**\chapter{Methodology}**

**\label{chap:methodology}**

JavaScript programming is challenging and an important aspect of front-end development. Some development obstacles are a result of inconsistent implementations of the ECMAScript and DOM specifications in web browsers. Other difficulties stem from fundamentally bad features in the language specification itself. However, the biggest challenge for front-end development is program understanding.

Program understanding centers on discovering the UI-JavaScript mapping within the user interface of a web application. The mapping process is an abstract representation of the cause and effect relationship between the JavaScript source code and the UI behavior.

In this chapter, we break our research problem down into its fundamental pieces and examine each in depth. We frame the UI-JavaScript mapping issue as a problem of connecting page elements to the corresponding JavaScript statements responsible for mutating them. We provide a concrete example of our research problem by presenting a case study involving software maintenance for our reference web application (Java Pet Store 2.0). We conclude by presenting our methodology for identifying the UI-JavaScript mapping and explain how it solves our research problem.

**\section{Understanding the UI-JavaScript Mapping Problem}**

**\label{sec:methodologyProblem}**

The main challenge for a developer ramping up on a new project or assigned to maintain software written by someone else, is gaining a competent understanding of how the code works. Often the language and the problem domain make code comprehension difficult to solve.

As discussed in Section \ref{sec:backgroundFrontEnd}, program understanding entails creating an abstract model to map the application logic to the application behavior. In front-end development, the application logic is JavaScript code and the application behavior is contained in the user interface. When the UI is running in the web browser it is called the browser view of the web application. Conversely, when the UI is represented by low-level source code, it is referred to as the code view of the application. Therefore, the abstract model must accurately match the behavior observed in the browser view with the corresponding source in code view.

Section \ref{subsec:backgroundJSDOM} explained that the DOM stores a tree representation of the entire web page within the browser’s memory. Hence, we will use the terms DOM and HTML page (or web page) interchangeably. The DOM API enables direct modification of the elements on the page using JavaScript. Through the DOM API, the developer can modify the entire in-browser web page dynamically without reloading the page.

The end-user observes application behavior in browser view as a series of dynamic changes to elements on the page. The web page dynamically changes when the corresponding HTML in code view is mutated. Within the browser, HTML elements are altered through calls to the DOM API, and invocation of DOM API functions can only be done through JavaScript code. Thus, at an abstract level we can reason for each piece of application behavior, there must be behavior associated with a specific change or a series of specific changes carried out through JavaScript code statements.

\begin{figure}[tbp!]

\centering

\includegraphics[width=1\columnwidth]{Visual-Mapping-2}

\caption[Mapping Behavior 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. In this figure we focus on the behavior of the page. Thus, we only show the mapping between the page and the JavaScript code file. Note the second and third JavaScript statements are marked as mutation statements, which indicates that they mutate the web page.}

\label{fig:Visual-Mapping-2}

\end{figure}

We argue that the important JavaScript statements in the application logic are the ones responsible for mutating the HTML elements on the page. These JavaScript mutation statements ultimately produce the appearance of a dynamic web page. Other types of JavaScript statements also influence the application behavior, such as conditional statements which implement control-flow. Most importantly, the mutation statements provide the most concrete mapping back to the user interface.

Upon executing a mutation statement, there is an immediate change in the web page. No other type of JavaScript statement has this direct effect on the user interface. Therefore, an accurate UI-JavaScript mapping should be based on JavaScript mutation statements. Figure \ref{fig:Visual-Mapping-2} illustrates this mental mapping between JavaScript mutation statements and the web page executing in the browser; the mutation statements are underlined.

We define the DOM representation of an HTML element on the web page to be a \textit{DOM element} or a \textit{DOM node}. Following the terminology used by Li and Wohlstadter in \cite{lw09:insight}, we define a \textit{DOM mutation} as a dynamic change in the state of the web page as a result of JavaScript code and the DOM API. We denote a \textit{DOM mutator} to be a JavaScript statement that mutates the state of the DOM. This could be the direct assignment of a node attribute to a value, such as \verb!node.style.height = ``55’‘!. Or it could be a function call to DOM API methods as specified in the W3C DOM specification. There are two specific API methods that create new DOM elements, \verb!node.createElement()! and \verb!node.appendChild()!. Figure \ref{fig:Visual-Mapping-2} shows an example of DOM mutators, where \verb!imgElem! has its \verb!style.height! and \verb!style.width! attributes set directly using assignment statements.

