**\chapter{Design and Implementation}**

**\label{chap:implementation}**

Chapter \ref{chap:methodology} formally defined our research problem and outlined the methodology for our solution. Our objective is to model the connection between the UI behaviors and the JavaScript source code that implements those behaviors. Our UI-JavaScript model is accurately represented by the causal relationship between DOM mutators and the DOM elements they mutate. Successfully constructing and visualizing this model should help improve the developer’s program understanding.

The most important information to capture about DOM mutators are their call-stacks. In addition, the most semantically meaningful way to model DOM mutators is to group them based on the event handler that is responsible for invoking them. Visualizing an event handler’s DOM mutators as a control-flow graph provides insight into the order in which DOM mutations are executed and insight into how this affects the state of the web page.

Our solution centers on creating a programming tool to automate this modeling process, called \textit{FireInsight}. Chapter \ref{chap:methodology} introduced a number of key feature requirements for FireInsight.

The first requirement is the developer can inspect DOM elements on the page. The developer can interactively navigate the page and select a DOM element in the browser view. Selecting an element immediately displays the corresponding DOM mutators, if any exist\footnote{Not all elements on the page are interactive. A web page will commonly have a portion of elements that mutate as a result of JavaScript behavior and a portion that are static. The exact percentage of elements that are mutable and interactive depends on the web application and its user interface.}. Each DOM mutator is displayed as a call-stack, where the top level of the stack is the JavaScript statement corresponding to the DOM mutation. Selecting any of the entries in the call-stack will take the developer directly to the corresponding source code file, with the exact line highlighted.

The second requirement is the developer can see which event handlers affect a selected DOM element. For each event handler, the developer can choose to view a control-flow graph representation called a DMG. An event handler’s DMG displays the DOM mutators as an ordered sequence of nodes, where the edges between nodes indicate the control-flow from one mutation to the next. The edges are directed, which means they have a clear one-way direction. For example, if a node \verb!i! has an edge pointing to a node \verb!j!, then DOM mutator \verb!i! is executed first, followed by DOM mutator \verb!j!. If the developer clicks on a particular node in the DMG, the call-stack will be displayed. From there the developer can choose to view the source code.

In this chapter we present a comprehensive review of our implementation details. We begin by providing an overview of the architecture for FireInsight. We list some technical assumptions used to simplify FireInsight’s implementation. We then examine each software component individually. For each component, we mention the feature requirement which it satisfies, any technical challenges we encountered, and where applicable, we use screenshots of FireInsight to help illustrate our implementation. We end the chapter with a brief summary of the known technical limitations of our programming tool.

**\section{FireInsight Architecture Overview}**

**\label{sec:implementationArchitecture}**

\begin{figure}[tbp!]

\centering

\includegraphics[width=0.8\columnwidth]{Proxy-2}

\caption[FireInsight Architecture]

{FireInsight’s architecture has two software components, which are highlighted above. The first is a plug-in for the web browser, which provides the user interface for our tool. The second is an HTTP proxy, which is responsible for instrumenting JavaScript source code being delivered to the client-side.}

\label{fig:Proxy-2}

\end{figure}

FireInsight’s architecture contains two software components, both of which are necessitated by the feature requirements. Figure \ref{fig:Proxy-2} shows the architecture of our tool and illustrates how it relates to the overall web application system. We will briefly describe the components in this section and leave the exact implementation details for Sections \ref{sec:implementationFirebug} and \ref{sec:implementationProxy}.

The first component is a plug-in for the web browser. FireInsight’s feature requirements dictate that we provide the developer with a browser view of the web page. Due to the complexity of implementing a browser view from scratch, we decided to integrate our tool into an existing web browser. This provides us with the full capabilities of the browser to render web pages and execute JavaScript behavior. As shown in Figure \ref{fig:Proxy-2}, we chose to integrate our tool with the Firefox browser by creating a plug-in application.

