**\chapter{Background}**

**\label{chap:background}**

In a short period of time the \textit{World Wide Web} (or the \textit{*Web*}) has gone from complete anonymity as a system used primarily in academia to becoming arguably one of the most significant technological advancements in history\footnote{Technically speaking, the success of the World Wide Web is tightly coupled to the success of the Internet, which is the computer network on which it runs on. Thus, it is the wide spread adoption of both systems that has resulted in the advancements we refer to.}. The Web is now an integral part of our daily lives. It provides us electronic access to a vast amount of information, allowing anyone with a computer and Internet access the ability to publish their own content for the world to see.

In effect, the Web represents a new type of platform for building software applications. We define a \textit{web application} (or simply a \textit{*webapp*}) as a software application that gets executed remotely by a client via a computer network such as the Internet \cite{webapp}. This necessitates the application be divided into two parts: (1) a \textit{client-side}, and (2) a \textit{server-side}. A program running on the client-side makes requests for data to a web server running on the server-side and the server responds by returning the requested data. \textit{Web application development} (or simply \textit{*web development*}) involves building components for both the client-side and server-side.

Over time, web applications have begun to rival traditional desktop applications in terms of popularity and complexity. Indeed, some applications are only possible on the Web. For example, social webapps like Facebook \cite{facebook} and Twitter \cite{twitter} would be impossible to implement without a client-side component and a server-side component. This client-server dichotomy is an essential difference between webapps and traditional desktop applications. Naturally, this difference causes web and desktop applications to have diverging software architectures.

The \textit{user interface} (\textit{UI}) is one key architectural difference between the two application types. A desktop application has its own unique UI which is integrated directly into the rest of the program, but a webapp must deliver its UI in the form of data from the server-side to the client-side. Server-side data is delivered as documents to a web browser on the client-side. The browser program is designed to recognize the document format and present the data to the user. The web browser itself is not part of the web application. However, its responsibility for rendering the UI is absolutely vital.

Different web browsers exist on the market, but each one must implement the same set of standards in order to render server-side data correctly. An international governing body called the World Wide Web Consortium (W3C) was created to develop and maintain standards for building the Web. The W3C created a standard format to define how server-side data is displayed (or rendered) within a web browser. That standard was called the \textit{Hyper Text Markup Language} (\textit{*HTML*}). Consequently, a webapp user interface is a series of HTML documents (or \textit{web pages}) containing textual data. Each document by itself is static and non-interactive. A combination of documents delivered in a sequence, with user input defining which document to display next, produces the interactive experience of the application.

However, this type of interaction alone is primitive. Displaying each new document requires a complete reload of the page which interrupts the user experience and decreases the usability of the application. A webapp appears less like a seamless application and more like a series of documents linked together. Each step of the way the browser window must empty and a new page must be rendered.

This limitation made web applications appear to lack richness and responsiveness when compared to desktop applications \cite{ajax}. It inspired Jesse James Garrett to popularize the Ajax approach for web development. As we shall see later in this chapter, Ajax became a definitive solution to this problem of unresponsiveness. Ajax increases interactivity through JavaScript code which executes based on user input and actively manipulates elements within a static document to alter its content\footnote{Technically, Ajax is composed of a collection of technologies, one of which is JavaScript. However JavaScript is the language that ties everything else together. In this sense, JavaScript is the central technology.}. All modern web browsers contain a JavaScript engine to execute JavaScript code.

In this thesis, we focus on JavaScript programming and user interfaces for web applications. Specifically, we want to improve the developer’s program understanding of the JavaScript code controlling user interface behavior. In this chapter, the primary challenge to program understanding is relating the webapp UI behavior in the browser to the JavaScript implementation code.

We first walk through the anatomy of a web application and show how it has evolved over the years as the Web has matured. We introduce the concept of Ajax and how it produces interactive web applications. We show how JavaScript is essential to Ajax and explain why it is an important language in web application development. We then present an illustrative example of how a web page is created by using all of the client-side technologies involved with web UI development. This is followed by a brief history of the JavaScript language. We point out the strengths and weaknesses of the language and establish why there is a need for JavaScript development tools. We define the type of developer we wish to assist with our research and elaborate on the concept of program understanding. We end the chapter by outlining the current best-of-breed programming tools in the market for JavaScript development and review existing research related to our work in the area of JavaScript code analysis. Our review includes a summary of the work done by Li and Wohlstadter, from which this thesis is based.

**\section{Evolving the Web: Ajax, JavaScript and User Interaction}**

**\label{sec:backgroundWebApp}**

In this section we review the components that make up a web application. We present webapp architecture in two iterations to illustrate the evolution of web applications. Throughout this review, we focus on the impact on the webapp user interface.

As defined in \cite{ComputerNetworks} a web application must have a standard format for defining documents, a web browser, a web server, and a network protocol for client-server communication. Communication is accomplished through the \textit{Hyper Text Transfer Protocol} (\textit{HTTP}) and documents are defined in HTML.

**\subsection{Iteration 1: Classic Web Application Model}**

**\label{subsec:backgroundIteration1}**

\begin{figure}[tb!]

\centering

\includegraphics[width=0.6\columnwidth]{Client-Server-2}

\caption[Iteration 1: A Classic Web Application Architecture]

{The simplest client-server architecture for a web application is a client machine (executing a web browser) and a server machine (executing the web server). The browser makes an HTTP request for content and waits for a response from the server. When the server responds it returns back HTML, which is then rendered in the browser. While the browser is waiting for a response the user sees a blank page in the browser. A multi-tier web application can dynamically assemble web documents by combining static content (images and downloadable files) with dynamic content (persisted in databases).}

\label{fig:clientServer2}

\end{figure}

A web application at its most basic level is a web server that serves static HTML documents to a client machine running a web browser. Figure \ref{fig:clientServer2} shows this simple architecture. In this classic webapp model, the web browser makes an HTTP request for a specific document and the web server fulfills the request by returning an HTTP response containing the desired page. Although HTML documents are static, the content can contain links that reference multimedia, such as images or downloadable files, or other HTML documents.

A multi-tier webapp architecture increases server-side complexity by adding multiple tiers of functionality; each tier being responsible for a separate task. Figure \ref{fig:clientServer2} shows some common tasks, including running business logic, persisting data to a database, and accessing a legacy system. The HTTP server receives all client requests, but may delegate the processing of requests to other tiers. Note that not all the tiers have to be running on the same server machine. Indeed, server-side architectures can be extremely complex in order to address performance and security concerns.

The user can dictate which document to view next by clicking a link referencing another HTML document. This prompts the web browser to request the next document from the web server. A simple HTML construct called a form allows the user to enter data directly into the HTML document and submit it to the server; such as typing a keyword query into a search form. This causes the web browser to send a request containing the user generated data to the web server. Thus, the user is provided with some additional interaction and control over what content a webapp delivers.

All components that are delivered from the web server to the client machine and then executed in the web browser are said to be client-side. Likewise, all components that execute in the web server are said to be server-side (Figure \ref{fig:clientServer2}).

\begin{figure}[tbp!]

