web Modeling Language (WebML): a modeling language for designing Web sites ," In *The Ninth International World Wide Web Conference (WWW9)*, Amsterdam, Netherlands, Available online at: `http://www9.org/w9cdrom/177/177.html`.

Conallen, J. (1999), *Building Web Applications with UML*, object technology, First Edition, Addison-Wesley Longman, Reading, Massachusetts, USA.

Finnigan, P. J., R. C. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. A. Müller, J. Mylopoulos, S. G. Perelgut, M. Stanley, and K. Wong (1997), "The software bookshelf," *IBM Systems Journal 36*, 4, 564–593, Available online at: `http://www.almaden.ibm.com/journal/sj/364/finnigan.html`.

Glass, R. L. (1992), "We have lost our way," *Systems and Software 18*, 3, 111–112.

Hassan, A. E. (2001), "Architecture Recovery of Web Applications," Master's thesis, University of Waterloo, Available online at: `http://plg.uwaterloo.ca/~aeehassa/home/pubs/msthesis.pdf`.

Hassan, A. E. and R. C. Holt (2002), "Architecture Recovery of Web Applications," In *IEEE 24th International Conference on Software Engineering*, Orlando, Florida, USA.

Hatzimanikatis, A. E., C. T. Tsalidis, and D. Christodoulakis (1995), "Measuring the Readability and Maintainability of Hyperdocuments," *Software Maintenance: Research and Practice 7*, 77–90.

Konrad, R. (2000), "Tech Employees Jumping Jobs Faster," Available online at: `http://news.cnet.com/news/0-1007-202-2077961.html`.

Lethbridge, T. C. and N. Anquetil (1997), "Architecture of a Source Code Exploration Tool: A Software Engineering Case Study," Tr-97-07, School of Information Technology and Engineering, University of Ottawa.

PBS (1998), "The Portable Bookshelf (PBS)," Available online at: `http://swag.uwaterloo.ca/pbs/`.

Penny, D. A. (1992), "The Software Landscape: A Visual Formalism for Programming-in-the-Large," Ph.D. thesis, University of Toronto.

Pressman, R. S. (2000), "What a Tangled Web We Weave," *IEEE Software 17*, 1, 18–21.

Ricca, F. and P. Tonella (2000), "Visualization of Web Site History," In *Proceedings of euroREF: 7th Reengineering Forum*, Zurich, Switzerland.

Sim, S. E. (1998), "Supporting Multiple Program Comprehension Strategies During Software Maintenance," Master's thesis, University of Toronto, Available online at: `http://www.cs.utoronto.ca/~simsuz/msc.html`.

Sim, S. E., C. L. A. Clarke, and R. C. Holt (1998), "Archetypal Source Code Searching: A Survey of Software Developers and Maintainers," In *Proceedings of International Workshop on Program Comprehension*, Ischia, Italy, pp. 180–187.

Sim, S. E., C. L. A. Clarke, R. C. Holt, and A. M. Cox (1999), "Browsing and Searching Software Architectures," In *Proceedings of International Conference on Software Maintenance*, Oxford, England.

Sugiyama, K. and K. Misue (1991), "Visualization of Structural Information: Automatic Drawing of Compound Digraphs," *IEEE Transactions on Systems, Man, and Cybernetics 21*, 4, 867–892.

Sugiyama, K., S. Tagawa, and M. Toda (1981), "Methods for Visual Understanding of Hierarchical System Structures," *IEEE Transactions on Systems, Man, and Cybernetics 11*, 2, 109–125.

Tilley, S. and S. Huang (2001), "Evaluating the Reverse Engineering Capabilities of Web Tools for Understanding Site Content and Structure: A Case Study," In *IEEE 23st International Conference on Software Engineering*, Toronto, Canada.

Tilley, S. R. (1999), "Web Site Evolution," Available online at: `http://www.cs.ucr.edu/~stilley/wse/index.htm`.

Tzerpos, V. and R. C. Holt (1996), "A Hybrid Process for Recovering Software Architecture," In *Proceedings of CASCON '96*, Toronto, Canada.

Tzerpos, V. and R. C. Holt (1998), "Software botryology: Automatic clustering of software systems," In

Proceedings of the International Workshop on Large-Scale Software Composition.