Therefore, the problem of determining the UI-JavaScript mapping becomes a problem of matching the DOM mutators to the corresponding DOM elements on the page. Gaining a better understanding of user interface behavior involves modeling the mapping between DOM mutators and their corresponding DOM elements.

An event handler can potentially invoke a sequence of DOM mutations in order to fulfill a single user event. Consequently, it is semantically meaningful to group those DOM mutations together. Since interactive behavior on a web page is reactive and mutates the page over time, an important element to program understanding is seeing the temporal ordering of the DOM mutators. In other words, we want to understand how DOM mutators execute in relation to each other in real-time to affect the appearance of the page. Building a tool to automate this modeling process will help improve the developer’s program understanding. The developer will see a visual representation of the model immediately, thus saving a significant amount of effort that would otherwise have been spent to create that model manually.

Our objective is to investigate how to build a programming tool that models event handlers as collections of DOM mutators and displays them in a semantically meaningful way to the developer. The developer should be able to select a DOM element in browser view and see a list of DOM mutators that affect the chosen element. Picking a specific DOM mutator should immediately show the developer the corresponding source code containing that JavaScript statement. Additionally, the developer will be able to see all of the event handlers affecting the chosen DOM element. Each event handler should be modeled as a graph of interconnected DOM mutators. The graphical representation of an event handler should reflect the actual control flow of the code, indicating the temporal order in which the DOM mutators are executed. The visualization will be similar to a state transition diagram or control-flow graph.

**\section{Motivating Example: Java Pet Store 2.0}**

**\label{sec:methodologyJPS2.0}**

To motivate our research problem and provide a benchmark to evaluate our approach in Chapter \ref{chap:results}, we introduce a web application called Java Pet Store 2.0, henceforth denoted as JPS2.0 \cite{petstore}. Its purpose is to showcase how to develop an Ajax enabled web application using Java technologies. JPS2.0 is a reference application released as part of the Sun Microsystems’ Blueprints project \cite{blueprints}, which is a set of code examples related to web development. The Blueprints project is officially packaged with each release of Glassfish, which is Sun Microsystems’ enterprise application server.

JPS2.0 simulates an online pet store, where users can come to peruse a listing of pets for sale as well as post their own listings for pets they want to sell. Pets for sale are grouped into various categories. The primary categorization is animal type, which includes dog, cat, bird, fish, and reptile. An alternative categorization is through user defined tags, which are created and associated with pet listings when they are first entered into the JPS2.0 database. Thus any pet listings that happen to share common tag descriptors can be queried by those tags. The most commonly used tag descriptors are displayed in various sections of JPS2.0. The web application contains numerous Ajax-driven user interactions which help improve the user experience.

In this section, we use JPS2.0 to present a real-world example of software maintenance. We propose a scenario involving a front-end developer assigned to modify the JavaScript behavior for JPS2.0’s RSS news feed area. This will illustrate the challenges of front-end development as discussed in our research problem and show how our approach addresses them.

\begin{figure}[tbp!]

\centering

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

\caption[Java Pet Store 2.0 – RSS New Feed Feature]

{A screenshot of the Home page from the Java Pet Store 2.0 web application in browser view. The RSS news feed area is located at the top of the page, directly below the JPS2.0 logo (top left-hand corner) and the site navigation menu (top right-hand corner). The RSS news feed area is populated with a news headline with a URL link to the actual news article.}

\label{fig:RSS-Example}

\end{figure}

The RSS news feed is a feature appearing on all pages within the JPS2.0 application. The purpose of the RSS news feed feature is to display news headlines in the news feed area, which is located at the top of the page. Figure \ref{fig:RSS-Example} shows the news feed area with respect to the Home page. The news feed area displays a single news headline at a time. Each news headline also contains a URL that links to the corresponding news article, located externally on the Java Blueprints website. The news feed cycles through a fixed set of news headlines, refreshing the news feed area with a new headline after a predefined time interval. The behavior is JavaScript driven because only the news feed area is mutated. In other words, the rest of the page remains static and does not need to be reloaded.