\centering

\includegraphics[width=1\columnwidth]{Client-Server-RTT-1}

\caption[Client-Server Communication for the Classic Web Application Model]

{The above sequence diagram shows the client-server communication for the classic webapp model. All communication is synchronous which means that the browser (and the user) must wait while a request is being processed by the server. Once the server sends a response back to the browser, the user can resume using the application.}

\label{fig:clientServerRRT1}

\end{figure}

As noted in \cite{ajax}, the classic webapp model was an attempt to develop software applications based on the Web’s original structure as a system of interlinked hypertext documents. The fundamental problem with this model was each user action required a synchronous request-response communication between the browser and the server. In general, this is because user actions require application logic to determine what to do next, but execution of application logic is done entirely server-side.

As shown in Figure \ref{fig:clientServerRRT1}, each request-response communication between the browser and the server produces a pause in the application. The user is left waiting for a response. During the pause, the user sees a blank page within the browser window and is interrupted from the user experience. Even worse, if the network connection is slow, then a user might also experience a noticeable wait before the page reloads again. This is a terrible setback in user experience. A poor UI leads to a poor application, regardless of how wonderful the software may be as a whole.

**\subsection{Iteration 2: Ajax Web Application Model}**

**\label{subsec:backgroundIteration2}**

The previous two webapp models have neglected the client-side of the web application. The answer to creating a truly interactive UI involves satisfying two important requirements. First, client-server communication should occur asynchronously. This would allow the browser to make a request to the server and not wait for an immediate response. The server-side processing happens in the background while the browser continued execution. This avoids an interruption in the user experience of the application. We define this feature as \textit{asynchronous communication}.

Second, the web browser must be allowed to mutate (or modify) the content of an HTML document after it has been delivered to the client-side. This creates a new set of possible interactions between the user and the web page. If the web page remains static, then asynchronous communication serves no purpose. The user cannot perform any interactions while waiting for server-side processing to finish. We define this feature as \textit{document mutation}. Therefore, both asynchronous communication and document mutation are necessary.

The Ajax webapp model satisfies both requirements. The term Ajax was coined and made famous in 2005 by Jesse James Garrett in his online article entitled ``Ajax: A New Approach to Web Applications’‘ \cite{ajax}. Technically speaking, Garrett did not invent anything new. The technologies behind Ajax existed well before Garrett published his article and the concept of Ajax was already being applied to existing web applications (e.g. Google maps). However, it was Garrett who formalized the technique in a clear and concise manner, gave it a name, and published it for the entire web development community to see. In essence, Garrett made the Ajax approach mainstream.

The term Ajax stands for \textit{Asynchronous JavaScript And XML}. It represents a specific approach for architecting a web application and involves a set of preexisting client-side technologies. These technologies are:

\begin{itemize}

\item{HTML/XHTML for defining semantic structuring}

\item{\textit{Cascading Style Sheets} (\textit{CSS}) for defining formatting and presentation}

\item{\textit{Document Object Model} (\textit{DOM}) for dynamic display and interaction}

\item{XML or \textit{JavaScript Object Notation} (\textit{JSON}) for data interchange}

\item{ \textit{XMLHttpRequest} (\textit{XHR}) for supporting asynchronous client-server communcation}

\item{ JavaScript for binding all of above components together}

\end{itemize}

We have seen HTML in the previous iterations. CSS is a declarative language used to separate document content (as defined in the HTML) from document presentation. Presentation includes aspects like color, font, size and layout but does not involve any dynamic behavior. The DOM is a cross-platform, language-independent specification for representing and manipulating objects in HTML/XHTML/XML. The DOM enables the browser to modify the current page. XML and JSON specify the format in which server data should be returned for asynchronous requests. XHR provides the in-browser functionality to support asynchronous communication with the server. Finally, JavaScript provides the means to connect all the other technologies together. JavaScript is responsible for handling user input events, invoking DOM API functions to manipulate the page, and making asynchronous calls to the server-side through the XHR. Thus, JavaScript is the central language within Ajax.

\begin{figure}[btp!]

\centering

\includegraphics[width=.8\columnwidth]{Client-Server-3}

\caption[Iteration 2: An Interactive Application Architecture.]

{The client-side now consists of: HTML, JavaScript, CSS, the DOM and an XHR engine. Two request-response scenarios are now possible. In scenario (a), the browser makes a \textit{synchronous} HTTP request to the server for content (classic webapp model). The browser must wait for the server’s response. The web server returns HTML with embedded JavaScript and CSS. The browser reloads the page to render the view. In scenario (b), a user action triggers a JavaScript call to the XHR engine. The XHR engine sends an \textit{asynchronous} HTTP request to the server for content. The browser does not wait for the serve’s response. The user can continue to view and interact with the current page. Eventually, the server responds (typically with JSON data). The XHR engine processes the JSON, and JavaScript code then generates HTML and CSS from the JSON. The browser refreshes a section of the current page. A reload of the page is not necessary.}

\label{fig:clientServer3}

\end{figure}

Figure \ref{fig:clientServer3} shows how these technologies are used within the Ajax webapp model. Scenario (a) shows the client-server communication from the classic webapp model. A normal synchronous request is made and the browser waits for a response. In the meantime the user must wait along with the browser. The response from the server will contain HTML, which most likely will contain embedded JavaScript and CSS.

Scenario (b) shows the new alternative client-server communication method, introduced by the Ajax approach. While the current page is loaded within the browser, a user action triggers a call to JavaScript code. Within the JavaScript code an XHR call is made, initiating an asynchronous request to the server. While server-side processing is happening to fulfill the request, the user continues to interact with the currently loaded page in the browser. When the server finally responds it returns data in JSON format. The data is passed through the XHR engine and returned to the calling JavaScript code. At this point the JSON data may have been converted into a fragment of HTML with CSS, which can be inserted into the current web page. Or JavaScript code pulls individual values from the JSON data and updates specific fields on the web page. In either case, the currently loaded web page has been modified without forcing the user to wait and stare at a blank page.

The architecture shown in Figure \ref{fig:clientServer3} provides an accurate model of the current state of web development. As we mentioned in Section \ref{subsec:backgroundIteration1}, the server-side architecture can be far more complex than what has been described. However, since this thesis focuses on the client-side components we only provide a high-level treatment of server-side technologies. An in-depth discussion of server-side technologies is out of the scope for this thesis.

**\section{The Client-Side: JavaScript, HTML, CSS and the DOM}**

**\label{sec:backgroundHTML}**

Having dissected the anatomy of web applications and identified the important role JavaScript has, we now take a step back and look at the client-side architecture as a whole. The combination of HTML, JavaScript, CSS and the DOM is commonly referred to as \textit{Dynamic HTML} (\textit{*DHTML*}). In this section, we present a simple example of how DHTML is used to create an interactive web UI. We use an incremental approach to build our example so the reader can see the role that each of the technologies plays in the UI.

The example begins with a document containing only HTML. The content is composed of a GIF image and a section of text. We then add some presentation code in the form of CSS to improve the style and layout. We then add JavaScript code to introduce a piece of dynamic user interaction to the document.

**\subsection{HTML}**

**\label{subsec:backgroundHTML}**

\begin{figure}[tbp!]

