FireInsight: Understanding JavaScript Behaviors in Web Pages by Visually Exploring the Browser

by

Steven Xinyue Gao

B.Sc., Dalhousie University, 2004

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE STUDIES

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

December 2009

© Steven Xinyue Gao, 2009

Abstract

JavaScript is one of the most important and prevalent programming languages today. It is the language of the web browser and it has seen a surge in active development since the rise of the Ajax approach for developing highly interactive web applications. Often misunderstood as a simple scripting language, JavaScript is in fact powerful and expressive. As web applications have grown in sophistication, so have their user interfaces, which are predominately implemented as client-side JavaScript. This growth in complexity has resulted in a vast array of JavaScript frameworks, best practices and design patterns. Since JavaScript applications involve creating user interfaces, adopting a visual development approach would seem a natural choice to help manage complexity and improve program understanding. One possible approach is to allow the developer to visually relate semantically meaningful elements on a web page to the actual JavaScript source code that implements its behavior. This can be accomplished by leveraging context information during JavaScript execution. Additionally, control-flow graphs are dynamically generated to help the developer understand how a series of function calls might be related based on events triggered in the user interface. Inspired by work from Li and Wohlstadter, we implement the above approach by creating a program called FireInsight. It integrates with the Mozilla Firefox web browser and the Firebug development tool. We use an HTTP proxy to instrument JavaScript source code with our own analysis code. We analyze this approach and evaluate its effectiveness by applying FireInsight to an open-source web application called Java Pet Store 2.0.

Contents

Abstract . . . . . **ii**

Contents . . . . . **iii**

List of Figures . . . . . **v**

Acknowledgments . . . . . **vii**

Chapter 1 Introduction **1**

1.1 Problem and Motivation 2

1.2 Contributions 5

1.3 Thesis Outline 6

Chapter 2 Background **7**

2.1 Evolving the Web: Ajax, JavaScript and User Interaction 10

2.1.1 Iteration 1: Classic Web Application Model 10

2.1.2 Iteration 2: Ajax Web Application Model 13

2.2 The Client-Side: JavaScript, HTML, CSS and the DOM 17

2.2.1 HTML 17

2.2.2 CSS 19

2.2.3 JavaScript and the DOM 23

2.3 The Front-end Developer 29

2.4 Program Understanding and Software Maintenance 32

2.5 Related Work 34
2.5.1 JavaScript Frameworks and Programming Tools 35
2.5.2 Visual Programming Tools 38
2.5.3 JavaScript Instrumentation 45
2.5.4 Script Insight 48

Chapter 3 Methodology 51
3.1 Understanding the UI-JavaScript Mapping Problem 52
3.2 Motivating Example: Java Pet Store 2.0 55
3.3 Extending Script Insight to Improve Program Understanding 63

Chapter 4 Design and Implementation 70
4.1 FireInsight Architecture Overview 71
4.2 Implementation Assumptions 74
4.3 Firefox, Firebug and FireInsight 75
4.3.1 DOM Mutation Graph and MxGraph 78
4.4 JavaScript Instrumentation using an HTTP Proxy 79
4.5 Known Limitations 83

Chapter 5 Results and Evaluation 87
5.1 FireInsight Evaluation 88
5.1.1 Case Study: RSS News Feed 88
5.1.2 Case Study: Catalog Browser Info Pane 96
5.1.3 Case Study: Pet Info Overlay Pop-up 108
5.2 Challenges and Drawbacks 114
5.2.1 JavaScript Frameworks 114
5.2.2 JavaScript Instrumentation 117

Chapter 6 Conclusions and Future Work 119

Bibliography 121

List of Figures

2.1 Iteration 1: A Classic Web Application Architecture 11
2.2 Client-Server Communication for the Classic Web Application Model 12
2.3 Iteration 2: An Interactive Application Architecture. 15
2.4 A Simple HTML Document 18
2.5 Screenshot of the HTML Document 19
2.6 An HTML Document with CSS code 21
2.7 Screenshot of the HTML Document with CSS 23
2.8 An HTML Document with CSS code and JavaScript code 24
2.9 HTML and the DOM 25
2.10 Screenshot of the HTML Document with CSS and JavaScript 28
2.11 Mapping Behavior and Presentation of Web Page, from Browser to Implementation 30
2.12 A screenshot of Adobe Dreamweaver 39
2.13 A screenshot of Firebug 42

3.1 Mapping Behavior of Web Page, from Browser to Implementation . . 53
3.2 Java Pet Store 2.0 - RSS New Feed Feature 57
3.3 RSS News Feed DOM Element 59
3.4 RSS News Feed JavaScript Behavior 60
3.5 Firebug HTML Inspection Mode 61
3.6 Graphical Model of Execution Context 62

3.7 Execution Stack 64
3.8 JavaScript Instrumentation through Proxy 67

4.1 FireInsight Architecture 72
4.2 FireInsight Plug-in 76
4.3 FireInsight HTTP Proxy 79

5.1 FireInsight Attribute View: RSS News Feed 90
5.2 FireInsight Source Code View: RSS News Feed 91
5.3 FireInsight Event Handler View and DMG View: RSS News Feed . . 93
5.4 Browser View and FireInsight DMG View: RSS News Feed 95
5.5 FireInsight Attribute View: RSS News Feed Part 2 97
5.6 FireInsight DMG View: RSS News Feed Part 2 98
5.7 Case Study: Catalog Browser Info Pane 99
5.8 FireInsight Attribute View: Catalog Browser Info Pane 101
5.9 FireInsight Event Handler and DMG Views Part 1: Catalog Browser Info Pane 103
5.10 FireInsight Event Handler and DMG Views Part 2: Catalog Browser Info Pane 104
5.11 Case Study Final Result: Catalog Browser Info Pane 107
5.12 Case Study: Pet Info Overlay Pop-up 108
5.13 JavaScript Bug: Pet Info Overlay Pop-up 110
5.14 Inspection Mode: Pet Info Overlay Pop-up 111
5.15 Event Handlers: Pet Info Overlay Pop-up 113
5.16 DMG View: Pet Info Overlay Pop-up 115
5.17 Handling JavaScript Frameworks 116

Acknowledgments

Thanks to my supervisor Eric Wohlstadter for the thoughtful advice and diligence in guiding me towards the completion of this thesis. Thanks to Loyal Chow for the many interesting discussions regarding JavaScript and web development in general. Thank you Kris De Volder for taking the time to be on my examining committee.

Thanks to my parents for their love and support, without which I would never have reached this point in my academic career. Finally, thanks to my lovely Chenoa, for supporting me through all the ups and downs in finishing my Masters degree.

Chapter 1

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 [24], 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 [33]. Software, such as

JSLint [4] and Aptana [2], provide invaluable assistance to help developers avoid the weaknesses of the language. Other tools such as Firebug [14] and Dreamweaver [1], enable developers to program the UI using visual and interactive techniques.

1.1 Problem and Motivation

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 `class` and `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 [19], 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 [18, 17], 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 [25]. 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 [23], 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 [21], 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 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.

1.2 Contributions

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 [21] and expanding upon their work in a number of meaningful ways.

The most important contribution is our software tool itself, which we call FireInsight[1]. 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 [14], a popular web development tool built on top of the Mozilla Firefox web browser [9]. 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 [21], they did not implement it as a feature within Script Insight. In

[1]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 [13].

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 4 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 5.

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.

1.3 Thesis Outline

The remainder of this thesis is organized as follows. Chapter 2 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 3 formalizes our research problem and delves into the methodology we used to solve it. Chapter 4 describes how we implemented our methodology in the form of a programming tool, called FireInsight. Chapter 5 provides a comprehensive evaluation of FireInsight using the Java Pet Store 2.0 application and discusses the results. Finally, Chapter 6 presents our conclusions and discusses possible future work for our research.

Chapter 2

Background

In a short period of time the *World Wide Web* (or the *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[1]. 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 *web application* (or simply a *webapp*) as a software application that gets executed remotely by a client via a computer network such as the Internet [32]. This necessitates the application be divided into two parts: (1) a *client-side*, and (2) a *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. *Web application development* (or simply *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

[1]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.

possible on the Web. For example, social webapps like Facebook [6] and Twitter [30] 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 *user interface* (*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 *Hyper Text Markup Language* (*HTML*). Consequently, a webapp user interface is a series of HTML documents (or *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 [12]. 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[2]. 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.

[2]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.

2.1 Evolving the Web: Ajax, JavaScript and User Interaction

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 [20] 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 *Hyper Text Transfer Protocol* (*HTTP*) and documents are defined in HTML.

2.1.1 Iteration 1: Classic Web Application Model

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 2.1 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 2.1 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.

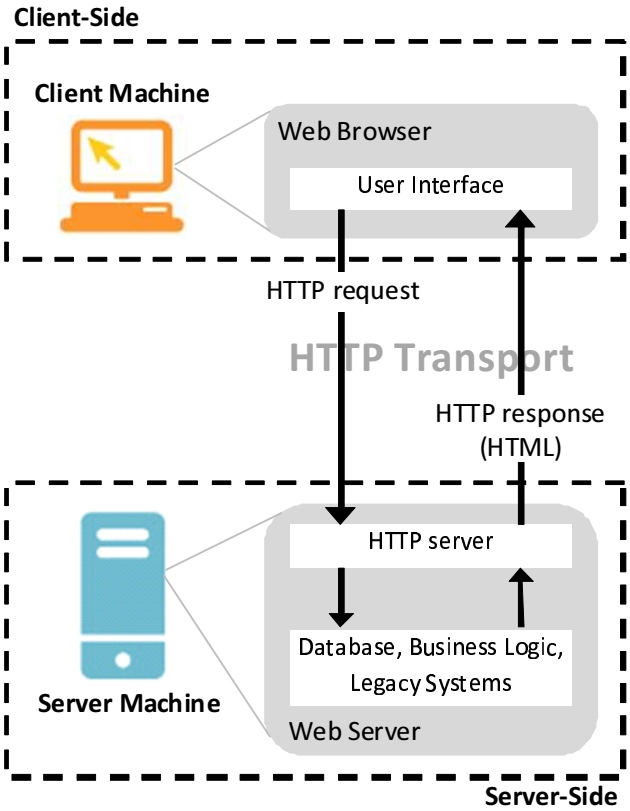


Figure 2.1: The simplest client-server architecture for a web application is a client machine (executing a web browser) and a server machine (executing the web server). The browser makes an HTTP request for content and waits for a response from the server. When the server responds it returns back HTML, 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).

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

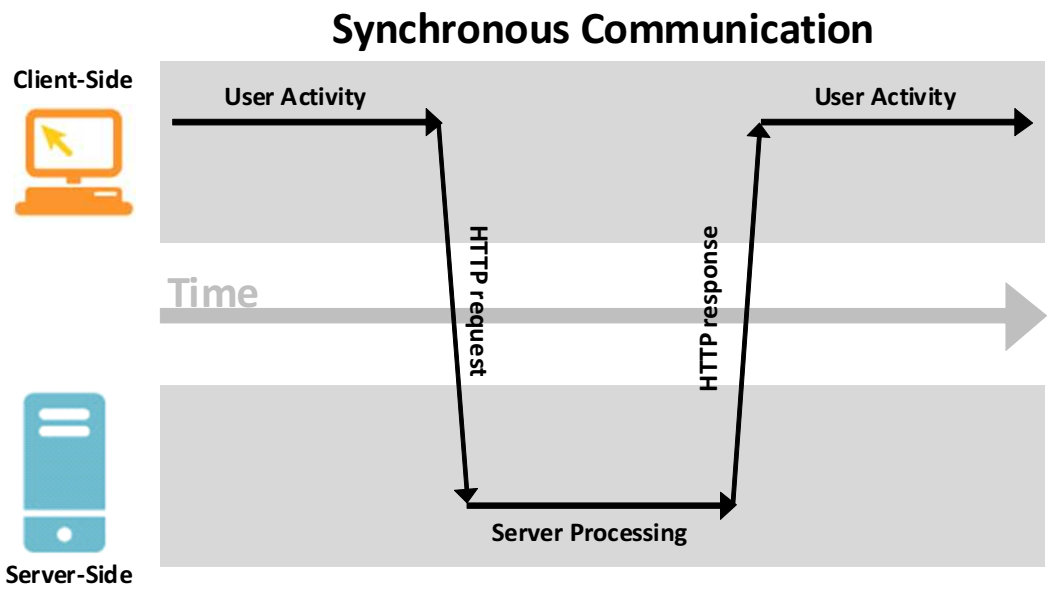


Figure 2.2: The above sequence diagram shows the client-server communication for the classic webapp model. All communication is synchronous which means that the browser (and the user) must wait while a request is being processed by the server. Once the server sends a response back to the browser, the user can resume using the application.