A developer could be interested in a specific JavaScript behavior for a number of reasons. Perhaps the JavaScript logic is working incorrectly and needs to be fixed. Another possibility is the developer has been assigned to modify the functionality to behave differently. In other cases, the developer might be ramping up on the project and would like to learn how the news feed feature is implemented. Finally, the developer could simply want to replicate the same behavior within another project and is thus interested in replicating the correct logic.

Assume in our scenario the RSS news feed is refreshing the news headline item too quickly. The developer has been given the task of modifying the refresh timer for the RSS news feed to slow down the refresh rate. There are a number of files related to the RSS news feed behavior. First there is the JSP page which contains the HTML markup for the news feed area (located in banner.jsp). Second, there are the JavaScript source files which are used to implement the news feed behavior. In this example, we have rssbar.js, which contains application logic for the RSS news feed area, and we have dojo.js, which contains the entire Dojo JavaScript framework. Dojo is important here because the RSS news feed logic is actually invoked through Dojo. These two source files have a combined 6,659 lines of code.

If the developer is not familiar with Dojo, then it will require significant effort to learn how Dojo could affect the refresh rate of the news feed behavior. The developer could consult with a fellow programmer on the project who has more knowledge about Dojo and the code base. Or the developer could consult some documentation and related resources about Dojo. These activities are all developer intensive.

Assume that our developer is familiar with Dojo and knows that it does not influence the refresh rate directly. This significantly reduces the complexity of the task because we are only concerned with rssbar.js now, which has 246 lines of code. At this point, there are two general challenges to overcome.

\begin{figure}[tbp!]

\centering

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

\caption[RSS News Feed DOM Element]

{The above HTML fragment (lines 68-75 within banner.jsp) represents the RSS news feed area on the page. The DOM element mutated by the RSS news feed behavior is the td element with id attribute ``rss-item’‘.}

\label{fig:RSS-Example-1}

\end{figure}

First, the developer needs to determine which DOM elements and corresponding element attributes are mutated to cause the UI behavior. This can be a difficult endeavor as the DOM mutations could potentially involve multiple elements and element attributes (e.g. \verb!height!, \verb!clip!, \verb!top!, \verb!width!) spread throughout the HTML code. Assume our developer discovers the DOM element that gets mutated is the \verb!td! element with id value ``rss-item’‘, as shown in Figure \ref{fig:RSS-Example-1}. Namely, an \verb!<a>! element is created with accompanying text and appended to the ``rss-item’‘ element. If an existing \verb!<a>! is already attached to the ``rss-item’‘ element, then that is removed first.

Second, the developer needs to understand the JavaScript application logic. Knowing the exact DOM mutation element allows our developer to search the JavaScript for references to ``rss-item’‘. Fortunately, there is only a single source file to search. After some investigating it becomes clear that the \verb!generateHref()! function is responsible for creating the headline title and also the headline URL. However, \verb!generateHref()! is called in multiple places in the code. Simply examining the calls directly does not clarify which one is related to the behavior of refreshing the headline. The developer must review the code and understand the \textit{calling context} for each invocation of the function within the code. This refers to the call-stack that corresponds to each execution context. It turns out the correct calling context involves an anonymous function, declared within \verb!replaceItem()!.

\begin{figure}[tbp!]

\centering

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

\caption[RSS News Feed JavaScript Behavior]

{The above JavaScript code snippet (lines 120-143 within rssbar.js) represents a portion of the the RSS news feed behavior. The function call to generateHref(), which contains the DOM mutator of interest is labeled above. The timer value that needs to be modified in order to slow down the refresh rate is labeled above.}

\label{fig:RSS-Example-2}

\end{figure}

Figure \ref{fig:RSS-Example-2} shows the JavaScript code for \verb!replaceItem()!. We can see the call to \verb!generateHref()!, which is invoked from within an anonymous function. With some effort we can see that the timer value is the second parameter of a call to the \verb!setTimeOut()! function, which is a standard method provided by the JavaScript language. The default value was 500, or 5 seconds. The developer can proceed to increase this static value and slow down the refresh rate for the news feed.