\centering

\includegraphics[width=1\columnwidth]{HTML-Example-1}

\caption[A Simple HTML Document]

{A simple HTML document with no JavaScript code or CSS code.}

\label{fig:HTMLExample1}

\end{figure}

We begin our example with a basic HTML document without any JavaScript code or CSS code. This document is shown in Figure \ref{fig:HTMLExample1} and is named \verb!“example.html”!.

HTML provides a way to define \textit{*structural semantics*} for the textual data within a document. Examples of semantic structure include headings, paragraphs, lists, links and tables. As stated previously, HTML also allows images and other objects, such as a downloadable multimedia file, to be embedded into the document. In addition, one can define interactive forms in HTML, which allow a user to send input data back to the web server. This is all accomplished by wrapping HTML elements around portions of text within the document.

We assume the reader is familiar with HTML and understands how it is used to mark sections of content. Furthermore, we assume the reader knows how the (\verb!<div>!) tag is used to wrap arbitrary blocks of HTML and make them referable from JavaScript using the \textit{*class*} and \textit{id} attributes.

\begin{figure}[tb!]

\centering

\includegraphics[width=1\columnwidth]{HTML-Example-Screenshot-1}

\caption[Screenshot of the HTML Document]

{A simple HTML document with no JavaScript code or CSS code.}

\label{fig:HTMLExampleScreenshot1}

\end{figure}

Finally, Figure \ref{fig:HTMLExampleScreenshot1} shows that our example document does not have any real layout or style when displayed in a web browser (i.e. Firefox), despite having structured content by way of HTML tagging. This is because we have not yet defined any style information for our document, excluding the image size.

**\subsection{CSS}**

**\label{subsec:backgroundCSS}**

We now introduce CSS code to our example document. We will place the GIF image on the left-hand side of the page and the text section on the right-hand side of the page. We will also add some color to the page by adding a primary background color for the body of the document and a secondary for the text section of the document. Finally we will alter the font of the text section to be \verb!“Verdana”! instead of the default \verb!“Times New Roman”!.

CSS is another specification maintained by W3C. Its purpose is to separate the style and formatting of a document from the actual document itself. When the HTML specification was initially created, it was not designed to accommodate presentation concerns such as page layout, font and color. The HTML 3.2 specification introduced tags and tag attributes for defining style, however it quickly became apparent that developing large web sites using the updated specification was impractical. The tight coupling between the textual data of the document and the presentation of the document meant that presentation code was replicated on every single page, even when it was the same values applied repeatedly. Changing a style that was used on all the pages entailed making the same change for each page individually. Consequently, code maintenance was a very expensive process. Using CSS allowed a separation of concerns between the data and the presentation of the data.

\begin{figure}[tbp!]

\centering

\includegraphics[width=1\columnwidth]{HTML-Example-2}

\caption[An HTML Document with CSS code]

{An HTML document with CSS code for style formatting: (a) the HTML code, (b) the CSS code. Some of the HTML code has been omitted for illustration purposes.}

\label{fig:HTMLExample2}

\end{figure}

As shown in Figure \ref{fig:HTMLExample2}, we now have two separate files, namely \verb!“example.html”! and \verb!“example.css”!. The HTML document now contains a \verb!<link>! tag within the \verb!<head>! tag, which was previously empty. This indicates that style information for the HTML document is located within a separate file (example.css). As well, the image width and height attributes have been moved from the HTML document to the new CSS definition file.

CSS syntax is defined by three parts: (1) a selector, (2) a property, and (3) a value. The format looks like this: selector { property: value; }. The selector defines which HTML element or tag will be affected. The property is the name of the attribute that will be modified, and the value specifies the new property value. Thus, to define a font style for all text within \verb!<p>! tags to be \verb!“sans serif”!, the CSS code would be: \verb!p {font-family: “sans serif”; }!. For a given selector one can define an arbitrary number of properties to change, as long as the properties are valid for the given tag based on the HTML specification.

As mentioned in the previous section, \verb!<div>! tags are important for grouping HTML elements together for the purposes of defining common style or common behavior. A \verb!<div>! tag is identified by either a class attribute or an id attribute. In CSS code, a selector for a given class name \verb!“foo”! is defined as \verb!“.foo”!. In CSS code, a selector for a given id name \verb!“foo”! is defined as \verb!“#foo”!.

Figure \ref{fig:HTMLExample2} shows the div with class value \verb!“center\_div”! groups all the text together into a block. In the CSS file, the class \verb!“center\_div”! has the following style attributes:

\begin{alltt}

border: 1px solid gray;

margin-left: auto;

margin-right: auto;

width: 90%;

height: 340px;

background-color: #d0f0f6;

text-align: left;

padding: 8px;

\end{alltt}

The width and height attribute values for the GIF image, which previously were defined directly within the \verb!<img>! tag, have now will moved into the CSS code. The \verb!<img>! tag was specifically identified by the id value \verb!“#image”!.

One can include CSS code directly into an HTML document by wrapping it within the \verb!<style>! tag. However, this is against best practices for web development as it couples the presentation logic with the document content.

\begin{figure}[tb!]

\centering

\includegraphics[width=1\columnwidth]{HTML-Example-Screenshot-2}

\caption[Screenshot of the HTML Document with CSS]

{An HTML document with CSS code for style formatting.}

\label{fig:HTMLExampleScreenshot2}

\end{figure}

Figure \ref{fig:HTMLExampleScreenshot2} shows the document formatted with styles defined in the CSS code. If we created additional documents and wanted to use the same style, we would simply need to reference the \verb!“example.css”! file within the other documents and ensure we use the same class and id values for the \verb!<div>! tags on those pages.

**\subsection{JavaScript and the DOM}**

**\label{subsec:backgroundJSDOM}**

\begin{figure}[tbp!]

\centering

\includegraphics[width=1\columnwidth]{HTML-Example-3}

\caption[An HTML Document with CSS code and JavaScript code]

{An HTML document with CSS code for style formatting and JavaScript code for behavior: (a) the HTML code, (b) the CSS code, (c) the JavaScript code. Some of the HTML code has been omitted for illustration purposes.}

\label{fig:HTMLExample3}

\end{figure}

The final component to add to our page is the interactive behavior, which requires JavaScript code. For our example page, we will add an interactive button that will increase the size of the GIF image when clicked by the user. If the user clicks the button again the GIF will return to its original size. Thus, the JavaScript must maintain state to remember whether the GIF image is currently the large size or the small size.

Figure \ref{fig:HTMLExample3} shows the complete implementation for this page. In addition to \verb!“example.html”! and \verb!“example.css”! we now have \verb!“example.js,”! which contains the JavaScript code. The HTML page includes an additional HTML element within the \verb!<head>! tag. Similar to the \verb!<link>! tag for CSS includes, the \verb!<script>! tag is used to import JavaScript code into an HTML document from a separate external file (example.js). To allow the user to increase the size of the GIF image an \verb!<input>! tag has also been added to the HTML document. The type attribute for the \verb!<input>! tag is set to \verb!”button”!. This instructs the web browser to display the input as a button on the page. The \textit{onclick} attribute for the \verb!<input>! tag is an example of an event handler attribute, as discussed in Section \ref{subsec:backgroundHTML}. It’s value \verb!”changeSize()”! specifies the JavaScript function located within \verb!”example.js”!. The onclick event is triggered when a user clicks on the input button. Once the event is triggered, the JavaScript function changeSize is invoked to handle the event.

