**\chapter{Introduction}**

**\label{chap:introduction}**

Web applications have evolved dramatically over the past decade. One component of the web application which has seen significant technical advancements is the user interface (UI). The widespread adoption of web programming specifications such as JavaScript and the W3C Document Object Model (DOM), gave developers the ability to create increasingly sophisticated UIs. As a result, web applications can now provide seamless and highly interactive user experiences, similar to desktop applications. Naturally, this additional interactivity has led to a corresponding rise in the UI’s complexity. Building and maintaining rich web UIs now require a greater development effort. Because JavaScript is the language of the web browser, most of the development effort is spent on JavaScript programming.

As the language matured, the development community has produced numerous frameworks, programming tools, best practices, and design patterns to help make JavaScript applications easier to build. Frameworks, such as Prototype and Dojo \cite{JSFrameworks}, provide reusable JavaScript libraries as well as a system for architecting the JavaScript code. Best practices, such as decoupling JavaScript code from presentation code (CSS), creating build and deployment procedures for JavaScript, and compressing JavaScript code for production environments, provide a means to manage complexity and improve the performance of the UI \cite{ProJSWebDev}. Software, such as JSLint \cite{jslint} and Aptana \cite{aptana}, provide invaluable assistance to help developers avoid the weaknesses of the language. Other tools such as Firebug \cite{firebug} and Dreamweaver \cite{dreamweaver}, enable developers to program the UI using visual and interactive techniques.

**\section{Problem and Motivation}**

**\label{sec:introProblem}**

However, there is currently very little research or progress in the area of JavaScript program understanding. Program understanding represents the knowledge of how the source code implements an application’s behavior. Within the domain of UI programming, this knowledge corresponds to an abstract mapping that connects the source code to the actual UI behavior that occurs in the browser.

Program understanding is crucial for software maintenance. Many of today’s most popular interactive web applications, such as Google Maps, Facebook, Twitter, and Gmail, have constantly evolving UI components. Because these interfaces are implemented in JavaScript, fixing and augmenting existing JavaScript code are common activities within web development. Many software projects face constantly changing user requirements and thus require significant investments in software maintenance.

In addition, for new developers joining an on-going project, program understanding is vital. In order to work efficiently on the project, a new developer must quickly ramp up on the implementation and understand how to modify the system or build on top of the existing code. This situation could occur either during the software maintenance phase or during the initial implementation phase of a project. There are other cases where a developer on one project may review the source code from another project in an attempt to understand the implementation. The purpose of this activity is to transfer functionality from one application to another.

In each of the above scenarios, program understanding is central to the success of the project. Acquiring program understanding in the domain of JavaScript programming is a manual process. As mentioned above, gaining an understanding of the UI behavior involves mapping the interactions observed in the web browser back to the JavaScript source code. For bug fixing or feature enhancements the developer must first replicate the appropriate UI behavior in the browser. Then the developer must trace the execution back to the underlying source code. For web UI development, this involves tracking HTML elements on the web page back to JavaScript source code. The only real connection between HTML and JavaScript are \verb!class! and \verb!id! attribute values. These values identify sections of HTML on the page and are also referenced in JavaScript to mutate the page. Depending on the developer’s level of JavaScript experience and familiarity with the application, mapping the behavior and source code could be relatively straightforward, or it could be extremely time consuming and arduous.

JavaScript instrumentation has become a common method to perform code analysis for an application. This technique involves injecting additional JavaScript code into the existing application logic before it is delivered to the client browser. JavaScript code files are intercepted by a proxy, injected with the analysis code, and then sent to the client. The injected code then performs meta-analysis on the application logic as it is executing in the browser. Kikuchi et al. propose a JavaScript instrumentation framework in \cite{coreScript}, called CoreScript, to enforce security constraints. The injected security code controls which portions of the original application logic get executed based on security rules, which are configured within the framework. In \cite{ajaxScope, ajaxView}, Kiciman and Livshits use JavaScript instrumentation to perform performance monitoring and program profiling. Their framework, called Ajax View, injects analysis code into the application logic to gather data regarding the performance of the UI on client machines. Ajax View also monitors the program execution for common JavaScript issues such as memory leaks and reports this data back to the server. Lastly, JSCoverage is an open-source framework for measuring code coverage for JavaScript applications \cite{jscoverage}. It uses instrumentation to inject analysis code to determine which lines of code get executed and which do not.

Each of aforementioned instrumentation frameworks perform code analysis on JavaScript applications. They assist the developer in managing various aspects of the application such as security and performance monitoring. However none of them directly address program understanding.