If our developer had chosen to use Firebug, then the first obstacle of trying to locate the DOM element for the news feed area could have easily been avoided. With Firebug’s HTML inspection mode, the developer could have visually navigated the page in browser view to the news feed area and seen in Firebug’s HTML panel the corresponding HTML code. This would have shown the developer that the news feed area corresponded to the \verb!<td>! element with the id value ``rss-item’‘.

\begin{figure}[tbp!]

\centering

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

\caption[Firebug HTML Inspection Mode]

{The above screenshot shows the Firebug HTML inspection mode being applied to the RSS news feed area on the JPS2.0 Home page.}

\label{fig:RSS-Example-3}

\end{figure}

This scenario is illustrated in Figure \ref{fig:RSS-Example-3}, showing the JPS2.0 Home page loaded within Firefox and the Firebug programming tool open at the bottom. We can see the news feed area at the top of the page and browser view has been selected; the corresponding HTML code is highlighted by Firebug in code view at the bottom.

Firebug’s HTML inspection feature is extremely useful because it allows the developer to intuitively search the page in browser view and immediately see the HTML code behind the DOM element. However, Firebug does not solve the second obstacle for the developer. The developer must still manually acquire a program understanding for the JavaScript logic implementing the news feed behavior.

With our approach, the developer still begins inspecting the page within browser view. However, once the news feed area has been selected, the developer can immediately see a listing of DOM element attributes that have been modified. Each of these DOM mutators has a corresponding execution call-stack that can be displayed. This takes the developer straight to the source code where the calling context is apparent right away. In the RSS news feed example, the developer is taken to the source code for \verb!replaceItem()! and can see that an anonymous function is responsible for modifying the \verb!<td>! element. From there it is straightforward as to what statement needs alteration to slow down the news feed refresh rate.

\begin{figure}[tbp!]

\centering

\includegraphics[width=.25\columnwidth]{RSS-Example-4}

\caption[Graphical Model of Execution Context]

{The above represents a conceptual model of the execution flow for the news feed behavior. More complex graphs can be found in Chapter \ref{chap:results}.}

\label{fig:RSS-Example-4}

\end{figure}

Our approach also lets the developer see a graphical model of the news feed execution. Figure \ref{fig:RSS-Example-4} shows a conceptual view of the graphical model that would be displayed for the RSS news feed behavior. Each node in the graph represents a DOM mutator context. To be precise, each node is a unique execution context for a given DOM mutator. If a single DOM mutator were invoked by two different calling contexts with differing call-stacks, then the graph would display two separate nodes. The directed edges linking nodes within the graph represent the control flow from one JavaScript mutator statement to the next. When a directed edge points from a node \textit{A} to a node \textit{B}, it indicates that the flow of control executes the DOM mutator represented by node A first, followed the DOM mutator represented by node B. Thus, the directed edges indicate order of execution.

The graph shows all execution contexts relating to a specific event handler function. We group all mutators under an event handler X if those mutators have execution contexts that were initiated by X. If a developer needs to fix or modify a specific user interaction, it will involve editing the appropriate event handler responsible for driving that interaction. Thus, grouping all DOM mutators by event handler will help the developer quickly understand the sequence of DOM changes involved for a given behavior. The developer can then make the suitable code modifications.

For the RSS news feed example, single event handler gets called repeatedly in order to refresh the news headline. Hence, the graph in Figure \ref{fig:RSS-Example-4} only has a single node. The edge linking the node to itself represents the repeated calls to refresh the news headline. The other edge, which has a single connected side, represents the beginning of the control flow. This single sided edge will always point to the initial node in the execution flow. We will see more complex models in Chapter \ref{chap:results}.