Using this event-driven model for associating event handler attributes with event handler JavaScript functions, we are able to provide a means to respond to user actions. Although, we still need to be able to alter HTML documents in order to provide the user with dynamic view of the web page.

\begin{figure}[tb!]

\centering

\includegraphics[width=1\columnwidth]{DOM}

\caption[HTML and the DOM]

{A simple web page is represented in two forms: (a) as an HTML document, and (b) as a DOM tree data structure.}

\label{fig:DOM}

\end{figure}

The DOM specification provides a language-independent, cross-platform outline for how to represent and access objects within a document (HTML, XHTML, or XML). The DOM is designed to be addressable through a programming language and provides an application programming interface (API) to do so. As shown in Figure \ref{fig:DOM} the DOM uses a tree representation of the entire web page. The DOM API provides efficient and precise control over the structure and content of the HTML document by allowing individual nodes to be added, removed, replaced, or altered. Each part of the page is a type of node containing different data. Although it is technically not mandatory, all major web browsers implement the DOM specification, to varying degrees of completeness. Without the DOM, there is no way to inspect the web page and the browser state within a JavaScript function \cite{ProJSWebDev}.

Figure \ref{fig:HTMLExample3}(c) shows the full source code for the function \verb!changeSize()!. JavaScript code executed within a web browser has access to a number of global objects that are made available through the DOM API. One such object is the document object. The first two statements within \verb!changeSize()! make calls to the getElementById function, which is part of the DOM API. As the name implies, getElementById traverses the DOM tree and attempts to locate an element in the HTML document with the passed in id name. In \verb!changeSize()!, we look for \verb!”image”! which is the \verb!<img>! element and \verb!“image\_input”! which is the \verb!<input>! element. Notice that in the HTML, the \verb!<img>! tag has an attribute named \verb!”imgsize”!. This is not one of the standard attributes allowed for the \verb!<img>! tag. It is a custom value we added in order to maintain the state of the GIF image to help determine whether the image is currently the large size or small size. From the HTML document one can see that it is initially set to \verb!”small”!. Within \verb!changeSize()!, we loop through the attributes for the \verb!<img>! element until we find the \verb!”imgsize”! attribute. We then examine the value of the attribute to determine whether the GIF image is currently small or large. If the GIF image is currently small, then we increase its size by using assignment statements to set \verb!style.width! to 225 pixels and \verb!style.height! to 206 pixels. We also change the text label of the input button to \verb!”Shrink”! (since the next time the user clicks the button it will change the image size back to small). If the GIF image is currently large, then we decrease the image size by setting style.width to 107 pixels and style.height to 98 pixels. We also change the text label for the input button to \verb!”Enlarge”!.

We note here that the values 107 pixels for the width and 98 pixels for the height are written once in the CSS code, to define the initial size of the image, and then again in the JavaScript code, to define the application logic to set the image size back to its initial size. This produces a tight coupling between the JavaScript and CSS because the JavaScript is partially responsible for the presentation of the page. As noted in \cite{ProJSWebDev}, current web development best practice is to define all presentation properties (such as the height and width of the image) within CSS code and then have JavaScript code dynamically change CSS class values. This reduces the coupling because the JavaScript code contains fewer lines of code that reference display properties. Our example does not adopt this best practice because the benchmark application we use to evaluate our programming tool later in this thesis does not incorporate this best practice. Thus, we focus on the more primitive style of mutating the page.

\begin{figure}[tbp!]

\centering

\includegraphics[width=1\columnwidth]{HTML-Example-Screenshot-3}

\caption[Screenshot of the HTML Document with CSS and JavaScript]

{Two views of an HTML document with CSS for style formatting and JavaScript for interactive behavior: (a) the page with a small GIF image, and (b) the page with a large GIF image.}

\label{fig:HTMLExampleScreenshot3}

\end{figure}

Figure \ref{fig:HTMLExampleScreenshot3} shows the two states of the web page. View (a) is the initial state when the HTML page first loads within the web browser. The input button has a text label that says \verb!”Enlarge”!. When the user clicks on the button, we are taken to view (b) where the state of the GIF image is now large and the input button says \verb!”Shrink”!. If the user clicks the input button from view (b) we will be taken back to view (a). The most important point here is that during this user interaction the HTML page has not reloaded within the web browser. In other words, the web browser does not need to make an HTTP request at any point during this user interaction. The combination of JavaScript and the DOM API allows our application to modify the static HTML page within the browser and the entire user interaction occurs on the client-side.

One can add JavaScript code directly into an HTML document by wrapping it within the \verb!<script>! tag, like this:

\begin{alltt}

<script type="text/javascript">

// JavaScript goes here …

</script >

\end{alltt}

However, this is against best practices for web development as it couples the behavior logic of the document with the document content. In general, JavaScript code should always be maintained in external files and imported into the document using the href attribute of the \verb!<script>! tag.

**\section{The Front-end Developer}**

**\label{sec:backgroundFrontEnd}**

In the previous section we provided a concrete example of how the client-side components are combined to create an interactive webapp UI. To clarify our subject matter we define a \textit{front-end developer} as a programmer who is responsible for implementing the client-side of a web application. Similarly, we refer to the act of implementing the webapp UI as \textit{front-end development}.

The front-end developer is fluent in JavaScript, HTML and CSS. The \textit{integrated development environment} (\textit{IDE}) consists of the web browser and a text editor that has some level of JavaScript support to make coding easier (e.g. syntax highlighting). The IDE is inherently simple because the JavaScript runtime is embedded within the web browser.

As we saw in the previous section the UI implementation is spread across three different languages\footnote{Technically the DOM API is independent from JavaScript. However in the domain of web development JavaScript is useless without the DOM API and vice versa. Thus we consider the DOM API to be part of the JavaScript for this discussion.}. This presents both an advantage and a disadvantage to front-end development.

The clear advantage being it separates the concerns of defining content, defining presentation and defining behavior from each other. Dividing the source implementation into different sets of files helps decouple the code and reduces code redundancy, since multiple documents can reference the same CSS file or JavaScript file. For example, when one needs to modify the style for a specific block that is present one many different web pages, there is a central location for the style as defined in a CSS file.

\begin{figure}[tbp!]

\centering

\includegraphics[width=1\columnwidth]{Visual-Mapping}

\caption[Mapping Behavior and Presentation of Web Page, from Browser to Implementation]

{A piece of functionality, shown as the GIF image on the web page within the browser, is mapped back to the actual implementation, which includes source code from an HTML page, a CSS file, and a JavaScript file. Note that the id value ``image’‘ ties the HTML with the matching CSS and JavaScript code.}

\label{fig:Visual-Mapping}

\end{figure}

However, spreading the implementation across different languages also creates a major obstacle. The front-end developer must maintain code written in different languages located in separate files, where each file contributes some function to the UI. Debugging or augmenting a specific interaction on a page could potentially involve changes to an HTML file, a CSS file, and a JavaScript file, as illustrated in Figure \ref{fig:Visual-Mapping}. In the example, the page element of interest is an image. It has a corresponding \verb!<img>! tag in the HTML code, a set of CSS properties to define its initial size, and a JavaScript function to change the image dynamically from a small size to a large size and vice versa.