Specifically, the FireInsight plug-in application integrates into Firebug, which itself is an add-on to Firefox. In this way we leverage the functionality provided by both Firefox and Firebug. Further details about the Firebug plug-in and the reasons for choosing Firefox in Section \ref{sec:implementationFirebug}.

The second component is an HTTP proxy. FireInsight’s feature requirements dictate that we must record execution call-stacks whenever a DOM mutator gets executed. This allows us to build a history of all DOM mutators that get invoked on the page, which is then used to generate the DMGs. To capture the call-stack data as JavaScript behavior is executing in real-time, we must create our own JavaScript code and hook it into the application logic. We accomplish this using JavaScript instrumentation. Our HTTP proxy performs the instrumentation on all JavaScript source files which are delivered to the web browser.

JavaScript instrumentation is a non-intrusive technique to inject additional code into existing application logic. The technique requires parsing JavaScript application code. We inject code that records call-stack information whenever DOM mutators are executed. Our code also determines the function names of event handlers and groups execution contexts by event handler. Further details about the proxy is in Section \ref{sec:implementationProxy}.

**\section{Implementation Assumptions}**

**\label{sec:implementationAssumptions}**

We make a number of assumptions regarding how FireInsight will be used. These assumptions help reduce the complexity of our implementation and also simplify our evaluation process in Chapter \ref{chap:results}.

Our first assumption is front-end developers will use our tool to inspect development-level code, not production-level code. This is an important distinction because the code-level affects the formatting of the source code significantly. Current best practices for JavaScript development require that production-level code be compressed and obfuscated \cite{ProJSWebDev}. This decreases the download time for JavaScript files and acts as a safeguard by deterring hackers from interpreting the source code.

When FireInsight creates the UI-JavaScript mapping, it provides a link back to the source code. The developer can click the link and be taken to the correct source file and line number for a given JavaScript statement. However, FireInsight depends on nicely formatted and commented source code. Otherwise, the source will be impossible to read and understand. Therefore, we assume the developer will use our tool in a development environment where the JavaScript code is not compressed or obfuscated, and is well formatted.

Our second assumption relies on the front-end developer having access to the original JavaScript source files being delivered to the browser. In other words, the developer has the JavaScript source code on a local machine, preferably on the same machine that is running the client-side of the webapp. This way, it is straightforward for the developer to locate and edit the source code, after using FireInsight.

This stipulation is important because FireInsight, like Firebug, does not provide a WYSIWYG editor.

Since the JavaScript loaded into the browser is a separate copy of the source code, changes made to it will not get reflected back in the original source files. Firebug provides the ability to edit the HTML, CSS, and DOM properties on the client-side, and have those changes reflected in the browser view of the page. However, the changes only affect the in-browser version of the page, not the original source code. FireInsight currently does not provide the ability to edit the JavaScript in the browser.

**\section{Firefox, Firebug and FireInsight}**

**\label{sec:implementationFirebug}**

In order to implement a browser view for FireInsight, we decided to use an existing web browser. We chose to integrate it with the Mozilla Firefox browser because of its large and active open-source development community. Another motivation for using Firefox was Firebug, which is an extremely useful add-on appliction. In fact, its HTML inspection feature is the inspiration behind the page inspection functionality for our own FireInsight. The HTML inspection feature allows the developer to inspect elements on the current page and immediately see the corresponding HTML code, as well as its CSS and DOM properties. Because the inspection mechanism is visual and interactive, the developer can easily see the association between the page element (in browser view) and its HTML representation (in code view). Since Firebug already has this excellent inspection mechanism, it was an obvious choice to integrate our programming tool into Firebug and leverage its HTML inspection feature.

\begin{figure}[tbp!]