In \cite{fireCrystal}, Oney and Myers developed an approach to assist program understanding by recording UI behavior in real-time as the developer interacts with the page. The developer can then replay the recordings from a timeline and see the corresponding JavaScript and other related code that were invoked during a given UI behavior. This technique is helpful but is still potentially time consuming if the developer does not know what to look for during the playback of the behavior.

In \cite{lw09:insight}, Li and Wohlstadter developed a model-based approach to help program understanding. They suggested that the key pieces of JavaScript code that relate back to the UI behavior are statements which mutate (or modify) the elements of the page (DOM). Since these statements result in actual changes to the DOM, they represent transitions during the UI behavior. A user interaction begins with a user event, such as a mouse click on a button, which then triggers an ordered sequence of these DOM mutations. Li and Wohlstadter reasoned that an intuitive representation for a sequence of DOM mutations is a control-flow graph. Constructing this graphical representation would then capture the abstract mapping between implementation and behavior that is central to program understanding.

We propose an interactive, model-based technique to improve developer understanding of the underlying UI software inspired by the research of Li and Wohlstadter. We implement our methodology as a programming tool that integrates with and leverages features from the Mozilla Firefox web browser, as well as the Firebug development tool. We use a proxy to perform JavaScript instrumentation on the application logic and gather meta-data on DOM mutation. Our tool then uses the analysis data to map the relationship between the JavaScript code and the UI behavior. We evaluate our tool using a benchmark web application, called Java Pet Store 2.0.

**\section{Contributions}**

**\label{sec:introContributions}**

Our objective was to create a programming tool to improve program understanding of JavaScript behavior within the web application user interface. Our tool allows developers to explore JavaScript code by visually interacting with the user interface running inside the browser. We achieved our objective by adopting the research presented by Li and Wohlstadter in \cite{lw09:insight} and expanding upon their work in a number of meaningful ways.

The most important contribution is our software tool itself, which we call FireInsight\footnote{The name “FireInsight” is based on a combination of Firebug and Script Insight. Firebug extensions commonly have names with the “Fire” prefix. We have adopted this convention in naming our software tool since it is an extension to Firebug \cite{firebugExtensions}.}. Using the ideas proposed by Li and Wohlstadter, we have created a programming tool that is a clear improvement upon their prototype application, called Script Insight. Enhancements were made in two key areas.

The first area of improvement is interoperability with existing programming software. As a program, FireInsight is more interoperable with existing web development tools than Script Insight. We accomplished this by developing our program as an extension to Firebug \cite{firebug}, a popular web development tool built on top of the Mozilla Firefox web browser \cite{firefox}. Firefox and Firebug are both widely used throughout the web development community. Since FireInsight runs within Firebug, it encourages web developers to readily adopt our tool. Additionally, we were able to leverage features within Firebug to help implement our feature set. For example, we leverage the inspect HTML feature from Firebug to implement the page inspection mechanism within our own tool.

The second area of improvement is functionality. While Li and Wohlstadter proposed a graphical representation of JavaScript code called the DOM Mutation Graph in \cite{lw09:insight}, they did not implement it as a feature within Script Insight. In contrast, FireInsight is able to display the DOM Mutation Graph dynamically as the developer explores and interacts with the user interface. Please refer to Chapter \ref{chap:implementation} for further details.

Another contribution to the research is our critique of the methodology behind Script Insight and FireInsight. As a result of building FireInsight from the ground up, we were able to see first-hand how each component of the tool was implemented. We experienced the same technical challenges as Li and Wohlstadter in building a system to analyze JavaScript source code. Thus, we provide a critique of their approach as a whole based on a number of criteria. We examine the approach in terms of program understanding, interoperability, and reliability. The review can be found in Chapter \ref{chap:results}.

Finally, though we evaluated Fire Insight using the same benchmark web application (Java Pet Store 2.0) as Li and Wohlstadter, we provide a more thorough analysis. They used a single in-depth example to illustrate how Script Insight improves developer understanding of JavaScript code. In contrast, we present a comprehensive series of examples that cover a variety of JavaScript-enabled functionality from Java Pet Store 2.0.

**\section{Thesis Outline}**

**\label{sec:introOutline}**

The remainder of this thesis is organized as follows. Chapter \ref{chap:background} reviews the evolution of web application development and the importance of JavaScript with respect to interactive user interfaces. The chapter also defines our problem domain and presents related work in the area of JavaScript programming tools. Chapter \ref{chap:methodology} formalizes our research problem and delves into the methodology we used to solve it. Chapter \ref{chap:implementation} describes how we implemented our methodology in the form of a programming tool, called FireInsight. Chapter \ref{chap:results} provides a comprehensive evaluation of FireInsight using the Java Pet Store 2.0 application and discusses the results. Finally, Chapter \ref{chap:conclusions} presents our conclusions and discusses possible future work for our research.