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 2.1).

As noted in [12], 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 2.2, 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.

2.1.2 Iteration 2: Ajax Web Application Model

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 *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 *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" [12]. 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 *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:

- HTML/XHTML for defining semantic structuring
- *Cascading Style Sheets* (*CSS*) for defining formatting and presentation
- *Document Object Model* (*DOM*) for dynamic display and interaction
- XML or *JavaScript Object Notation* (*JSON*) for data interchange
- *XMLHttpRequest* (*XHR*) for supporting asynchronous client-server communcation
- JavaScript for binding all of above components together

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.

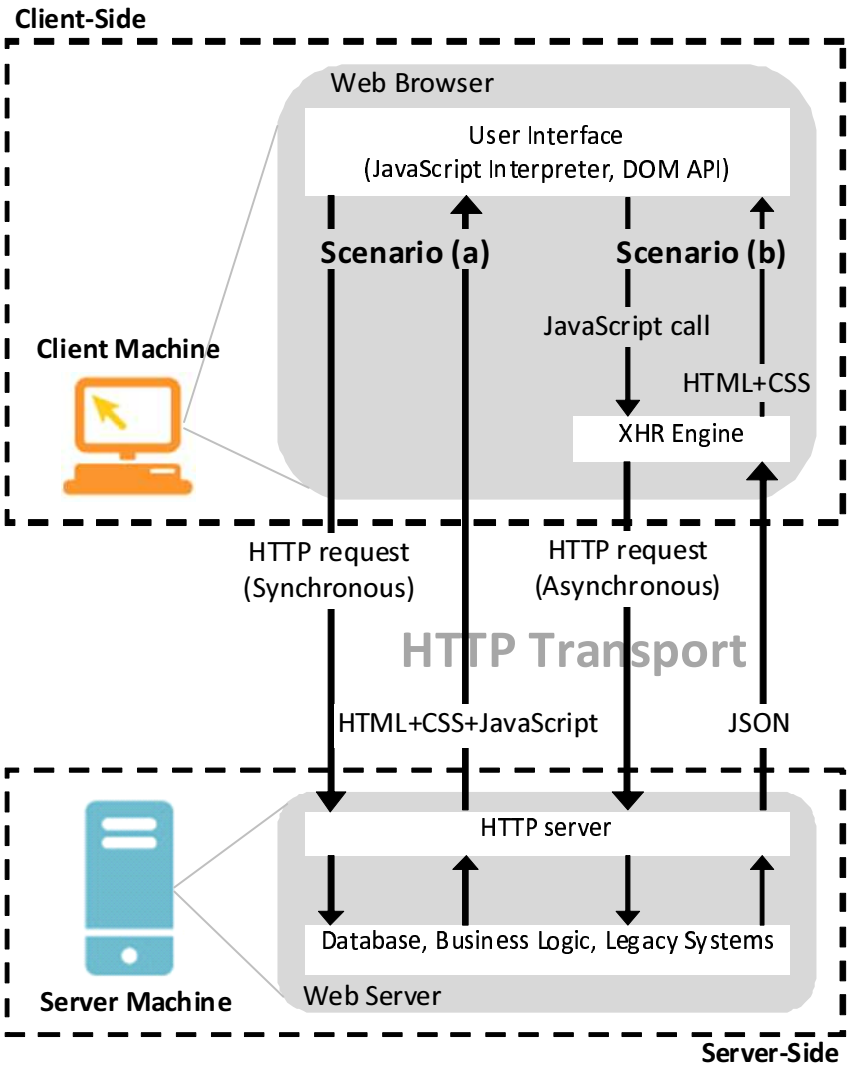


Figure 2.3: The client-side now consists of: HTML, JavaScript, CSS, the DOM and an XHR engine. Two request-response scenarios are now possible. In scenario (a), the browser makes a *synchronous* HTTP request to the server for content (classic webapp model). The browser must wait for the server's response. 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 *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.

Figure 2.3 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 2.3 provides an accurate model of the current state of web development. As we mentioned in Section 2.1.1, 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.

2.2 The Client-Side: JavaScript, HTML, CSS and the DOM

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 *Dynamic HTML* (*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.

2.2.1 HTML

We begin our example with a basic HTML document without any JavaScript code or CSS code. This document is shown in Figure 2.4 and is named `"example.html"`.

HTML provides a way to define *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 (`<div>`) tag is used to wrap arbitrary blocks of HTML and make them referable

example.html:

```
<html>
  <head>
  </head>
  <body>
     <div class="container">
        <div id="image_div">
           <img id="image" src="../image.gif"
                width="107" height="98" imgsize="small"/>
        </div>
        <div class="center_div">
           <h1>Thesis Chapter 2: JavaScript, HTML, CSS and DOM</h1>
           <p>JavaScript, HTML, CSS and the DOM together provide
              a means to create an interactive user interface.</p>
           <p>With HTML you can define the structure of the text
              in your document.</p>
           <p>With CSS you can control the text (like font,
              color, size, etc.) and the layout (like
              backgrounds, margin, padding, etc.) of a website,
              in one single file.</p>
           <p>With JavaScript you can define the behavior of
              your document and how it reacts to user inputs.</p>
           <p>With the DOM your JavaScript code can directly
              modify the structure, text, and display of your
              document.</p>
       </div>
     </div>
  </body>
</html>
```

Figure 2.4: A simple HTML document with no JavaScript code or CSS code.

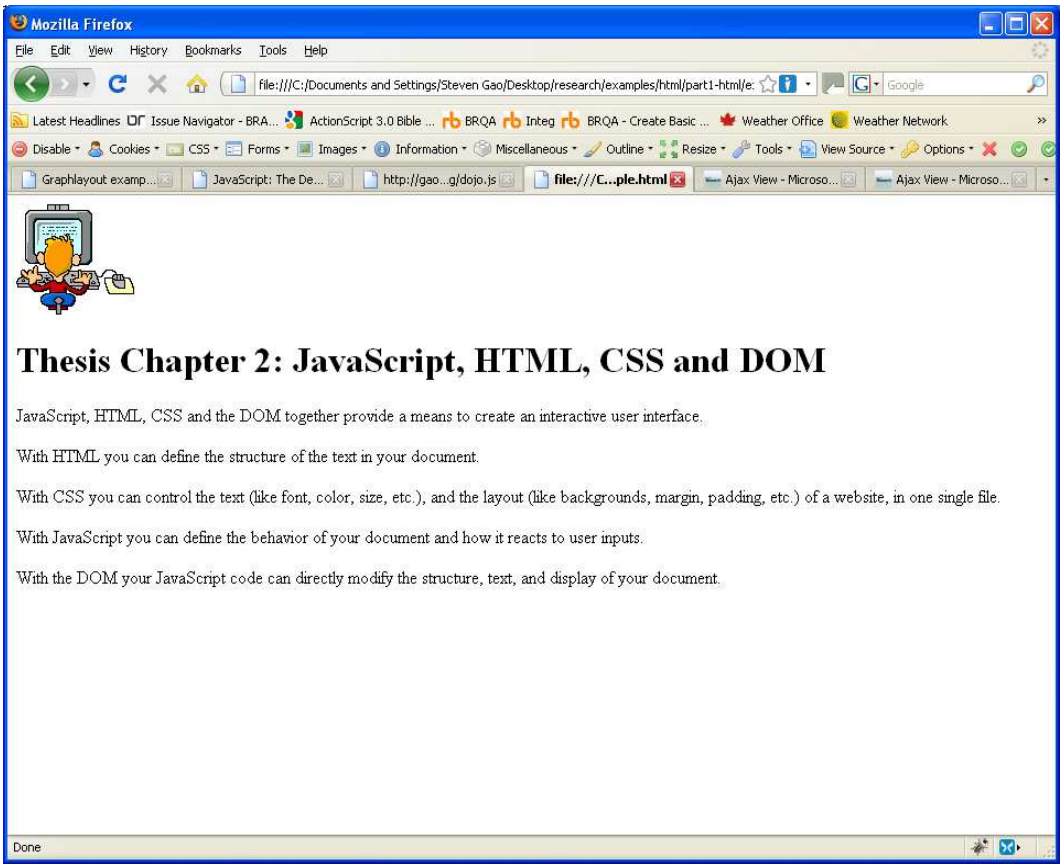


Figure 2.5: A simple HTML document with no JavaScript code or CSS code.

from JavaScript using the *class* and *id* attributes.

Finally, Figure 2.5 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.

2.2.2 CSS

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 "Verdana" instead of the default "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.

As shown in Figure 2.6, we now have two separate files, namely "example.html" and "example.css". The HTML document now contains a <link> tag within the <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 <p> tags to be "sans serif", the CSS code would be: p {font-family: "sans serif"; }. For a given selector one

(a) example.html:

```html
<html>
  <head>
     <link rel="stylesheet" type="text/css" href="example.css"/>
  </head>
  <body>
     <div class="container">
        <div id="image_div">
           <img id="image" src="../image.gif"
                width="107" height="98" imgsize="small"/>
        </div>
        ...
     </div>
  </body>
</html>
```

(b) example.css:

```css
body {
   font-family: verdana;
   margin-left: 210px;
   background: #5d9ab2;
}
.container {
   text-align: center;
}
.center_div {
   border: 1px solid gray;
   margin-left: auto;
   margin-right: auto;
   width: 90%;
   height: 340px;
   background-color: #d0f0f6;
   text-align: left;
   padding: 8px;
}
#image_div {
   position: absolute;
   left: 0px;
}
#image {
   width: 107px;
   height: 98px;
}
```

Figure 2.6: 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.

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, `<div>` tags are important for grouping HTML elements together for the purposes of defining common style or common behavior. A `<div>` tag is identified by either a class attribute or an id attribute. In CSS code, a selector for a given class name `"foo"` is defined as `".foo"`. In CSS code, a selector for a given id name `"foo"` is defined as `"#foo"`.

Figure 2.6 shows the div with class value `"center_div"` groups all the text together into a block. In the CSS file, the class `"center_div"` has the following style attributes:

```
border: 1px solid gray;
margin-left: auto;
margin-right: auto;
width: 90%;
height: 340px;
background-color: #d0f0f6;
text-align: left;
padding: 8px;
```

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

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

Figure 2.7 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 `"example.css"` file within the other documents and ensure we use the same class and id values for the `<div>` tags on those pages.

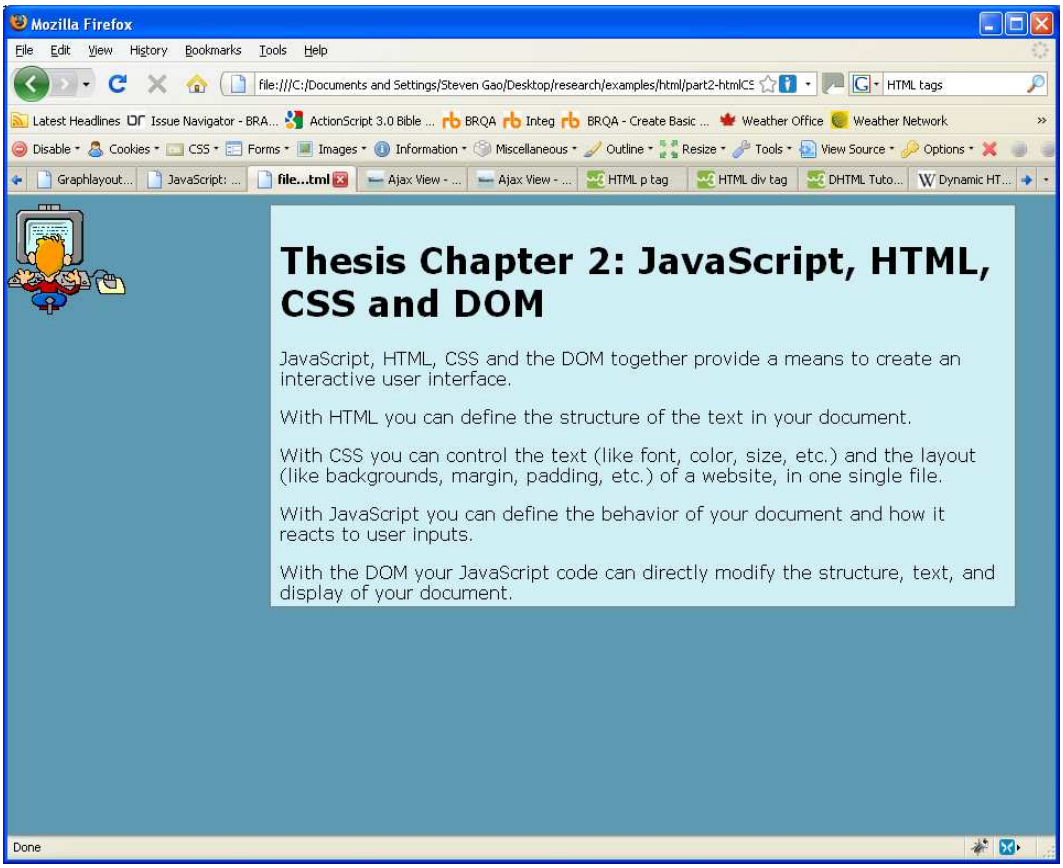


Figure 2.7: An HTML document with CSS code for style formatting.