\centering

\includegraphics[width=1\columnwidth]{Plug-in}

\caption[FireInsight Plug-in]

{The FireInsight plug-in component acts as the user interface. The above diagram depicts how our tool integrates with the Mozilla Firefox browser and the Firebug add-on application. (1) The developer inspects the page and selects a DOM element. (2) Firebug automatically switches to its HTML tab and shows the corresponding HTML code for the selected element. (3) Our FireInsight plug-in operates as a side-panel within Firebug. It was three different views. (4) The Attribute view shows all DOM mutators that change attribute values for the selected DOM element. (5) The DMG view shows the DMG representation of a selected event handler. (6) The Event Handler view shows a list of the event handlers that mutate attributes belonging to the currently selected DOM element. }

\label{fig:Plug-in}

\end{figure}

Our FireInsight plug-in was developed on Firefox 3.0 and Firebug 1.4, respectively. Figure \ref{fig:Plug-in} shows how our plug-in integrates with both Firefox and Firebug. Firebug has access to all of the browsers capabilities, such as the JavaScript runtime environment. Similarly, our plug-in runs within Firebug and thus has access to all of Firebug and Firefox’s capabilities.

Firebug also has multiple tabs, one of which is the HTML tab. When the developer activates the inspection mode in browser view, Firebug automatically switches to the HTML tab. While the developer is in HTML inspection mode, any DOM element that the mouse hovers over will cause the HTML tab to display the corresponding HTML code. This is shown in Figure \ref{fig:Plug-in} (1) and (2). Once the developer clicks the mouse on an element on the page, HTML inspection mode gets deactivated. The HTML code remains on the currently selected element. At this point the developer can view the corresponding CSS, layout dimensions and DOM properties for the selected element, which are located in the side-panel. With this inspection mode the developer can quickly and intuitively learn how the presentation of the page is implemented.

We add to the HTML inspection mode by presenting a new side-panel to display JavaScript related code for a selected DOM element. The side-panel is added to the existing HTML tab. This is shown in Figure \ref{fig:Plug-in} (3).

Our plug-in has three distinct views. The first lists all the attributes belonging to the currently selected DOM element and a corresponding DOM mutator in the JavaScript code (Figure \ref{fig:Plug-in} (4)). We call this the \textit{Attribute} view. From the Attribute view the developer can click on an attribute and see the execution call-stack. The developer can then click any of the entries in the call-stack and see a pop-up window displaying the source code with the exact JavaScript statement that mutates the attribute highlighted.

The second view displays the DMG for a selected event handler (Figure \ref{fig:Plug-in} (5)). We call this the \textit{DMG} view. Recall that a DMG is a directed graph of DOM mutators, which are all invoked by the same event handler. In the Figure \ref{fig:Plug-in} (5), we see an example DMG containing five DOM mutator nodes\footnote{The first node in our DMGs is not a DOM mutator. It indicates the beginning of the control-flow and always appears in the DMG.} Note that the DMG also contains a cycle. Meaning the sequence of the first three nodes can repeat an arbitrary number of times. For each node in the graph the developer can right-click on a node and see the corresponding call-stack, just like in the Attribute view. From there the developer can click on any entry in the call-stack and see the source code.

The third and final view displays a listing of event handlers which have altered one or more attributes belonging to the currently selected DOM element (Figure \ref{fig:Plug-in} (6)). We denote this as the \textit{Event Handler} view. So for each event handler in the Event Handler view, they will have invoked at least one of the DOM mutators listed in the Attribute view. The developer can double-click on any of the event handlers and be taken to the DMG view, where the corresponding graph is displayed.

The FireInsight plug-in defines its components using the Mozilla XML User Interface Language (XUL), which is an XML-based declarative language for building cross-platform applications \cite{mdc}. All Firefox add-on applications, including Firebug, use XUL.

**\subsection{DOM Mutation Graph and MxGraph}**

**\label{sec:implementationDMG}**

To display DMGs within our plug-in we use a third-party JavaScript library called MxGraph \cite{mxgraph}. The library provides a framework for building browser-based interactive drawing and diagramming applications. When the developer selects an event handler from the Event Handler view, the plug-in switches to the DMG view and loads the MxGraph code. Using the MxGraph library we construct a new graph object based on the set of DOM mutator call-stacks that have been recorded for the currently selected event handler.