In \cite{fireCrystal}, Oney and Myers point out the languages used to create an interactive web page consist of a combination of imperative and declarative syntax. JavaScript is imperative, while HTML and CSS are declarative. This presents a unique challenge for a developer attempting to understand the implementation, when compared to traditional applications written in a single imperative language, such as Java or C++. It arguably increases the effort required to understand the functional dependency between code written in JavaScript, CSS and HTML.

The only direct connection between the UI behavior (JavaScript code) and the UI content (HTML and CSS code) is through string identifiers. Recall from Section \ref{subsec:backgroundHTML}, class attribute values and id attribute values identifying blocks of HTML content are referenced in the CSS code and JavaScript code. CSS code references the id and class values in order to attach style properties to the corresponding HTML blocks. JavaScript code references the id and class values in order to programmatically mutate the HTML blocks.

We saw an example in Section \ref{sec:backgroundHTML}, where the id attribute \verb!“image”! associated with an \verb!<img>! element on the HTML page was used in both CSS code (to define the image size) and JavaScript code (to mutate the size of the image). Figure \ref{fig:Visual-Mapping} demonstrates this coupling where the id \verb!“image”! is spread across the HTML, CSS and JavaScript code.

As described in Section \ref{subsec:backgroundJSDOM}, JavaScript performs document mutation by calling the DOM API. Whenever JavaScript mutates the page it must query for specific HTML elements using the DOM API. To perform a DOM query, JavaScript must have the correct id or class value.

Mapping an element on the web page to corresponding JavaScript code responsible for mutating it is a time consuming process. Without a programming tool to perform the matching of page elements to JavaScript code, the developer must create a mental model to represent the mapping. Typically, for a given element on the page, the developer will first scan the HTML code to find the corresponding id or class value. Then the developer will search the JavaScript source file(s) for references to the id or class value. This will ultimately locate the correct JavaScript code.

This is a manual process and can be quite time consuming, depending on the developer’s level of JavaScript experience and familiarity with the code base. As the level of complexity within the JavaScript code increases, the level of difficulty in creating and keeping track of the mental mapping between the page elements and JavaScript code increases as well.

In the next section, we formalize define this concept of mapping page elements to JavaScript code and focus on how this affects software maintenance in front-end development.

**\section{Program Understanding and Software Maintenance}**

**\label{sec:backgroundProgramUnderstanding}**

In the previous section we established that our problem domain is front-end development. One major challenge for front-end developers is creating a mental model of the relationship between JavaScript code and the HTML elements which get mutated by the JavaScript code. This obstacle is a problem in program understanding. In this section we formally define program understanding and explore how it applies to software maintenance. We then frame these ideas within the context of front-end development.

We define \textit{program understanding} as the knowledge of how the source code implements the software behavior. This is a cause and effect relationship between the implementation and the functionality; the source code causes the observed effect in the application.

As noted in \cite{programUnderstanding}, program understanding is most important for software projects that involving the modification of an existing system. This process has been given many names over the years. The most common are software renewal, software evolution, program redevelopment, reverse engineering, and software maintenance. According to \cite{programUnderstanding} \textit{software maintenance}, is the most commonly used term. However, Li and Wohlstadter use term reverse engineering in \cite{lw09:insight}. In order to appeal to the broader audience while still remaining compatible with Li and Wohlstadter, we use both software maintenance and reverse engineering interchangeable for the remainder of this thesis.

Corbi argues in \cite{programUnderstanding}, that program understanding is central to software maintenance because adding functionality to an existing system requires that one abide by the existing data and structural constraints. Thus, much of the development effort in software maintenance is spent studying the existing application; to understand how new code will work correctly with old code. On the contrary, one does not need to worry about such restrictions when developing a new application from scratch.

The developer gains program understanding by learning which parts of the source code affect which pieces of behavior in the user interface. The developer achieves understanding by creating a mental model to abstractly represent the underlying software. We define the \textit{UI-JavaScript mapping} as the abstract relationship connecting a piece of JavaScript source code to the resulting behavioral effect on the web page in browser view. If the developer’s program understanding is correct, then the mental model will accurately capture UI-JavaScript mapping of the web application.

We define the \textit{browser view} as the representation of the user interface when it is running within the web browser. Similarly, we define the \textit{code view} as the representation of the user interface when it is in source code form. Therefore, we can say that the UI-JavaScript mapping connects elements in browser view to the JavaScript in the code view.

The learning process for gaining program understanding involves studying the software implementation. One can study the implementation by manually reading the source code \cite{programUnderstanding}. Or one could study the implementation by executing the source code and observing the resulting dynamic behavior \cite{programUnderstanding}. For the latter case, the developer must interact with the UI in browser view, while simultaneously tracing the execution of the JavaScript in code view. Tracing JavaScript execution could be something primitive, such as using print statements to display variable values, i.e. logging. Or tracing could be more advanced, such as using a programming tool with debugging features to step through the code.

The task of discovering the UI-JavaScript mapping is the key to program understanding for front-end developers. It is an activity that is required not only during bug fixing and normal development, but also when a developer joins a new project and must learn how the webapp works. In developer jargon, the process of gaining familiarity with a code base is known as \textit{ramp up}. We believe that ramping up on a project is a very common and important activity in software maintenance.

Our thesis focuses on developing an efficient way to construct the UI-JavaScript mapping for developers to improve their program understanding. We want to present the developer with a visualization of the UI-JavaScript mapping and allow the developer to immediately see the source code driving a piece of behavior in the browser. Productivity is improved by automating this activity within a programming tool so the developer does not have to perform it manually.

**\section{Related Work}**

**\label{sec:backgroundWork}**

As reported by \cite{JSGoodParts}, the current shortage of JavaScript development tools can be explained by one observation: JavaScript is still a relatively young language. It was originally created to perform simple tasks in web pages, and excelled at these tasks where other alternatives such as Java, failed. However, as JavaScript programming evolved, it became apparent the language was quite powerful and could be used to build complex applications. To meet the demand for building JavaScript-intensive web applications, there has been a significant increase in JavaScript software development. Evidence of this is in the sheer volume of technical books published recently focusing on JavaScript design patterns \cite{JSDesignPatterns}, JavaScript frameworks \cite{JSFrameworks}, JavaScript best practices \cite{JSGoodParts,ProJSWebDev} and general JavaScript programming \cite{JSDefGuide,ProJSWebDev}.

In this section, we begin by surveying the general programming tools currently available for front-end developers and comment on how they influence UI-JavaScript mapping and program understanding. Next, we selectively review tools designed to improve program understanding through visual techniques. This is followed by a look at tools that employ code instrumentation to improve JavaScript development. Lastly, we summarize the research done by Li and Wohlstadter and describe how our work is different.

**\subsection{JavaScript Frameworks and Programming Tools}**

**\label{subsec:backgroundJSTools}**

Similar to software development in other languages, large-scale JavaScript applications are not built from scratch. Numerous JavaScript frameworks (or toolkits) have recently appeared to help front-end developers manage complexity and accelerate the implementation process. Some provide reusable JavaScript libraries that handle common programming tasks for the developer, while others introduce architectural principles that impose better development practices.