**\section{Extending Script Insight to Improve Program Understanding}**

**\label{sec:methodologySolution}**

Having defined our research problem and outlined exactly what our approach will be, we now describe the methodology for achieving our objective. In Section \ref{subsec:backgroundScriptInsight}, Li and Wohlstadter had two key insights as for how to represent the UI-JavaScript mapping. Our methodology involves incorporating these ideas into our programming tool.

\begin{figure}[tbp!]

\centering

\includegraphics[width=1\columnwidth]{Stack-Trace}

\caption[Execution Stack]

{An execution context for a DOM mutator, as represented by a call-stack. The top of the call-stack is a JavaScript statement at line 565 for the source file scroller.js. This corresponds to the current execution, which in this example is an assignment statement. The bottom of the call-stack, at line 19 of source file catalog.js, corresponds to the initial function call that started this execution context.}

\label{fig:Stack-Trace}

\end{figure}

The first idea is to capture the complete execution context when recording a DOM mutator. Tracking the exact JavaScript statement responsible for changing a DOM element is not enough, because some DOM mutators are used within multiple execution contexts. Specifically, a single DOM mutation might be called by different event handlers. Therefore, the entire call-stack needs to be recorded for each execution context. Figure \ref{fig:Stack-Trace} shows an example of an execution context. A call-stack is a succinct representation of the execution context. We can see that the call-stack is four levels deep. The top of the stack is a DOM mutation statement at line 565 of the JavaScript source file called \verb!scroller.js!. This corresponds to the current execution. The call-stack helps uniquely identify one execution context from another instance when they happen to cause the invocation of the exact same DOM mutator.

In addition, the developer is often interested in examining the source for one of the functions that precedes the DOM mutator in the call-stack. For example, in Figure \ref{fig:Stack-Trace} the developer might be more interested in the function at line 521 of source file \verb!scroller.js! than the DOM mutators. Or perhaps the developer wants to change the function call at line 521 instead of editing the DOM mutator itself. Thus, the complete execution stack contains valuable information that should be recorded. We define a \textit{mutator context} as the execution context for a DOM mutator.

The second idea is that UI behaviors often involve multiple DOM mutators and thus need to be grouped in some semantically meaningful way. Many UI behaviors involve animations on the page. Common examples are navigation elements that pop out when selected and then retract when deselected, or an image gallery showing animated changeovers while simultaneously browsing through images. These scenarios involve more than one DOM mutation per user event. A sequence of DOM mutations correspond to transitions in the state of the page. For example, an image is animated to move across a panel. Each shift of the image corresponds to a DOM mutation as well as a transition in state. These transitions become difficult to trace in real-time as the number of DOM mutations increases. Therefore in order to assist the developer, a history of mutator contexts need to be captured in real-time as the behavior is executing in the browser. The developer can then review the history to determine how the transitions occurred for specific animations.

Yet a challenge arises when searching through the history of mutator contexts. When a high frequency of DOM mutators are executed, it becomes tedious for a developer to manually review them. So, when an animation invokes a large number of DOM mutators, showing all of the mappings between mutators and mutated elements will overload the developer with too much information.

One way to combat this information overload is to organize DOM mutators in a logical manner. Li and Wohlstadter argue that DOM mutators should be grouped based on the event handlers that invoke them. This is because event handlers correspond to meaningful units of UI behavior. The overall behavior of the user interface is composed of individual user interactions on the page. Each user interaction begins with a user event. The event then triggers an event handler, which executes application logic. During execution, the event handler will invoke one or more DOM mutators to communicate a change in the state of the application to the end-user.

Organizing DOM mutators based on the event handlers that invoke them provides deeper semantic information to the developer. Once mutator contexts are organized by event handler the issue becomes how to display them. The most straightforward method is to display the contexts in an unordered list. But that does not account for the order in which they occurred. An alternative is then to display the context in an ordered list, sorted by order of execution. In spite of this, there are instances when the number of mutators invoked by a single event handler may still result in information overload. Another visualization technique is needed.