2.2.3 JavaScript and the DOM

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 2.8 shows the complete implementation for this page. In addition to `"example.html"` and `"example.css"` we now have `"example.js,"` which contains the JavaScript code. The HTML page includes an additional HTML element within

(a) example.html:

```html
<html>
  <head>
     <link rel="stylesheet" type="text/css" href="example.css"/>
     <script type="text/javascript" src="example.js"/>
  </head>
  <body>
      <div class="container">
         <div id="image_div">
            <img id="image" src="../image.gif" imgSize="small"/>
            <input id="image_input" type="button"
                   onclick="changeSize()" value="Enlarge"/>
         </div>
      </div>
      ...
  </body>
</html>
```

(b) example.css:

```css
body {
   font-family: verdana;
   margin-left: 210px;
   background: #5d9ab2;
}
.container {
   text-align: center;
}
.center_div {
   border: 1px solid gray;
   margin-left: auto;
   margin-right: auto;
   width: 90%;
   height: 340px;
   background-color: #d0f0f6;
   text-align: left;
   padding: 8px;
}
#image_div {
   position: absolute;
   left: 0px;
}
#image_input {
   position: absolute;
   left: 0px;
   top: 200px;
}
#image {
   width: 107px;
   height: 98px;
}
```

(c) example.js:

```javascript
function changeSize() {
   var imgElem =
          document.getElementById("image");
   var buttonElem =
          document.getElementById("image_input");

   var attr = imgElem.attributes;
   for (var i=0; i<attr.length; i++) {
      if (attr[i] &&
          attr[i].nodeName === "imgsize") {
         if (attr[i].nodeValue === "small") {
            imgElem.style.height = "206";
            imgElem.style.width = "225";
            attr[i].nodeValue = "large";
            if (buttonElem) {
               buttonElem.value = "Shrink";
            }
            return;
         }
         else {
            imgElem.style.height = "98";
            imgElem.style.width = "107";
            attr[i].nodeValue = "small";
            if (buttonElem) {
               buttonElem.value = "Enlarge";
            }
            return;
         }
      }
   }
}
```

Figure 2.8: 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.

(a) HTML page

```
1  <html>
2    <head>
3      <title>Example Page</title>
4    </head>
5    <body>
6       <div>Content</div>
7    </body>
8 </html>
```

(b) DOM

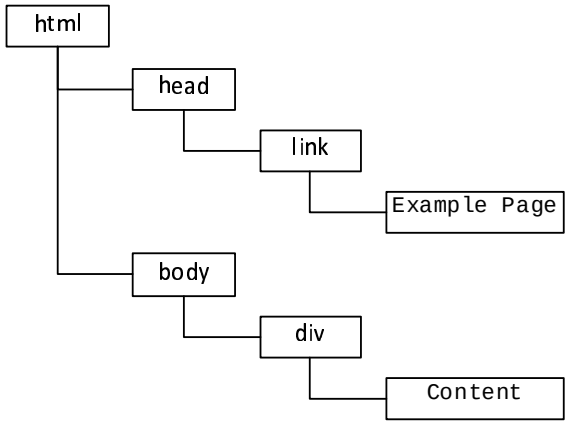


Figure 2.9: A simple web page is represented in two forms: (a) as an HTML document, and (b) as a DOM tree data structure.

the `<head>` tag. Similar to the `<link>` tag for CSS includes, the `<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 `<input>` tag has also been added to the HTML document. The type attribute for the `<input>` tag is set to `"button"`. This instructs the web browser to display the input as a button on the page. The *onclick* attribute for the `<input>` tag is an example of an event handler attribute, as discussed in Section 2.2.1. It's value `"changeSize()"` specifies the JavaScript function located within `"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.

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 2.9 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 [33].

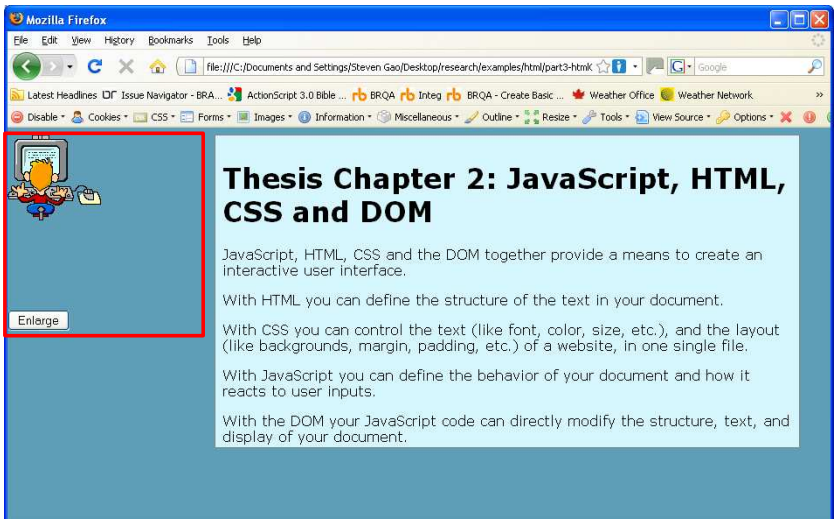
Figure 2.8(c) shows the full source code for the function `changeSize()`. JavaScript code executed within a web browser has access to a number of global objects that are made available through the DOM API. One such object is the document object. The first two statements within `changeSize()` make calls 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 `changeSize()`, we look for `"image"` which is the `<img>` element and `"image_input"` which is the `<input>` element. Notice that in the HTML, the `<img>` tag has an attribute named `"imgsize"`. This is not one of the standard attributes allowed for the `<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 `"small"`. Within `changeSize()`, we loop through the attributes for the `<img>` element until we find the `"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 `style.width` to 225 pixels and `style.height` to 206 pixels. We also change the text label of the input button to `"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 `"Enlarge"`.

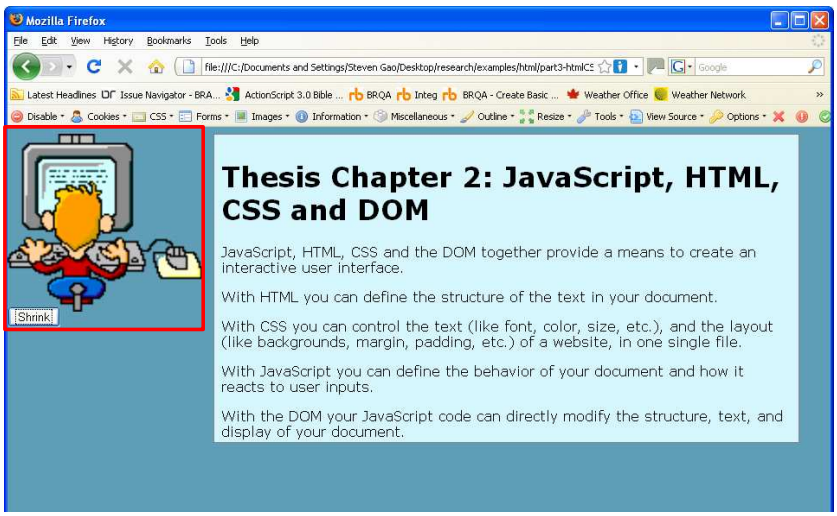
We note here that the values 107 pixels for the width and 98 pixels for the height are written once in the CSS code, to define the initial size of the image, and then again in the JavaScript code, to define the application logic to set the image size back to its initial size. This produces a tight coupling between the JavaScript and CSS because the JavaScript is partially responsible for the presentation of the page. As noted in [33], 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.

Figure 2.10 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 `"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 `"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 `<script>` tag, like this:



(a) Small GIF Image



(b) Large GIF Image

Figure 2.10: 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.

```
<script type="text/javascript">
// JavaScript goes here
</script >
```

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 `<script>` tag.

2.3 The Front-end Developer

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 *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 *front-end development*.

The front-end developer is fluent in JavaScript, HTML and CSS. The *integrated development environment* (*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[3]. 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

[3]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.

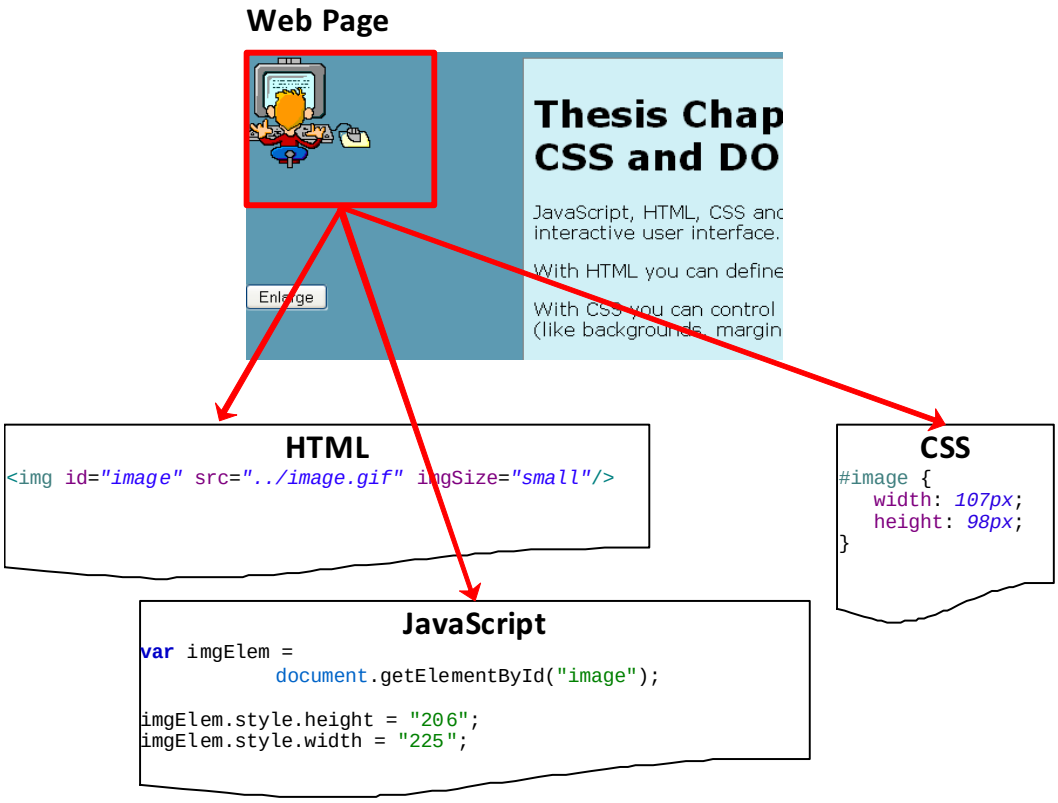


Figure 2.11: A piece of functionality, shown as the GIF image on the web page within the browser, is mapped back to the actual implementation, which includes source code from an HTML page, a CSS file, and a JavaScript file. Note that the id value "image" ties the HTML with the matching CSS and JavaScript code.

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.

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 2.11. In the example, the page element of interest is an image. It has a corresponding `<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 [23], 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 2.2.1, 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 2.2, where the id attribute `"image"` associated with an `<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 2.11 demonstrates this coupling where the id `"image"` is spread across the HTML, CSS and JavaScript code.

As described in Section 2.2.3, 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 re-

sponsible 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.

2.4 Program Understanding and Software Maintenance

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 *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 [3], 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 [3] *software maintenance*, is the most commonly used term. However, Li and Wohlstadter use term reverse engineering in [21]. 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 [3], 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 *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 *browser view* as the representation of the user interface when it is running within the web browser. Similarly, we define the *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 [3]. Or one could study the implementation by executing the source code and observing the resulting dynamic behavior [3]. 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 *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.

2.5 Related Work

As reported by [5], 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 [15], JavaScript frameworks [24], JavaScript best practices [5, 33] and general JavaScript programming [7, 33].

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.

2.5.1 JavaScript Frameworks and Programming Tools

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 [24]. 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 rou-

tinely 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 `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 [24]. 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 *JSLint*, created by Douglas Crockford [4, 5]. 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 [5].

JSLint's name is derived from a similar program, called *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 [2]. 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.