Prototype was one of the first JavaScript libraries to appear on the market \cite{JSFrameworks}. One of its objectives was to solve cross-browser compatibility issues for the developer. Prototype provides a layer of abstraction on top of differing browser implementations for XHR calls, DOM manipulation, and event handling. Thus, using Prototype’s API means one’s own source code contains less browser-specific logic. Meaning, Prototype offers a solution to creating portable JavaScript code. Another objective was to provide programmers with extensions and functions that were routinely needed during development. Prototype extends the native JavaScript Object to have methods for object serialization and determining data type for objects. Helper functions allow easy traversal and modification of arrays for both JavaScript objects and DOM elements. The native Function object is also extended with useful methods, such as \verb!wrap()!, which allows one to easily inject additional logic around methods to perform important tasks such as logging. It provides functionality without imposing any specific design principles on how developers write their own code. For these reasons and many more, Prototype is a very popular framework.

In contrast to Prototype, Dojo provides a complete packing system for organizing JavaScript code into modules. As we discussed earlier, one of the major drawbacks of the JavaScript language is its dependency on a global namespace for linking code together. Dojo’s packing system provides a solution to this issue. Its namespace mechanism behaves similar to how Java or Python package standard libraries \cite{JSFrameworks}. Dojo started as an effort to consolidate a variety of smaller DHTML toolkits and JavaScript libraries. Thus, it also contains a wide array of utility functions similar to Prototype. Dojo also provides a system for building user interfaces using declarative HTML. Rather than setting up application behavior and writing DOM elements through JavaScript code, Dojo provides a set of Dojo-specific attributes and conventions that can be added to HTML pages to help set up the behavior. None of the above architectural mechanisms are mandatory. But if used correctly they can help to dramatically reduce development time and effort.

Apart from Prototype and Dojo, there are many other valuable frameworks, such as Mootools, JQuery, YUI, and Est JS. Each has its own unique set of design principles and functionality to help accelerate JavaScript development. However frameworks do not directly improve program understanding. Although frameworks help reduce the complexity of the application-specific logic, the developer must still create a mental model of the software in order to understand how it works. Additionally, one must understand how the framework itself works before one can use it effectively. What we really need is a software tool that assists the developer to analyze JavaScript code and build an abstract model of how the implementation works. We continue our survey by examining some existing JavaScript tools.

One example of excellent JavaScript software tool is \textit{JSLint}, created by Douglas Crockford \cite{jslint, JSGoodParts}. It is a code quality tool, which reads JavaScript source code and scans it for potential problems. According to Crockford, problems could be syntax errors, bad programming style, or poor structure. The tool returns an error message and an approximate location within the source for each detected problem. JSLint is intended as a deterrent for badly written code, but does not guarantee the correctness of a JavaScript program. It simply applies a set of best practices for JavaScript programming based on principles defined in \cite{JSGoodParts}.

JSLint’s name is derived from a similar program, called \textit{lint}, which was used in the early days of C programming. When C was still a young language its compilers routinely missed some common programming errors. Thus, lint was used to scan source code and provide an extra level of protection against buggy code. JSLint and lint are both static analysis tools. In essence, JSLint simulates compile-time error checking for a language that does not get compiled. But it is a very useful tool because it imposes a discipline to code writing and directly combats the poor JavaScript design features that encourage bad programming practices. However, it does not directly address program understanding.

Another interesting software tool is Aptana Studio, an IDE solution for front-end development \cite{aptana}. It is an open-source application built on top of Eclipse, which is a popular IDE for Java programming. Aptana Studio includes features such syntax highlighting and code-completion for JavaScript, HTML, CSS, and DOM, JavaScript debugging, integrated documentation (similar to Java Docs), and error/warning notification. By bundling a variety of JavaScript development tools together, Aptana Studio gives developers several options to simplify routine tasks like reading code, writing code, and debugging code. This in turn helps developers concentrate on program understanding. However, Aptana Studio still does not have a tool to directly assist developers model the UI-JavaScript mapping.

**\subsection{Visual Programming Tools}**

**\label{subsec:backgroundVisualTools}**

An obvious approach for simplifying front-end development is to use the What You See Is What You Get (WYSIWYG) technique. This approach allows the developer to work within a view that renders the user interface source code exactly as it will appear to the end-user. In other words, the source code editor provides a view that simulates the browser view.

\begin{figure}[tbp!]

\centering

\includegraphics[width=1\columnwidth]{Dreamweaver}

\caption[A screenshot of Adobe Dreamweaver]

{Adobe Dreamweaver is a WYSIWYG editor for front-end development. It provides a code view and a browser view of the web page. Note that the browser view is called design view in Dreamweaver. In the above screenshot a section of the page is highlighted in both the code and design view. This creates a mapping between the UI and the implementation. Although Dreamweaver is able to produce this mapping for HTML and CSS, it does not for JavaScript.}

\label{fig:Dreamweaver}

\end{figure}

An excellent example of a WYSIWYG editor for front-end development is Adobe Dreamweaver \cite{dreamweaver, dreamweaverAbout}. Dreamweaver has three different views of the HTML page: (1) a code view, (2) a browser view, and (3) a split view. The code view shows the page in the source code form. The browser view, known as design view within Dreamweaver, renders the page visually as it would appear within a browser. The split view shows both the code and design view side-by-side. Figure \ref{fig:Dreamweaver} shows a screenshot of Adobe Dreamweaver 10.0 with an HTML page loaded in split view.

WYSIWYG editors such as Adobe Dreamweaver offer two main benefits in terms of program understanding. First, the developer can see the resulting effect of a piece of code immediately. For example, in Dreamweaver’s split view, saving the source file after writing code will cause a refresh of the design view. When the page is re-rendered in design view it will reflect the code changes. Second, the design view editor allows the developer to visually construct the user interface. For example, in Dreamweaver’s design view, the developer can interactively build a web page by dragging and dropping elements from a set of predefined visual components onto the page. In the design view the HTML code is hidden and the developer only sees the final rendered web page. When the developer inserts a visual component onto the page in design view, the editor automatically inserts the corresponding HTML code into the page. This visual style of programming improves program understanding by allowing the developer to visually observe the mapping between the source code and its effect on the application.

WYSIWYG editors for HTML and CSS editing are quite common. As mentioned earlier, HTML and CSS are declarative languages. Thus they do not contain logic and are static in terms of behavior. Consequently, rendering HTML and CSS code is straightforward.

However, JavaScript is a different matter. It is an imperative language used to define application behavior. It can maintain state information and dynamically change state. It has programming constructs like control-flow and conditionals that allow multiple paths of execution. It involves user interaction in the form of mouse clicks and keystrokes. These characteristics make JavaScript hard to render in a WYSIWYG format. To observe the effect of JavaScript code, one must execute the code and then interact with components on the page in order to see the resulting behavior.

To run the code, the developer must use a web browser or equivalent JavaScript runtime environment. A fully functional web browser must support HTML, CSS, JavaScript, DOM, and XHR. From a feasibility standpoint, it is not practical to implement an environment to simulate a web browser when one can simply use a real web browser. In fact, Adobe Dreamweaver contains a preview feature allowing the developer to view a web page within any web browser.