Li and Wohlstadter propose that visualizing the contexts as a control-flow graph will help resolve information overload. This is because, an event handler calls DOM mutators in a specific sequence based on control-flow logic. In the graph, each node represents a unique mutator context. Directed edges between nodes represent the order in which the contexts are executed. In many cases, the high frequency of DOM mutators is due to the same mutators being invoked repeatedly. The call-graph eliminates this overload of context information by representing the repeated execution contexts as a single node. In other words, duplicate call-stacks are collapsed into a single node. This reduces the extra noise caused by redundant execution contexts, and provides a clearer picture of the event handler control-flow. As mentioned in Section \ref{subsec:backgroundScriptInsight}, these graphs are called DOM mutation graphs (DMGs).

\begin{figure}[tbp!]

\centering

\includegraphics[width=0.6\columnwidth]{Proxy-1}

\caption[JavaScript Instrumentation through Proxy]

{The standard JavaScript instrumentation technique involves injecting additional code into existing source files, by using a proxy program. The proxy sits between the client and server and modifies all JavaScript files sent from the web server to the client browser.}

\label{fig:Proxy-1}

\end{figure}

The methodology for our programming tool involves creating analysis code to record the execution of context information while the application is running. To capture the context information, our analysis code will need to run on the client-side, since JavaScript code runs within the client’s web browser. To deliver the analysis code to the client-side, we will use the JavaScript instrumentation technique, which was explained in Section \ref{subsec:backgroundInstrumentation}. As JavaScript code is delivered to the client browser we will intercept the source files and inject our analysis code. This technique is desirable because it is non-intrusive. This means the original JavaScript source located on the server-side remains unchanged. Figure \ref{fig:Proxy-1} illustrates the standard JavaScript instrumentation setup. Injection is done by placing a proxy program conceptually between the client and the web server. The proxy intercepts all client-server communication. For server responses containing JavaScript, the proxy instruments the code file and sends the modified version to the client browser.

The analysis code will need to record the execution contexts in real-time in order to build a history of all DOM mutations that have occurred. We will rely on the flexible nature of JavaScript to replace some default language functions with our own versions. Specifically we will need to alter the \verb!node.createElement()! and \verb!node.appendChild()! functions, as mentioned in Section \ref{sec:methodologyProblem}. This will allow us to run our own code each time a new DOM element is created. Likewise, we will need to find a way to alter the JavaScript assignment operator so that we can run our own code each time a DOM element is modified. In both cases, our code will capture the DOM mutations as they happen. The history will then be an array of execution contexts stored in chronological order.

To visualize the recorded data for the developer, we will create a plug-in for the web browser. This enhances interoperability since we can leverage the functionality that is already available with the browser to render the page and execute JavaScript behavior. The plug-in must allow the developer to interactively explore the web page. This means when a developer uses our plug-in to select an element on the page, the tool must display a listing of all mutator contexts that modify that particular element. To emphasize this activity, we will refer to the interactive exploration of a page as the \textit{inspection} of that page. Selecting one of the mutator contexts should display the corresponding call-stack. Selecting one of the entries in the call-stack should open a view of the JavaScript source code, indicating the exact line for that entry. This sequence of interactions will allow the developer to easily comprehend the mapping between the mutated DOM elements on the page and the DOM mutation statements in the JavaScript source code.

Our page inspection functionality is inspired by Firebug’s HTML inspection feature. The inspection mechanism provides the developer an intuitive and visual way to locate the behavior of interest. Since Firebug already has an excellent inspection mechanism, an obvious choice is to integrate our programming tool into Firebug and leverage its HTML inspect feature.

Finally, the plug-in application will also read the array of chronological execution context data and process it to create DMGs. When the developer inspects the page and selects a DOM element that appears to get mutated as part of a user interaction, our plug-in tool will display a listing of event handlers affecting the selected element. The DMGs will be displayed within the plug-in as interactive control-flow graphs. In particular, the developer will be able to click on individual nodes within a DMG and see the call-stack. From here the developer can again click on an entry in the stack and be taken directly to the JavaScript source code. The interactive DMGs will allow the developer to interact with a piece of behavior on the page and then quickly see the control-flow for the event handler. At this point the developer can jump into the code. This sequence of activities provides the developer with an immediate insight into how the page behavior maps back to the implementation.