Many times a single execution context will be invoked repeatedly. This is recorded as a long series of DOM mutators. In order to reduce the number of execution contexts we display and present only the semantically meaningful transitions between DOM mutators, we represent repeated consecutive executions of the same DOM mutator with a single graph node. One execution context is considered a repeat of another if they share the exact same call-stack.

Once we have reduced the original sequence of mutator contexts into a sequence of unique nodes, we can construct the graph. We set an edge between each pair of unique nodes within the ordered sequence. For each node we label it with the name of the attribute that was mutated. To indicate start of the control-flow, we always insert an unlabelled start node.

**\section{JavaScript Instrumentation using an HTTP Proxy}**

**\label{sec:implementationProxy}**

\begin{figure}[tbp!]

\centering

\includegraphics[width=0.8\columnwidth]{Proxy-3}

\caption[FireInsight HTTP Proxy]

{FireInsight’s HTTP proxy component intercepts all communication between the web browser and the web server. The diagram depicts the scenario when a JavaScript file is requested by the web browser. (1) An HTTP request from the web browser is intercepted by the proxy. (2) The proxy creates a new HTTP request and sends this to the destination server. (3) The server processes the proxy’s request and sends back the corresponding JavaScript file. (4) The proxy determines whether or not the file contains JavaScript and uses the Rhino framework to parse the source code. (5) Our analysis code is injected into original application code. (6) The resulting instrumented JavaScript file is set into an HTTP response and returned to the web browser.}

\label{fig:Proxy-3}

\end{figure}

The FireInsight plug-in uses analysis data gathered in real-time to provide the developer with details about the web application’s UI-JavaScript mapping. Up until this point, we have not explained how our plug-in calculates the analysis data. In fact, the plug-in does not perform any such computations. The analysis data is gathered using our own JavaScript code. Since this code is separate from the JavaScript application logic, we denoted it as \textit{analysis code}. The analysis code is executed alongside the webapp’s JavaScript code. To make this work, we must attach our analysis code to the JavaScript source files as they are delivered to the client browser. We accomplish this with our HTTP proxy. Figure \ref{fig:Proxy-3} shows the architecture for our HTTP proxy.

Our HTTP proxy acts as a gateway connecting the client browser to the outside network. We accomplish this by configuring the browser to access the Internet through the proxy. All requests for data originating from the client browser are filtered through the HTTP proxy. Our proxy is implemented in Java, using the Java 5 SDK \cite{java5sdk}. We leverage an existing HTTP communication framework created by the Apache Software Foundation, called HTTP Core \cite{httpcore}, to handle all HTTP request and response processing. We used HTTP Core 4.0 for our project.

As of HTTP 1.1 most web browsers send multiple concurrent requests to handle situations when a page contains additional files, being the majority of the cases. Examples of additional files are JavaScript source files, CSS files, image files, and other media. Our HTTP proxy leverages the Java 5 Executor framework for multi-threading \cite{javaconcurrent}.

Figure \ref{fig:Proxy-3} also illustrates the sequence of events which occur when the browser requests a JavaScript source file. For each HTTP request made by the client browser, the proxy creates and sends a new request to the destination server for the same resource. Once the server processes the request, an HTTP response is sent back to the proxy. The proxy extracts the contents of the response and determines whether the data is a JavaScript source file.

Determining whether the content is JavaScript involves more than examining the file extension type of the requested resource. Typically, JavaScript source files will have the \*.js extension, but this is not guaranteed. Since JavaScript can be embedded within an HTML page, there is the danger for some of the JavaScript source code will remain within HTML pages; depending on the server technology, the source files could have any number of file extensions other than \*.html, such as \*.jsp for JavaServer Pages or \*.asp for Microsoft .Net. In addition, Ajax requests will contain resources without file extensions even though the content returned will be JavaScript.