WYSIWYG editors help program understanding when the developer is creating a new web page with new JavaScript behavior. For example, the split view within Dreamweaver lends well to the incremental progression of building a new UI component. At each step of the development process, one can see immediately how the page looks in design view. And for JavaScript behavior, the developer can load the page in the browser and interact with the page to test the application logic. However, for the developer who is ramping up on an existing project, debugging existing code, or maintaining existing code, the WYSIWYG does not offer assistance in understanding the behavior already implemented.

Another useful JavaScript development tool is Firebug \cite{firebug}. Firebug is an open-source application which integrates into the Mozilla Firefox web browser. Since Firebug runs within the Firefox browser, it has access to in-browser functionality, such as the JavaScript runtime environment. It provides a split view of the web page similar to Adobe Dreamweaver. Although there are some key differences.

\begin{figure}[tbp!]

\centering

\includegraphics[width=1\columnwidth]{Firebug-Inspect}

\caption[A screenshot of Firebug]

{Firebug is a plug-in for the Mozilla Firefox browser. It provides a variety of useful development tools. One of the most popular is its HTML inspection feature, which allows the user to interactively select a section of the page in browser view, and then see the corresponding (1) HTML code, (2) CSS properties, (3) Layout dimensions and (4) DOM properties. In the above screenshot we have inspected the Gmail home page and selected the Gmail logo. For illustration purposes the matching HTML is linked to the Gmail logo.}

\label{fig:Firebug-Inspect}

\end{figure}

First, Firebug’s browser view is the web browser itself. Unlike Dreamweaver, Firebug does not need to make an external call to a web browser to preview the JavaScript behavior. A side-effect of running within the browser is Firebug does not allow the developer to directly edit the source code. What, Firebug allows is modification of HTML, CSS and JavaScript associated with the web page currently loaded in the browser. But, the code loaded into the browser is not the same instance as the original source code. The former is loaded within the browser’s memory, while the latter is located in physical files on the developer machine. Code changes made to the in-browser web page using Firebug are not reflected back in the original source code. Therefore, Firebug is not an actual WYSIWYG editor.

Second, Firebug provides a unique feature called HTML inspection, shown in Figure \ref{fig:Firebug-Inspect}. Once the document is loaded into the browser, the developer can interactively inspect the page and select any HTML element within the browser view. Firebug will dynamically show the corresponding details of that element in a code view. For a selected element, Firebug provides an HTML tab in code view, showing the corresponding HTML code, as well as the CSS properties, the DOM properties, and visual display of the element’s dimensions.

Third, Firebug provides a tab to display JavaScript in code view. The JavaScript tab shows all JavaScript code included for the current page. As mentioned earlier, Firebug has access to the Firefox JavaScript engine and can therefore run JavaScript code in real-time. As a result, Firebug provides a JavaScript debugger allowing the developer to add breakpoints to any of the code included on the page. Once a breakpoint is reached the developer is able to step through the code. Additionally, there is a console tab where the developer may enter arbitrary JavaScript statements and execute them within the context of the current page. This feature provides significant utility for front-end developers.

Lastly, Firebug provides additional tabs for displaying the CSS properties and DOM values for the entire page. Any HTML code, CSS properties and DOM values associated with the page are editable and changes are reflected within the browser in real-time. A net tab is used to show all of the files that the browser loads in order to render the page. This includes the HTML page itself, as well as JavaScript files, CSS files, image files and other media.

According to the usage statistics on Mozilla’s add-on section \cite{addonstats}, Firebug has been downloaded over 19 million times and has an average of 1.7 million daily active users. Firebug’s popularity should come as no surprise considering the numerous valuable features it offers to front-end developers. Be that as it may, the tool still does not directly address the UI-JavaScript mapping problem. The inspection feature is useful for seeing the mapping between the elements on the page in browser view and the corresponding HTML code, CSS properties, and DOM values. However, it does not produce a mapping between the HTML elements on the page and the JavaScript code controling its behavior. On the other hand, Firebug’s JavaScript debugger is another effective tool. Again, the developer must still have a solid understanding the existing code in order to know where to set breakpoints to debug an issue. Thus, the debugger cannot be used to actively acquire program understanding. We shall discuss Firebug further in Chapter \ref{chap:implementation} because our own FireInsight tool integrates into Firebug and leverages Firebug’s HTML inspection functionality.

Lastly, Oney and Myers created a Firefox plug-in tool called Firecrystal as part of their research in \cite{fireCrystal}. Their research objective is identical to ours, which is to improve program understanding for developers by visualizing the mapping within the implementation and the user interface. Firecrystal attempts to accomplish this goal by recording the execution path of user interactions occurring on the page in real-time. For each user interaction the tool captures the corresponding JavaScript code executed, the DOM changes made, and the user events that were triggered. The developer is able to define when Firecrystal begins recording execution and when it stops. The developer can then replay the captured interactions and Firecrystal displays the JavaScript, HTML and CSS code that were affected. Firecrystal has a timeline bar which allows the developer to jump to any point in the recording. This way, it is easier to search for a specific user event if a large number of interactions were captured.

A difference between Firecrystal and our own FireInsight, is Oney and Myers attempt to map page elements to the CSS and HTML code, in addition to JavaScript code. This arguably provides a more refined mapping between the implementation and the user interface than our research. Notwithstanding this fact, we contend that JavaScript code is the most complex portion of the user interface implementation by far. In addition, we are interested entirely in the behavior of the user interface, which is found within the JavaScript code.

Another difference between Firecrystal and FireInsight is our tool actively searches for mappings between the page elements and the JavaScript code. FireInsight focuses on JavaScript code responsible for mutating the page and only records those specific statements. In contrast, Firecrystal records everything and relies on the developer to search through the timeline for specific user events that are interesting. This is a key difference in our methodologies. We believe the most important part of a user interaction is when the page is mutated to update the UI. Thus we assume the developer is most concerned with JavaScript code that is mutating the page. Once the developer understands the mapping between the page elements and the JavaScript code that mutates them, it is straightforward to examine the surrounding logic.

FireInsight also attempts to group JavaScript mutation statements together in semantically meaningful ways. The assumption being that a single user interaction will commonly produce a series of page mutations. Therefore, it is semantically meaningful to group all the JavaScript mutation statements responsible for a single user interaction together, to visually represent how the elements relate to each other.

**\subsection{JavaScript Instrumentation}**

**\label{subsec:backgroundInstrumentation}**

JavaScript is executed using the load and go delivery method. Because JavaScript is delivered as-is in text form, it presents a challenge in analyzing source code. One solution takes advantage of JavaScript’s dynamic nature. Since JavaScript allows us to attach arbitrary functions and objects to existing functions or objects, we can write our own analysis code in JavaScript and then inject it into the original source code. However, we would like our code to be non-intrusive, by not altering the original application logic. Specifically, we would like our tool to remain transparent to the application. This can be accomplished by intercepting the JavaScript files before they are delivered to the web browser. The intercepted files are then injected with our analysis code and sent to the web browser. Once everything is loaded in the browser, our analysis code will be executed along with the original source code. This technique is known as \textit{JavaScript instrumentation} \cite{coreScript, ajaxScope, ajaxView}.