2.5.2 Visual Programming Tools

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.

An excellent example of a WYSIWYG editor for front-end development is Adobe Dreamweaver [1, 31]. 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 2.12 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

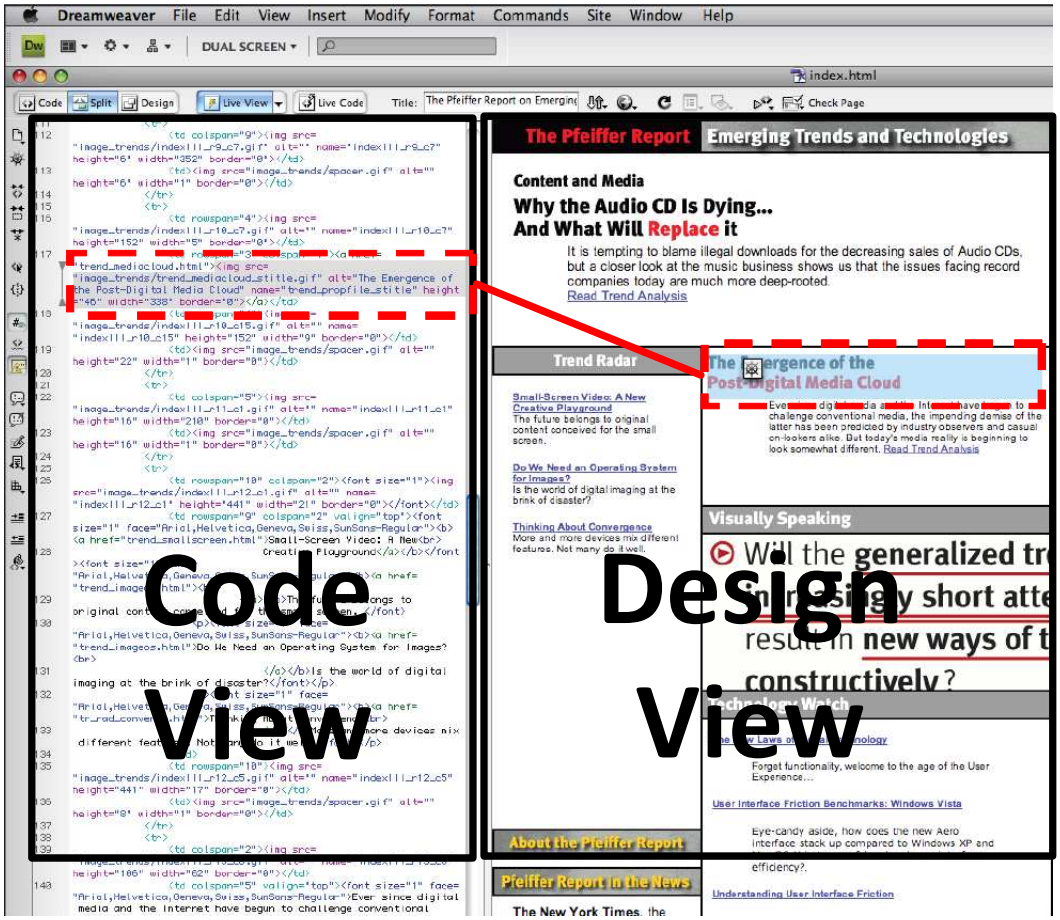


Figure 2.12: Adobe Dreamweaver is a WYSIWYG editor for front-end development. It provides a code view and a browser view of the web page. Note that the browser view is called design view in Dreamweaver. In the above screenshot a section of the page is highlighted in both the code and design view. This 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.

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 [14]. 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.

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 2.13. 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

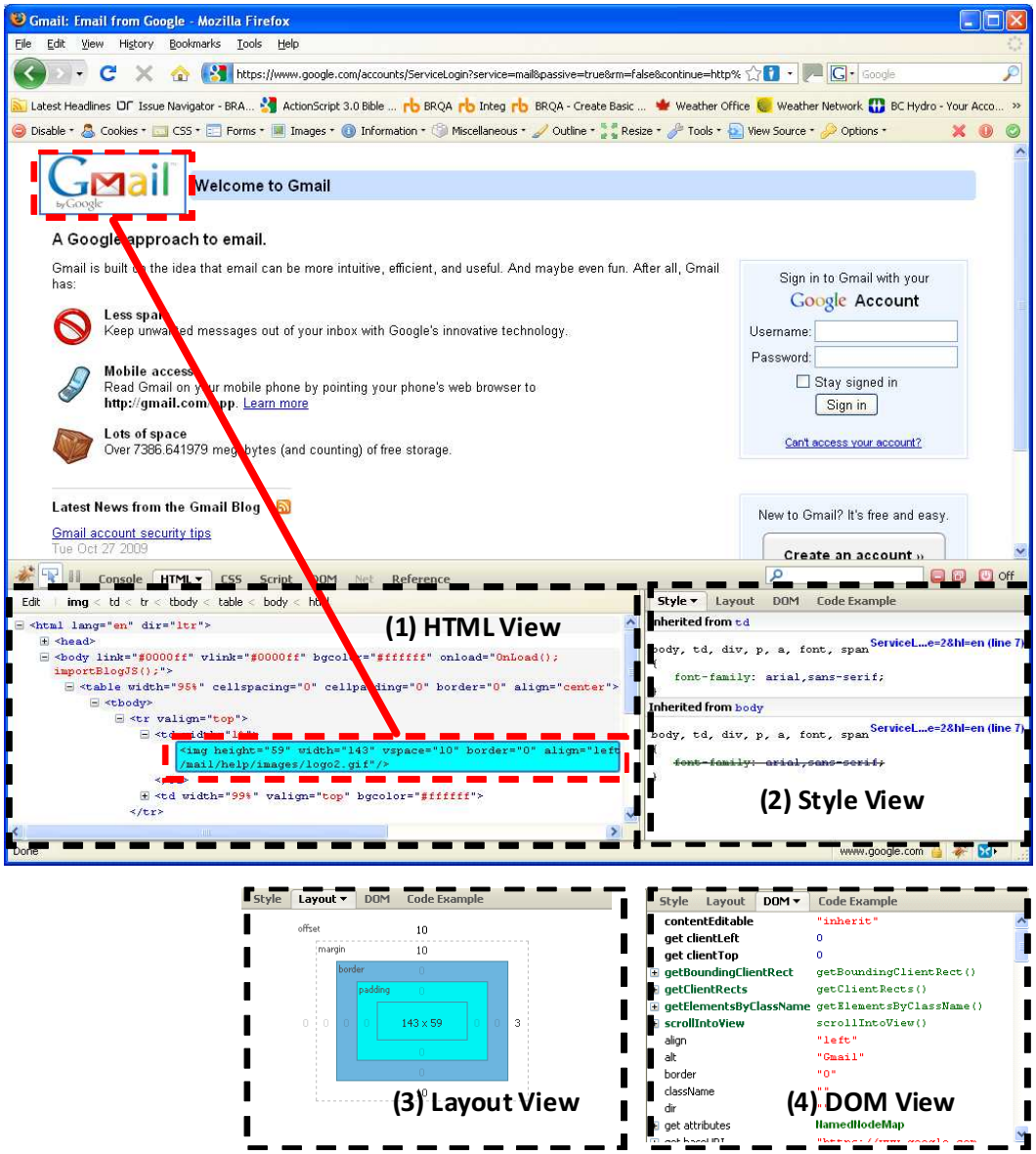


Figure 2.13: Firebug is a plug-in for the Mozilla Firefox browser. It provides a variety of useful development tools. One of the most popular is its HTML inspection feature, which allows the user to interactively select a section of the page in browser view, and then see the corresponding (1) HTML code, (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.

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 [11], 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 4 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 [23]. 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.

2.5.3 JavaScript Instrumentation

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 *JavaScript instrumentation* [19, 18, 17].

In [19], 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 [18, 17], 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 [25]. 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.

2.5.4 Script Insight

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 [21], Li and Wohlstadter introduce a novel approach for determining the UI-JavaScript mapping in a web application, and prototype implementation called *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 2.4, 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 *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 *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 *DOM mutation graph* (or *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 [21]. 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 [21] 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 be-

havior. We also present some technical challenges associated with identifying event handlers for the DMG, which were not addressed by Li and Wohlstadter.

Chapter 3

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.

3.1 Understanding the UI-JavaScript Mapping Problem

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 2.3, 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 2.2.3 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

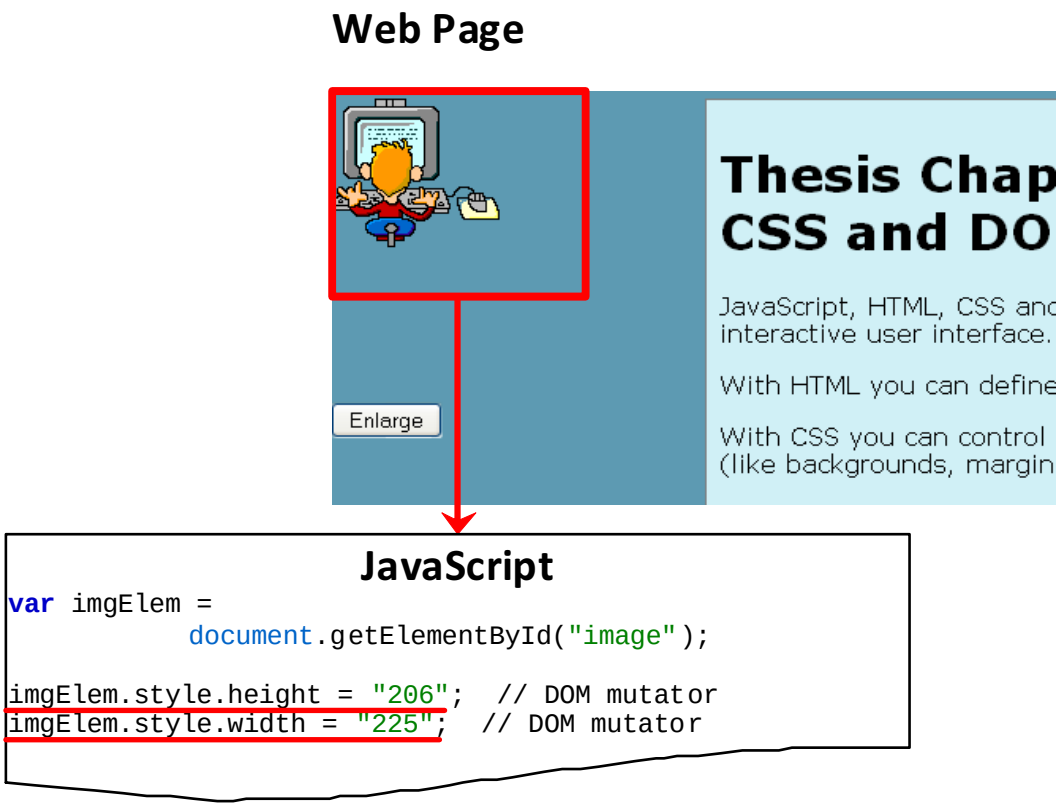


Figure 3.1: A piece of functionality, shown as the GIF image on the web page within the browser, is mapped back to the actual implementation. In this figure we focus on the behavior of the page. Thus, we only show the mapping between the page and the JavaScript code file. Note the second and third JavaScript statements are marked as mutation statements, which indicates that they mutate the web page.

code statements.

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 3.1 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 *DOM element* or a *DOM node*. Following the terminology used by Li and Wohlstadter in [21], we define a *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 *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 `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, `node.createElement()` and `node.appendChild()`. Figure 3.1 shows an example of DOM mutators, where `imgElem` has its `style.height` and `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.

3.2 Motivating Example: Java Pet Store 2.0

To motivate our research problem and provide a benchmark to evaluate our approach in Chapter 5, we introduce a web application called Java Pet Store 2.0, henceforth denoted as JPS2.0 [28]. 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 [27], 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.

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 3.2 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

Figure 3.2: 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.

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.

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. `height`, `clip`, `top`, `width`) spread throughout the HTML code. Assume our developer discovers the DOM element that gets mutated is the `td` element with id value "rss-item", as shown in Figure 3.3. Namely, an `<a>` element is created with accompanying text and appended to the "rss-item"

```
<div id="rss-bar">
  <table border="0" cellpadding="0" cellspacing="0">
      <tr>
            <td id="rss-channel" nowrap="true"></td>
            <td id="rss-item" nowrap="true"></td>
      </tr>
  </table>
</div>
```

Mutated DOM element

Figure 3.3: 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".

element. If an existing `<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 `generateHref()` function is responsible for creating the headline title and also the headline URL. However, `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 *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 `replaceItem()`.