Fortunately, two factors help our situation. First, HTTP headers require that a Content-Type field specify the type of payload that is expected. For most well-behaved web servers, this requirement is respected and JavaScript content will have the Content-Type field set to \verb!“javascript”!. Second, best practices state that JavaScript source code should be stored in their own separate files, with \*.js extensions. As explained in Section \ref{sec:backgroundFrontEnd}, this decouples the HTML from the JavaScript. Thus, the proxy examines a combination of the Content-Type field in the HTTP header and the file extension to determine whether or not the content is JavaScript.

Any content determined as non-JavaScript is allowed to pass through the HTTP proxy unaltered. The proxy will simply copy the content data from the server’s response and sends it to the client browser. On the other hand, for content deemed to be JavaScript, the proxy proceeds to instrument the source file.

Code instrumentation involves two conceptual steps. First the JavaScript source is parsed. Parsing involves transforming the code from a stream of characters into an abstract syntax tree (AST) data structure. The AST allows us to manipulate the source code and ensure the result is valid JavaScript. Second, additional JavaScript code, which we denote as analysis code, is strategically injected into the source. We define strategic as analysis code inserted at specific locations in the source file.

To parse JavaScript code we utilize an open-source JavaScript engine written in Java, called Rhino \cite{mozilla:rhino}. We augmented the Rhino parser to inject our own analysis code into the application source during the parsing process. As illustrated in Firgure \ref{fig:Proxy-3}, Rhino is embedded within the HTTP proxy. For each JavaScript code file, the Rhino parser converts the source into an AST and allows us to determine exactly where DOM mutators occur in the code. The DOM mutators are either assignment statements, such as \verb!node.attribute=value!, or calls to the DOM API functions \verb!createElement! and \verb!appendChild!.

Once the original JavaScript source has been instrumented, it is converted back to text form and written to an HTTP response. The response is then delivered from the HTTP proxy to the client browser. From there the instrumented code is executed by the browser, which entails running both the application code and our analysis code.

Our analysis code performs a number of tasks. First, it records the JavaScript execution context in real-time. We achieved this by using a global stack to keep track of the execution call-stack. Each entry in the call stack contains the source file and exact line number of a JavaScript statement. As we parse each JavaScript file using the Rhino parser, we examine each syntax token. For any JavaScript token from the original source code that involves an assignment operation, declaration of a variable, or declaration of a function, we inject our analysis code around it to record the execution context. We attained this by adding a preceding statement to push the context information into the global stack and a proceeding statement to pop the context information. We replace the DOM API functions \verb!createElement! with our own wrapper function. We also inject a call to our own function wherever an assignment statement occurs in the original source. Thus, whenever a DOM mutation occurs in real-time, our analysis code will record the execution context.

It is important to note that we attach the execution context information to the corresponding DOM element that was just mutated. In this way, the FireInsight plug-in can access this analysis data using the Firebug HTML inspection functionality.

The second task our analysis code performs is to compile a history of all DOM mutations that have occurred so far on the current page. This is possible using the exact same technique. The only difference is we use a different global stack object. This time we do not attach the execution context information to specific DOM elements. We simply append each execution context to the global history stack. This creates a large timeline of all DOM mutators that have been executed thus far on the page.

However, the timeline as-is contains too much data and does not provide enough semantic information for the developer. What we want is to group the call-stacks based on the event handler that initiated the given execution context. To do this we need to determine the event handlers name in real-time. Fortunately, the parsing process allows us to identify when JavaScript functions are declared in the code. Thus, we can use a similar technique to record the stack of function names in real-time just as we did for recording the current call-stack for DOM mutators. We use a third global stack to record the names of all functions currently on the execution call-stack. Therefore, during any call to mutate a DOM element we also record the event handler name.

Additionally, we turn our global history stack into a stack array, where each event handler has its own history stack, which is indexed by the event handler name. This allows us to display a listing of all the event handlers that affect a selected DOM element. And, for a selected event handler we can now retrieve the history of DOM mutations and pass this to our MxGraph code to display the DMG.