In \cite{coreScript}, Kikuchi et al. propose a security framework called CoreScript for preventing web browsers from executing malicious JavaScript code. The framework enforces security by instrumenting the original JavaScript code before it is delivered to the client browser. CoreScript uses an HTTP proxy program as a gateway between the client machine and the Internet. This allows them to have full control over what JavaScript code is delivered to the client browser. Within the proxy they use an instrumentation framework to alter sections of the original code to enforce security. Their instrumentation framework is configurable; one can define rules for what JavaScript statements get modified and how those statements are modified.

Although CoreScript and FireInsight share a similar architecture for instrumenting JavaScript, they have considerably different goals. Kikuchi et al. are concerned with enforcing security constraints on the JavaScript code executing in the client browser in order to protect end-users. In contrast, we are interested in providing developers a programming tool to improve their program understanding.

Kiciman and Livshits present another interesting instrumentation system. In \cite{ajaxScope, ajaxView}, they describe a JavaScript instrumentation framework for performance monitoring and profiling, called Ajax View (previously known as AjaxScope). Again the setup involves using a proxy sitting in-between the web server and the client browser. As data is delivered from the web server to the client, JavaScript files are captured by the proxy and injected with additional code. The injected code is executed normally with the rest of the original source code in the client browser. This instrumented code generates performance metrics, call graphs, information about application state and user interactions to help provide a full picture of the client-side user experience. The analysis data is sent back to the proxy. The proxy is implemented as a plug-in for Microsoft’s Internet Information Server, which is a web server technology similar to Apache Tomcat, JBoss, and IBM WebSphere.

Similar to CoreScript, Ajax View is designed to be configured so developers can evolve the kinds of analytical information being captured on the client machines. Ajax View can deliver different instrumented code to different client machines in real-time. Since instrumented code is executed on client machines it is important to avoid significant degradation of performance for each end-user. Ajax View has a variety of analysis code that can be executed to gather monitoring and profiling data. Running all the analysis on each client machine could potentially lead to significant performance slowdowns. The best approach is to divide the analysis tasks into modules and spread the instrumentation code across the end-users. Consequently, the burden of gathering performance data is shared equally and fortunately the overall amount of monitoring code running on each client machine is small. Code reuse for this project is promoted in the form of pluggable instrumentation policies used within the rewriting engine. These policies are also reused for other web applications.

Like Ajax View, our tool injects instrumentation code into the existing JavaScript files to analyze the original application logic. However, Ajax View uses its instrumentation code to gather performance metrics and detect bugs in the original JavaScript source. In contrast, FireInsight attempts to gather semantic data on how the original JavaScript source maps to the mutation of page elements. This semantic data is then processed and displayed within the client browser.

Lastly, JSCoverage is a JavaScript instrumentation framework designed to determine the code coverage of a JavaScript application \cite{jscoverage}. Specifically, JSCoverage performs analysis on how many lines of source code are executed and how many are not. The tool provides a number of ways to carry out the instrumentation. The tool can read JavaScript files from a specified source directory, instrument the code and write the modified files to a destination folder. The modified files can then be opened in a browser to execute the analysis. The tool also provides a server program which can serve HTML pages referencing JavaScript files. The server instruments JavaScript code before delivering it to the web browser. Once the code is loaded in the browser it begins running its analysis. The server may also be configured to run as a proxy. In the proxy mode, instrumentation is performed in a similar fashion to CoreScript, Ajax View, and FireInsight.

Code coverage analysis contributes to program understanding because it assists the developer in verifying if the application logic behaves as expected. Increases in code complexity also exacerbate the difficulty of verifying code correctness. Although code coverage analysis cannot guarantee code correctness, it can help determine which areas of the code are executing regularly and which areas are left untouched. This information is important when designing tests to run against the application.

Although code coverage analysis can enable the developer to verify application code is in fact running correctly, it does not address the UI-JavaScript mapping problem. Thus, JSCoverage and FireInsight do not have the same objective.

**\subsection{Script Insight}**

**\label{subsec:backgroundScriptInsight}**

Having seen a variety of research, programming tools, and programming frameworks centered on JavaScript development, we now focus on the work that has directly influenced this thesis. In \cite{lw09:insight}, Li and Wohlstadter introduce a novel approach for determining the UI-JavaScript mapping in a web application, and prototype implementation called \textit{Script Insight}.

They propose the most intuitive way for the developer to explore the user interface and gain program understanding, is to interact with the page in the browser view. From the browser view, the UI is easy to understand and semantically meaningful. Conversely, the code view of the UI is hard to understand, and identifying semantic structure is time consuming. Recall from Section \ref{sec:backgroundProgramUnderstanding}, the browser view represents the user interface from the end-user perspective as it appears in the web browser, while the code view represents the user interface in the form of source code.

Therefore, when the developer is attempting to understand how a piece of the UI behavior works, the best place to begin is in the browser view. Once the developer has located the module of interest on the page in browser view, the next step is to jump to the corresponding JavaScript code responsible for implementing the behavior. Li and Wohlstadter suggest that this scenario represents the most efficient way to explore the UI behavior and also captures the UI-JavaScript mapping.

The developer must interact with the page and trigger the desired behavior. This will cause JavaScript code to execute and mutate the page. During this process, Script Insight will capture the JavaScript statement responsible for mutating the page. However, recording the JavaScript mutation statement alone is not enough, because it may be executed in multiple scenarios for separate behaviors. Thus, it is important to capture the entire execution context. We define \textit{execution context} as the runtime data structure storing all of the active functions invoked to arrive at the current execution point in the program. The execution context is also commonly known as the call-stack, execution stack or function stack. This call-stack is uniquely identified as the context from which the JavaScript statement was executed.

A single user interaction can potentially involve a sequence of changes to the page, instead of a single mutation. When a user event is triggered, a corresponding JavaScript function will handle the user action and execute some application behavior. We define an \textit{event handler} as a JavaScript function registered to respond to a user event. An example of a user event is the mouse clicking a button on the page. An event handler that executes multiple JavaScript statements will produce a sequence of execution contexts. When the developer is interested in the behavior for a user event, each execution context in the sequence is important. Li and Wohlstadter suggest that this sequence of execution stacks can be visualized as a control-flow graph, which they refer to as a \textit{DOM mutation graph} (or \textit{DMG}). They argue that presenting an event handler as a visual graph helps the developer see the UI-JavaScript mapping and gain program understanding.

Our work differs from Li and Wohlstadter primarily in its implementation. We create new programming tool called FireInsight, which contains all of the functionality described in \cite{lw09:insight}. However, our tool is more than a prototype. It is an implementation that integrates with existing software on the market, such as Firefox and Firebug. The UI for our tool consists of a fully functioning Firefox plug-in. We leverage Firebug’s existing HTML inspection feature to allow the developer to explore web pages in browser view. It is unclear in \cite{lw09:insight} how Script Insight implemented its inspection mode in the web browser or its DMG functionality. In contrast, we explicitly show step-by-step how FireInsight allows the developer to visually inspect a web page and dynamically generate DMGs based on the executing JavaScript behavior. We also present some technical challenges associated with identifying event handlers for the DMG, which were not addressed by Li and Wohlstadter.