Figure 3.4 shows the JavaScript code for `replaceItem()`. We can see the call to `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 `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.

```
function replaceItem() {
    if (itemIntervalId) {
        clearTimeout(itemIntervalId);
    }
    // fadeout the current item and pop the next one in.
    var cItem = document.getElementById("rss-item");
    dojo.lfx.html.fadeOut(cItem.getElementsByTagName('a')[0], 500);
    var waitId = setTimeout(function(waitId) {
        clearTimeout(waitId);
        //cItem.removeChild(cItem.firstChild);
        if (currentItem < (rssItemNum -1)) {
            currentItem += 1;
        } else {
            currentItem = 0;
        }
        var decTitle = decodeURL(jsonData.channel.item[currentItem].title);
        var shortTitle = cutStringatWs(decTitle, limitCharNum) + " ...";
        generateHref(shortTitle, jsonData.channel.item[currentItem].link, "rss-item");
        // attach event for onmouseover(pause) and onmouseout(resume)
        var aNodes = cItem.getElementsByTagName("a");
        dojo.event.connect(aNodes[0], "onmouseover", "pauseCycle");
        dojo.event.connect(aNodes[0], "onmouseout", "resumeCycle");
        cycleRss();}, 500);
}
```

Refresh Timer

DOM Mutator Function

Figure 3.4: 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.

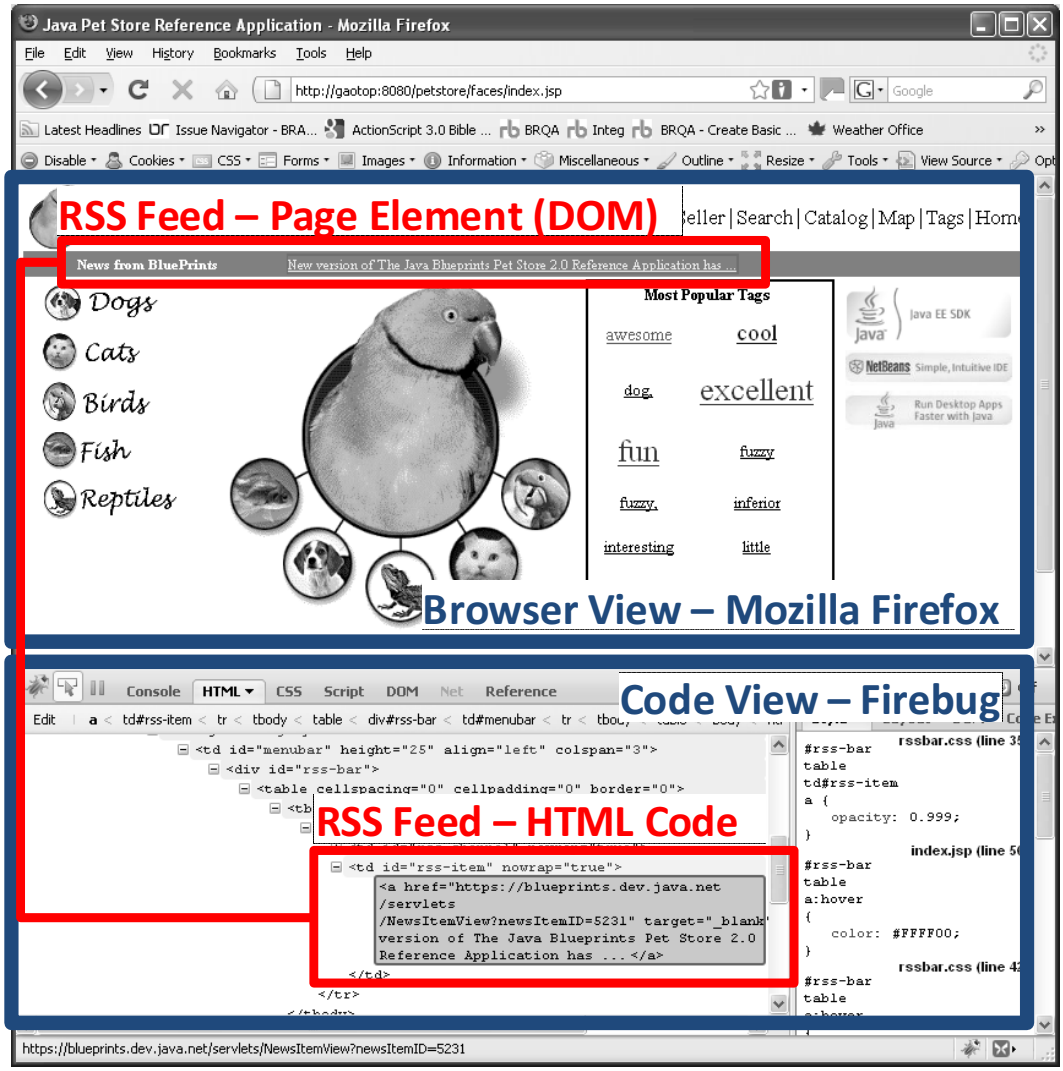


Figure 3.5: The above screenshot shows the Firebug HTML inspection mode being applied to the RSS news feed area on the JPS2.0 Home page.

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 `<td>` element with the id value "rss-item".

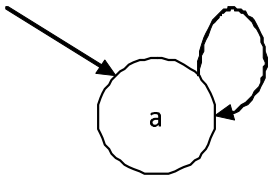


Figure 3.6: The above represents a conceptual model of the execution flow for the news feed behavior. More complex graphs can be found in Chapter 5.

This scenario is illustrated in Figure 3.5, 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 `replaceItem()` and can see that an anonymous function is responsible for modifying the `<td>` element. From there it is straightforward as to what statement needs alteration to slow down the news feed refresh rate.

Our approach also lets the developer see a graphical model of the news feed execution. Figure 3.6 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 A to a node 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 3.6 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 5.

3.3 Extending Script Insight to Improve Program Understanding

Having defined our research problem and outlined exactly what our approach will be, we now describe the methodology for achieving our objective. In Section 2.5.4, Li and Wohlstadter had two key insights as for how to represent the UI-JavaScript

Figure 3.7: 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.

mapping. Our methodology involves incorporating these ideas into our programming tool.

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 3.7 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 `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 3.7 the developer might be more interested in the function at line 521 of source file `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 *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 2.5.4, these graphs are called DOM mutation graphs (DMGs).

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,

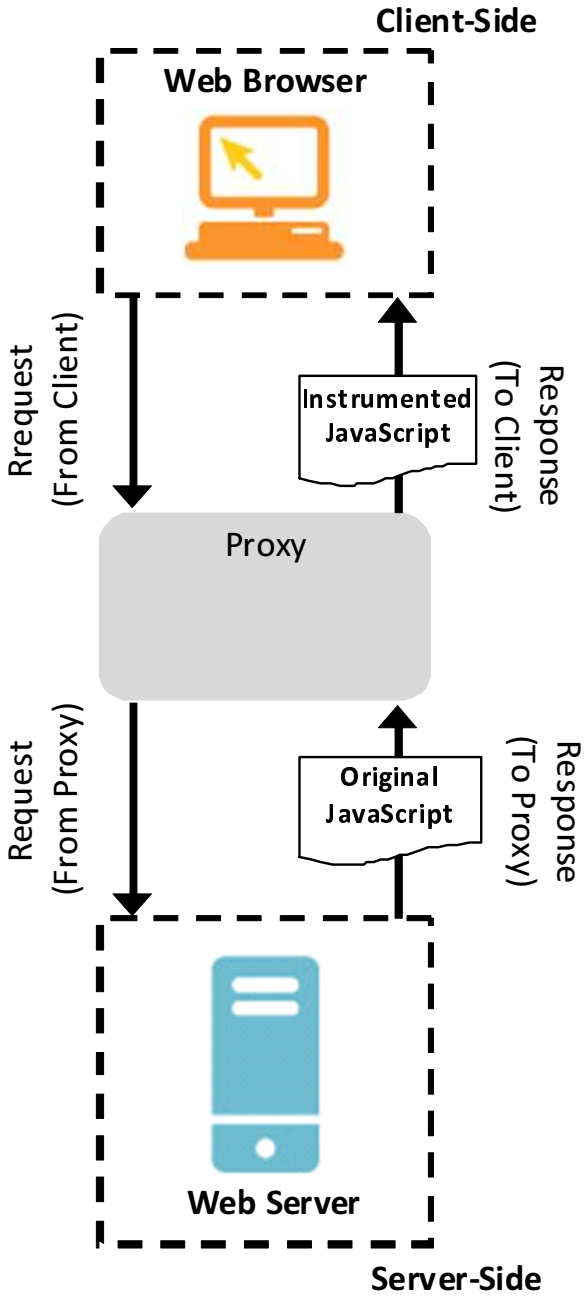


Figure 3.8: The standard JavaScript instrumentation technique involves injecting additional code into existing source files, by using a proxy program. The proxy sits between the client and server and modifies all JavaScript files sent from the web server to the client browser.

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 2.5.3. 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 3.8 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 `node.createElement()` and `node.appendChild()` functions, as mentioned in Section 3.1. 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 *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.

Chapter 4

Design and Implementation

Chapter 3 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 *FireInsight*. Chapter 3 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[1]. 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 `i` has an edge pointing to a node `j`, then DOM mutator `i` is executed first, followed by DOM mutator `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.

4.1 FireInsight Architecture Overview

FireInsight's architecture contains two software components, both of which are necessitated by the feature requirements. Figure 4.1 shows the architecture of our tool

[1]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.

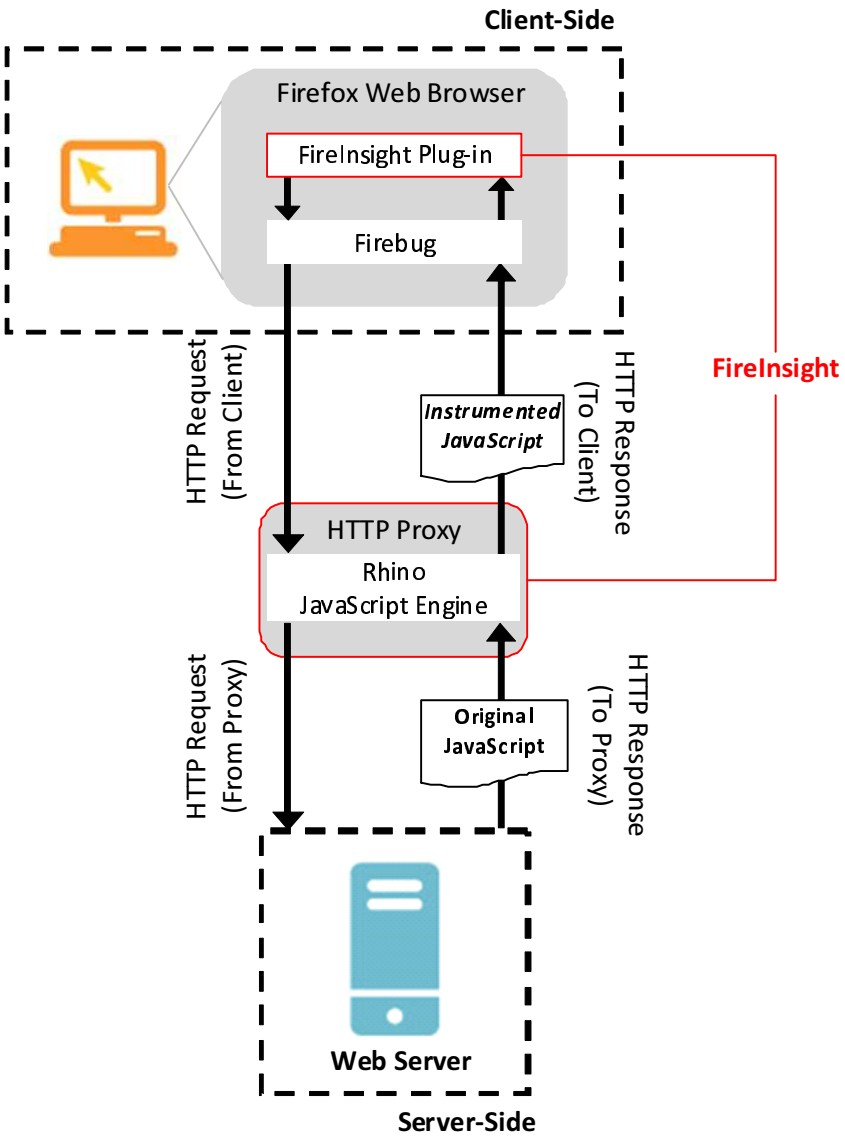


Figure 4.1: FireInsight's architecture has two software components, which are highlighted above. The first is a plug-in for the web browser, which provides the user interface for our tool. The second is an HTTP proxy, which is responsible for instrumenting JavaScript source code being delivered to the client-side.