**\section{Known Limitations}**

**\label{sec:implementationFeatures}**

We encountered a number of obstacles while implementing our programming tool. We review them here because they are separate from any issues discovered during the evaluation of our tool in Chapter \ref{chap:results}. Namely, the issues we list here are problems that occurred during implementation and are known limitations to our programming tool.

The first limitation is FireInsight permanently alters the JavaScript source code executed in the web browser as a result of the instrumentation procedure. In Section \ref{sec:implementationAssumptions}, we explained FireInsight’s dependence on correctly formatted and commented JavaScript source code. In order words, the code delivered to the browser should not be compressed or obfuscated.

Unfortunately, the instrumented JavaScript code from our HTTP proxy is significantly different in formatting from the original source code by the time it reaches the client browser. Most importantly, the instrumentation changes the line numbers. If left unresolved, this issue would cause FireInsight to display and highlight the incorrect line number when the developer chooses to view the source code for a selected DOM mutator.

Thankfully, our tool circumvents this formatting issue by grabbing non-instrumented versions of the source code when the developer chooses to view the source code. FireInsight does this by making Ajax requests to the proxy to grab clean copies of the original JavaScript source code. A flag is set in the HTTP header so the HTTP proxy recognizes that the requested JavaScript file should not be instrumented. The clean copies of JavaScript source are only used for display purposes and do not execute in the browser. It is the instrumented JavaScript code that continues to run in the browser.

Although we resolved the issue for FireInsight, this limitation also affects Firebug’s JavaScript features because Firebug will only have access to the instrumented code. Recall from Section \ref{subsec:backgroundVisualTools}, Firebug allows the developer to view all JavaScript source files that are loaded for the current page and provides a debugger. A developer that has both Firebug and FireInsight installed will have a very difficult time using Firebug’s JavaScript panel correctly, since the source code will be incorrectly formatted.

The second limitation is FireInsight only works within the Mozilla Firefox web browser. We outlined a number of reasons for integrating specifically with Firefox. The main rational was to improve interoperability and leverage features from existing software, particularly Firebug. Unfortunately, due to the implementation differences of various web browsers, such as Internet Explorer, Safari, Opera and Chrome, it is not possible to extend the FireInsight codebase run within other browsers.

The third limitation of FireInsight is its ability to correctly identify event handlers. This has a significant effect on the accuracy of our DMGs, since the ordered sequence of DOM mutator nodes in every DMG depends on our analysis code correctly grouping DOM mutators together based on event handler. Because event handlers are bottom level functions within the call-stack they represent the division in execution between when the browser is running and when application specific JavaScript is running. In other words, the first function to be called once execution is passed from the browser to the application UI code is the event handler.

It would seem to be straightforward to always treat the bottom level function as the event handler in terms of recording execution context information. However, this is not the case. Certain JavaScript frameworks provide a system for connecting and initializing application specific code. Thus, we can use the framework to register application event handlers for user events instead of doing it ourselves using HTML tag attributes. The framework provides a layer of abstraction in terms of event handling that separates our application code from the HTML. This is great for software maintenance. However it is bad for FireInsight, because this means the bottom level function within a call-stack is no longer an application event handler. Instead it is a framework level function.

In the case of JPS2.0, Dojo is the JavaScript framework that is used to initialize and register application event handlers. If FireInsight were to group DOM mutators based on the bottom level Dojo function, the resulting DMG would be meaningless. The DMG would no longer be application specific as it could potentially include DOM mutators from multiple application event handlers. We bypassed this issue for JPS2.0 by intentionally ignore the Dojo source file (dojo.js) during JavaScript instrumentation within the HTTP proxy. This makes Dojo code invisible to our Firebug plug-in and therefore the bottom level functions within our call-stacks are application event handlers specific to JPS2.0. The solution is not robust as it requires configuring our HTTP proxy to ignore explicit JavaScript source files, for each JavaScript framework that is being used by the application. However, it is currently our best solution.