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 4.3 and 4.4.

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 4.1, 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 4.3.

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 mutator 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 4.4.

4.2 Implementation Assumptions

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 5.

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 [33]. 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.

4.3 Firefox, Firebug and FireInsight

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.

Our FireInsight plug-in was developed on Firefox 3.0 and Firebug 1.4, respectively. Figure 4.2 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 4.2 (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

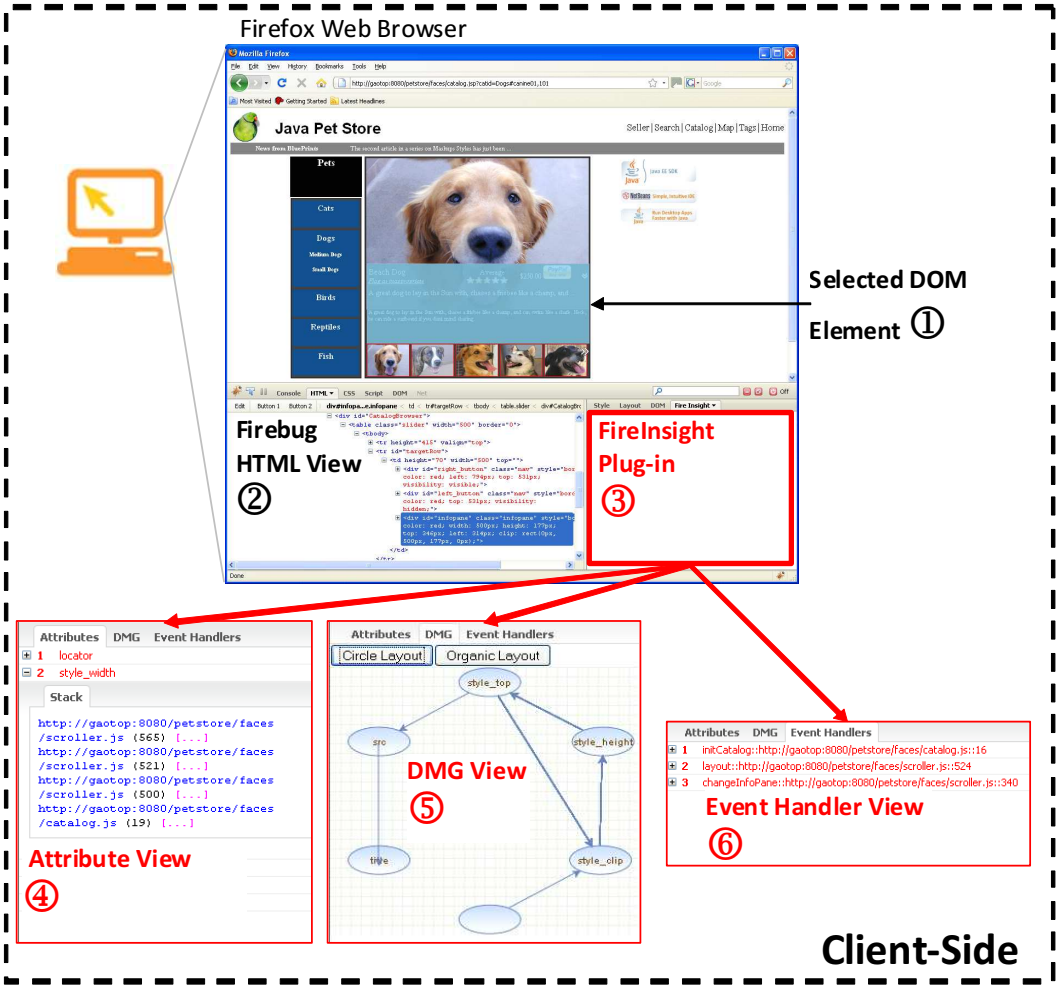


Figure 4.2: The FireInsight plug-in component acts as the user interface. The above diagram depicts how our tool integrates with the Mozilla Firefox browser and the Firebug add-on application. (1) The developer inspects the page and selects a DOM element. (2) Firebug automatically switches to its HTML 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.

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 4.2 (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 4.2 (4)). We call this the *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 4.2 (5)). We call this the *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 4.2 (5), we see an example DMG containing five DOM mutator nodes[2] 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 4.2 (6)). We denote this as the *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

[2]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.

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 [10]. All Firefox add-on applications, including Firebug, use XUL.

4.3.1 DOM Mutation Graph and MxGraph

To display DMGs within our plug-in we use a third-party JavaScript library called MxGraph [16]. 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.

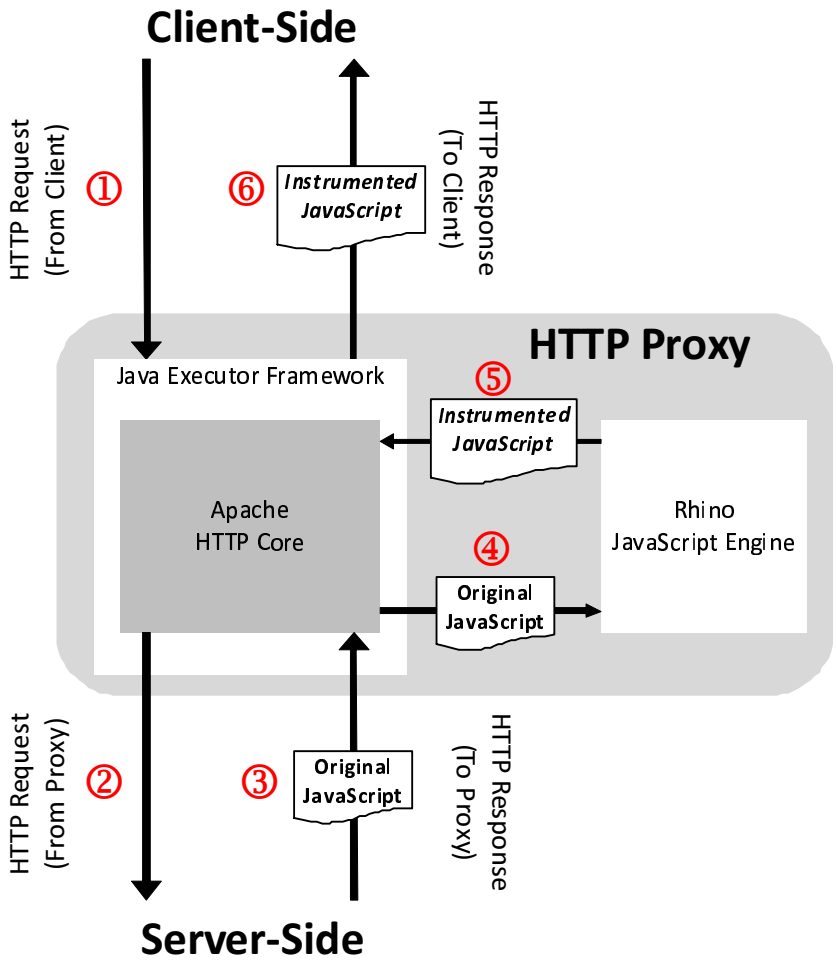


Figure 4.3: FireInsight's HTTP proxy component intercepts all communication between the web browser and the web server. The diagram depicts the scenario when a JavaScript file is requested by the web browser. (1) An HTTP request from the web browser is intercepted by the proxy. (2) The proxy creates 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.

4.4 JavaScript Instrumentation using an HTTP Proxy

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 *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 4.3 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 [29]. We leverage an existing HTTP communication framework created by the Apache Software Foundation, called HTTP Core [8], 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 [26].

Figure 4.3 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 `"javascript"`. Second, best practices state that JavaScript source code should be stored in their own separate files, with *.js extensions. As explained in Section 2.3, 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 [22]. We augmented the Rhino parser to inject our own analysis

code into the application source during the parsing process. As illustrated in Firgure 4.3, 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 `node.attribute=value`, or calls to the DOM API functions `createElement` and `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 `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.

4.5 Known Limitations

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 5. 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 4.2, 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 2.5.2, 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 poten-

tially 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.

Chapter 5

Results and Evaluation

In this chapter we evaluate FireInsight by applying it to our benchmark web application, JPS2.0. Our primary objective is to determine whether or not FireInsight can effectively model the UI-JavaScript mapping within JPS2.0. Our secondary goal is to investigate the interoperability of our tool with respect to JavaScript frameworks. In particular, we examine FireInsight's effectiveness at mapping UI behavior to source code, when the application logic is integrated with a JavaScript framework. In the case of JPS2.0, the application logic integrates with the Dojo JavaScript framework.

Instead of using a quantitative measurement approach, we evaluate our tool by presenting multiple case studies. Each case study involves a hypothetical scenario where the developer wants to modify a specific piece of JavaScript behavior in the JPS2.0 user interface. For each scenario we show how our tool might improve developer understanding. We utilize a narrative approach when presenting each case study to illustrate the obstacles that a JavaScript developer would face when attempting to understand a piece of functionality. After presenting those obstacles we show how FireInsight can be used to overcome them with less development effort.

Following the evaluation section we close the chapter by discussing some drawbacks to our approach and specifically some technical issues in our implemen-

tation. This will lead into our final chapter where we provide conclusions and future work for our project.

5.1 FireInsight Evaluation

To provide a comprehensive evaluation of FireInsight, we present three case studies, each centered on a different piece of JavaScript behavior in the JPS2.0 user interface. For clarity, we denote these units of JavaScript functionality in the user interface as *user interactions*. For each case, we begin the discussion by introducing a real-world scenario requiring the front-end developer to perform a software maintenance task.

We then discuss the program understanding challenges that the developer must overcome to complete the programming task. In Section 3.2, we looked at two major challenges. The first was determining which DOM elements are affected by the user interaction and locating the corresponding HTML code. The second was determining the correct JavaScript logic that mutates the DOM elements and understanding how the logic works. We saw that using Firebug's HTML inspection mode resolves the first obstacle.

However, the second challenge remains unaddressed. Thus, for each case study we focus on illustrating the problem of mapping the DOM elements to the corresponding JavaScript logic and understanding the control-flow for the user interaction. We end each case study by explaining how FireInsight alleviates this challenge by generating the UI-JavaScript mapping for the developer.

5.1.1 Case Study: RSS News Feed

We begin our evaluation by looking back at the RSS news feed scenario that was introduced in Section 3.2. The developer's objective is again to slow down the refresh rate of the news feed. Having already explored the scenario, we know that one major obstacle to using Firebug is it cannot assist the developer with locating the JavaScript logic driving the user interaction and visually modeling how the code

works.

Using FireInsight, the developer will begin by visually inspecting the page in browser view to locate the RSS news feed area. Since our tool integrates with Firebug, our page inspection mechanism is the same as Firebug's HTML inspection mode. Once the developer has selected the news feed area in the browser view, Firebug will show the corresponding HTML code for the DOM element. At this point the developer can open the FireInsight side-panel from within Firebug. This will show FireInsight's Attribute view, which lists the attributes of the currently selected DOM element that have a corresponding DOM mutator in the application code.

Figure 5.1 illustrates the current scenario and the interplay between (1) the DOM element selected in browser view, (2) the HTML code in Firebug, and (3) the execution context in FireInsight's Attribute view. We can see that there is only a single attribute that was found by FireInsight and it is called locator. This is actually not a standard HTML attribute. It is a specific value that is inserted by FireInsight's analysis code to record when a DOM element is first created. In this scenario, a news headline item corresponds to an `<a>` DOM element, which is first created at line 105 of rssbar.js. More importantly, looking down the call-stack, we can see the calling context for this DOM mutator. We can see the calling context was initiated on line 37 in rssbar.js. It would be useful to view the actual source code.

If the developer clicks on one of the entries in the execution call-stack, FireInsight opens the Source Code view that is a pop-up window containing the source code. Figure 5.2 shows an example of the Source Code view where the developer has clicked on the call-stack entry for line 105 of source file rssbar.js. FireInsight highlights the exact JavaScript statement and shows the line number at the bottom of the pop-up window. Using this feature, the developer can quickly pin point the code behind a specific DOM mutation within the browser view. Looking at the source

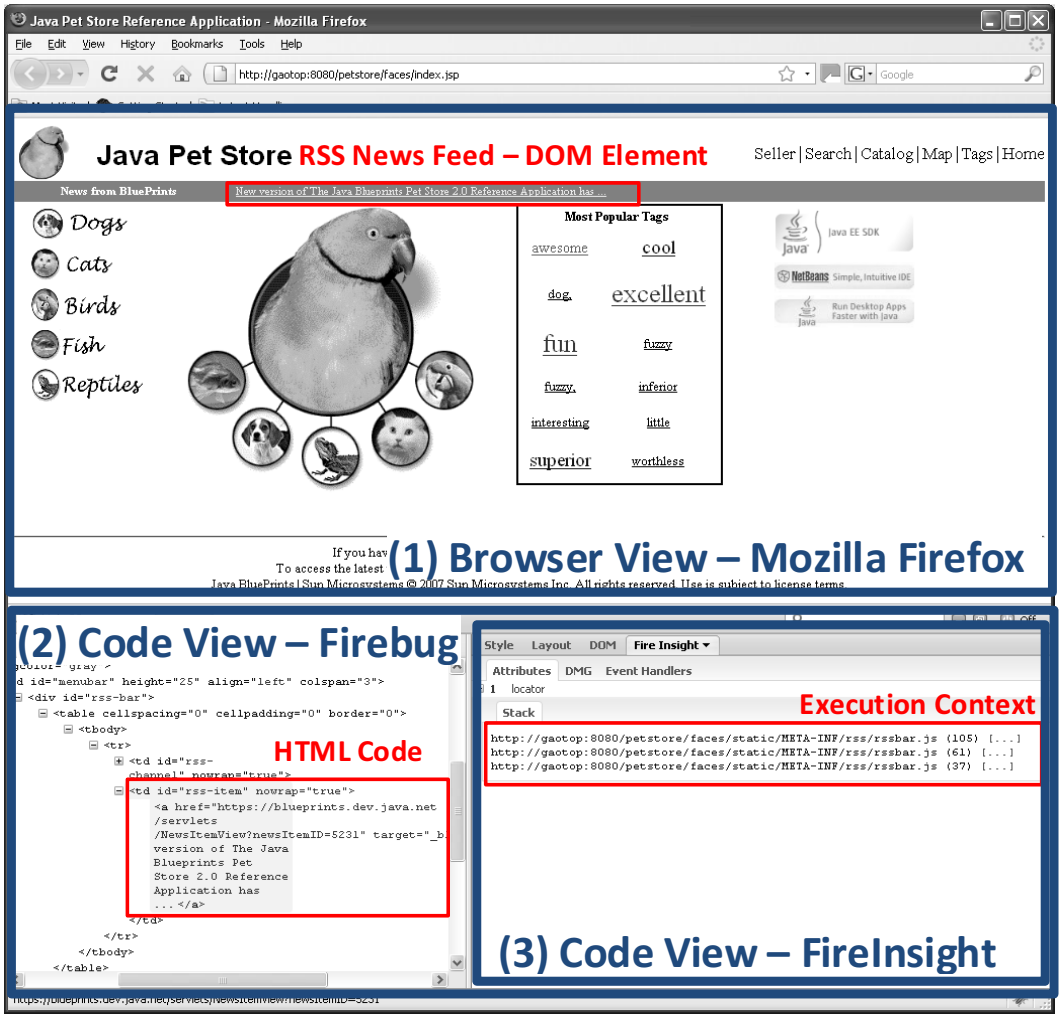


Figure 5.1: The above screenshot shows the FireInsight inspection mode being applied to the RSS news feed area on the JPS2.0 Home page. The FireInsight panel is showing the Attribute view.

Source of: file:///C:/Documents%20and%20Settings/Steven%20Gao/Application%20Data/Mozilla/Firefox/Pr...

File Edit View Help

```
            if (typeof desc != 'undefined') {
                content += "<p style='background-color:#FFEFD5;font-size:80%'>" + decodeURL(c
            }
            content += "</li>";
        }
        content += "</ul><br>";
        content += "<a href=" + jsonData.channel.link + " target='_blank'><i><b>...More on th
        childW.document.body.innerHTML = content;
    }

    function generateFeedView(title, site, nodeId) {
        var node = document.getElementById(nodeId);
        var aNode = document.createElement("a");
        aNode.setAttribute("href", titleLink);
        //aNode.setAttribute("onClick", "javascript:void(rss.openChild());");
        aNode.appendChild(document.createTextNode(title));
        if (node.hasChildNodes()) {
            node.removeChild(node.firstChild);
        }
        node.appendChild(aNode);
    }

    function generateHref (title, link, nodeId) {
        var node = document.getElementById(nodeId);
        var aNode = document.createElement("a");
        aNode.setAttribute("href", link);
        aNode.setAttribute("target", "_blank");
        aNode.appendChild(document.createTextNode(title));
        if (node.hasChildNodes()) {
            node.removeChild(node.firstChild);
        }
        node.appendChild(aNode);
```

DOM Mutator

Line 105, Col 1

Line Number

Figure 5.2: The above screenshot shows an example of the FireInsight Source Code view. This is a pop-up window that appears when the developer clicks on a specific entry in an execution call-stack. The Source Code view displays the source code corresponding to the call-stack entry, with the exact JavaScript statement highlighted and the line number shown at the bottom of the window.

code in Figure 5.2, we can see that the `createElement()` function is used to create the `<a>` element, representing the news headline. We can see that the `href` and `target` attributes are set for the `<a>` element. A text node, containing the actual headline text, is also created and appended as a child to the `<a>`. These mutations are not shown in the Attribute view, because our current version of FireInsight does not look for `setAttribute()` and `appendChild()`[1]. Therefore, our HTTP proxy does not add analysis code to record those mutations.

If the developer views the source code on line 37 it will become clear that the calling context is part of an initialization procedure called `getRssInJson()`. Looking at the HTML page containing the RSS news feed DOM element (i.e. banner.jsp) we can see a block of JavaScript code containing a call to Dojo to register `getRssInJson()` as an event handler for the on load event. This means `getRssInJson()` will get executed through Dojo when the page first loads. Thus, the developer gains a deeper understanding of the application logic by viewing the execution context.

However, the developer still needs to figure out the control-flow in cases when a user interaction involves multiple DOM mutations. In this example, the initial invocation of the DOM mutator is not actually the calling context that we want, because it is executed once and then never again. There is in fact a second calling context that is executed to handle updating the news feed area once the page has fully loaded. The reason we cannot see it is because it is invoked by a separate event handler.

For this reason FireInsight also models the control-flow for the event handlers attached to the page. Once the developer has inspected a DOM element on the page, FireInsight will locate all of the event handlers that affect the selected DOM element. A listing of the event handlers that affect the currently selected DOM element will

[1]Since we had to modify the Rhino parser source code directly, we found some DOM API methods easier to capture than others. Namely, createElement() was straightforward while appendChild() and setAttribute() were not.

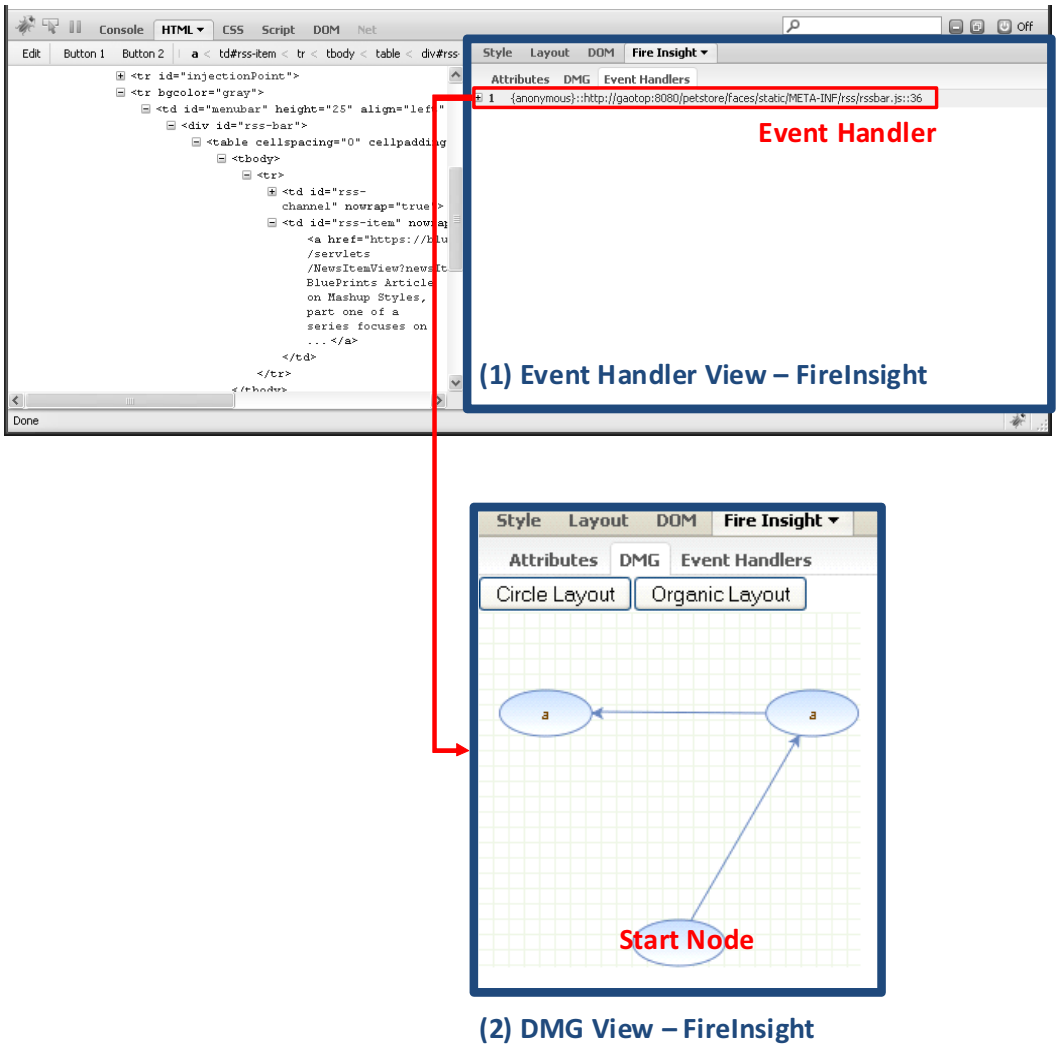


Figure 5.3: The above screenshot shows an example of the FireInsight Event Handler and DMG views. (1) The Event Handler view shows a listing of event handler functions that affect the currently selected DOM element. (2) Clicking on one of the event handlers will take the developer to the DMG view, which will display a graphical model containing all DOM mutators that are invoked by the event handler.

appear in FireInsight's Event Handler view.

Figure 5.3 illustrates what the developer would see in FireInsight's Event Handler view for the RSS news feed example. In this case, there is only one event handler, an anonymous function declared on line 36 of rssbar.js. If the developer

clicks on this event handler, FireInsight will switch from Event Handler view to DMG view, as shown in Figure 5.3. In the DMG view, we can see a graphical representation of the control-flow for this event handler. Note that there is always an empty node in the graph, because it is used to indicate the starting point of the control flow. In this example, there are two distinct DOM mutator nodes, both of which create `<a>` elements.

The developer can right-click on a node and see its execution context, just like in the Attribute view. The developer can then click on an entry in the call-stack, and FireInsight will show the source code, again just like in the Attribute view. In this example, if the developer explores the execution context for both nodes in the DMG, it will become apparent that only one of them affects the headline section of the RSS news feed area. Figure 5.4 shows the two sections of the RSS news feed area and how they are related to the DOM mutators represented in the DMG. Specifically, we are interested only in the "rss-item" section, which contains the news headline, and not the "rss-channel" section.

The first node in the DMG mutates the "rss-channel" section, and the second node affects the "rss-item" section. However, when the developer jumps into the JavaScript code to examine the DOM mutator for the "rss-channel" section it will become clear that it is not the section of code that we want. This is because the code does not contain any logic to periodically refresh the news headline. We know from observing the browser view of the page that the news headline is supposed to get updated. This means that neither DOM mutator node is actually what we are looking for. At first glance, it may appear that FireInsight missed a DOM mutator during the instrumentation of the application code, when in fact it did not.

The second DOM mutator for the "rss-item" section belongs to a separate event handler. That is, the second execution context is part of another anonymous event handler. If the developer waits for the news feed area to refresh in real-time on the page, and then inspects the same area again, this time, FireInsight will show a

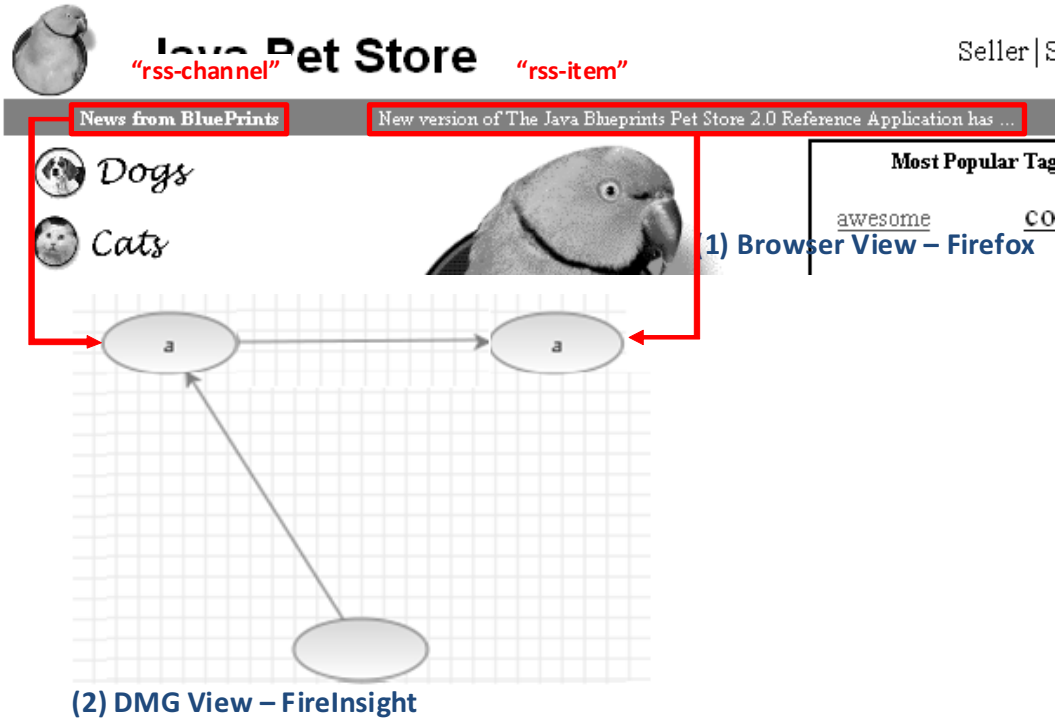


Figure 5.4: The above screenshot illustrates how the DMG for the news feed relates to the sections on the page.

different call-stack for the locater. Recall that DOM mutators are recorded in real-time as JavaScript behavior is executing on the page. The RSS news feed cycles through a set of headlines without requiring user input. This is a subtle difference compared to more common user interactions that execute based on user events. As the news feed automatically refreshes. FireInsight records the corresponding DOM mutations. After the initial DOM mutation, all subsequent DOM mutations of the "rss-item" area will be grouped under the second event handler.

Figure 5.5 shows the call-stack for the locator value after the developer has inspected the "rss-item" section again. This time clicking the bottom level entry in the call-stack shows the second anonymous event handler. Viewing the source code for the event handler on line 137 of rssbar.js shows the default refresh rate is 500. From here the developer can increase the value to slow down the refresh rate.

Similarly, the DMG has changed as well, as shown in Figure 5.6. The graphical model now contains a single node, instead of two as we saw in Figure 5.4. The single node has a self-directed edge, indicating that the control-flow entails invoking the same DOM mutator repeatedly, without any transitions to other DOM mutators. This DMG is consistent with the conceptual model we presented for the RSS news feed in Section 3.2, namely Figure 3.6.

5.1.2 Case Study: Catalog Browser Info Pane

Our next case study involves a complex user interaction on the JPS2.0 Catalog Browser page. This page allows the user to peruse pet listings stored in the JPS2.0 database. On the left-hand side, the navigation menu displays the categories of animal types (e.g. cats) and sub-types (e.g. hairy cats). On the right-hand side, the main panel allows the user to explore pet listings within the current category sub-type. A gallery of thumbnail images located at the bottom of the main panel allows the user to scroll through pet listings. The user can view more details for each pet by clicking the thumbnail image. This action loads the selected pet's information into

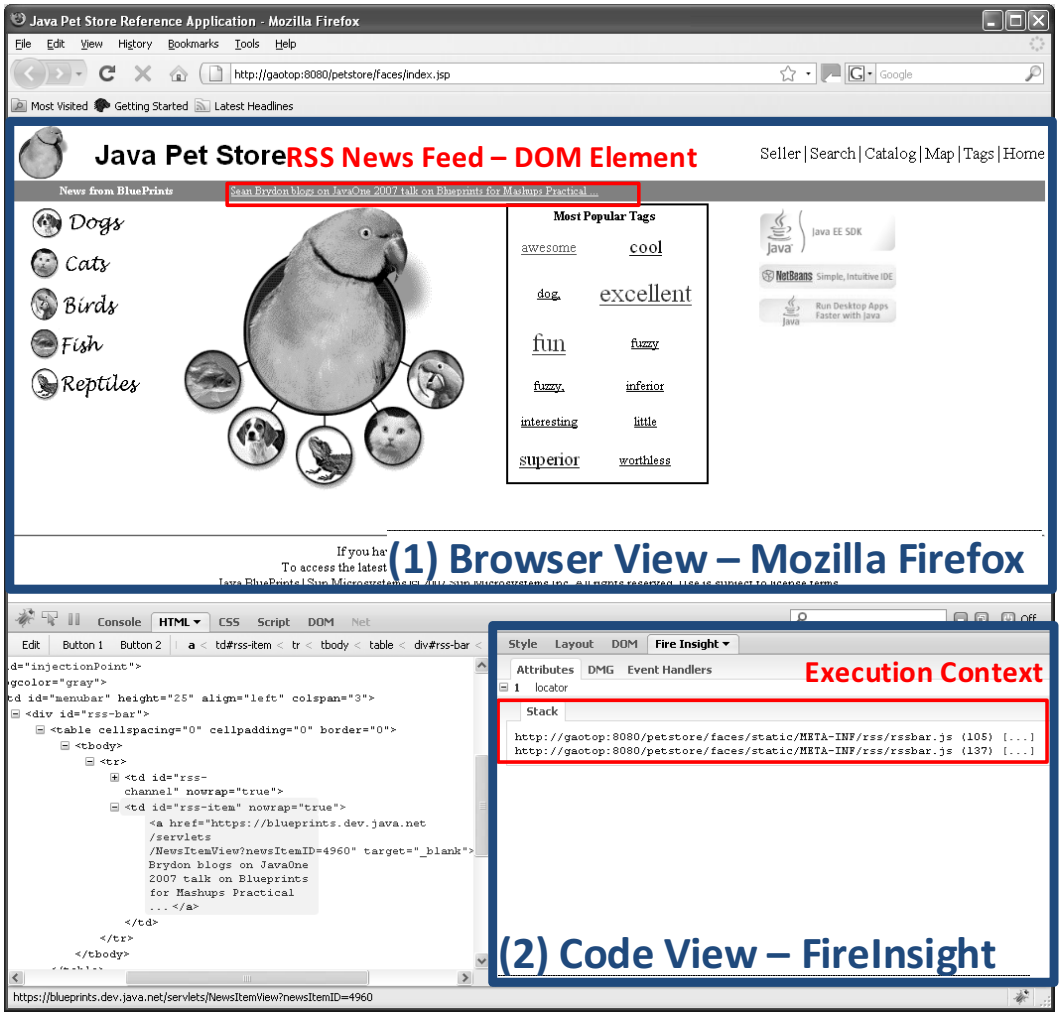


Figure 5.5: The above screenshot shows the FireInsight inspection mode being applied to the RSS news feed area for a second time. The Attribute view shows a different execution context this time.

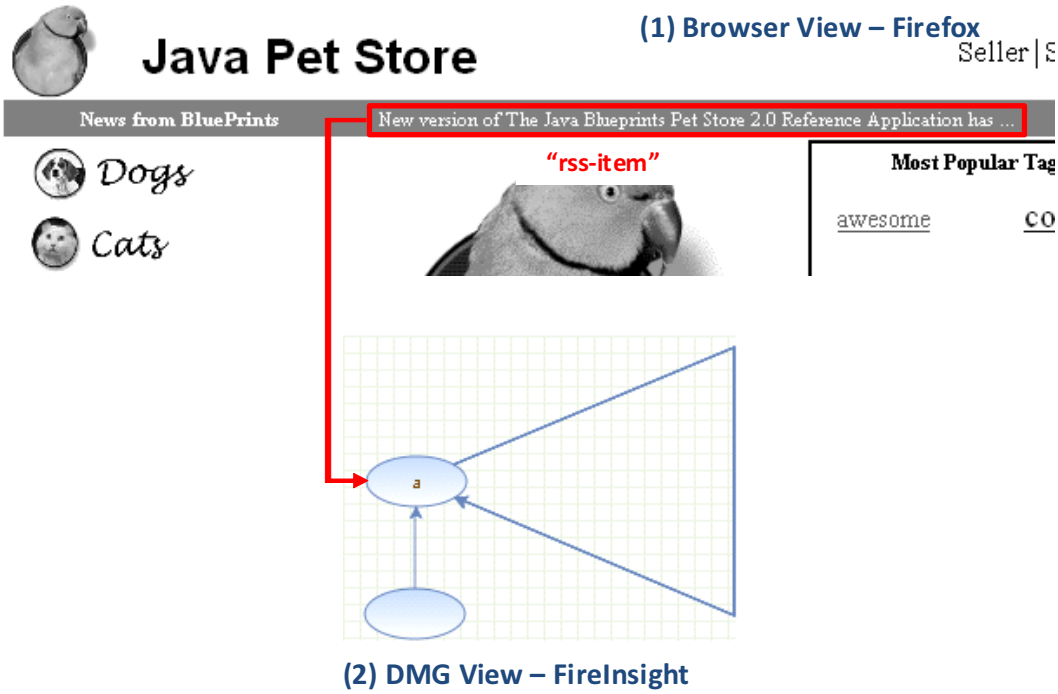


Figure 5.6: The above screenshot shows the FireInsight inspection mode being applied to the RSS news feed area for a second time. The DMG view shows a different event handler this time.

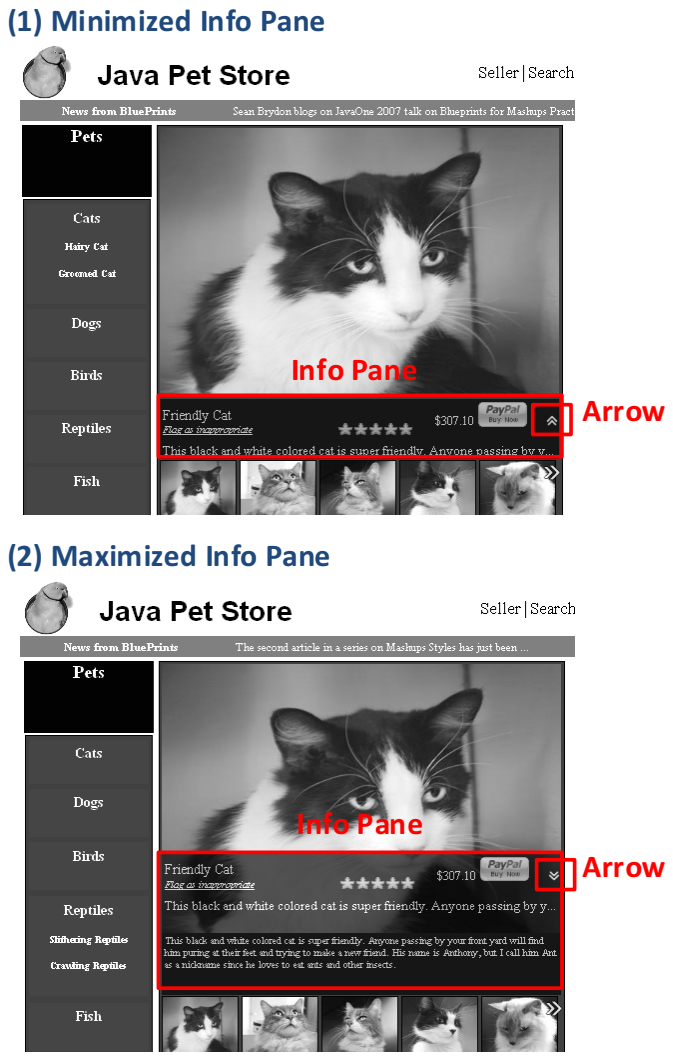


Figure 5.7: The above screenshot shows the JPS2.0 Catalog Browser page. In particular it shows the two states of the Info Pane area: (1) minimized, and (2) maximized.

the main panel area, directly above the image gallery. Two items are loaded into the main panel area: (1) a large snapshot image of the pet, (2) summary information about the pet. The summary information is loaded into an Info Pane area, which is located above the image gallery but in front of the snapshot image.

The Info Pane area can be in a minimized state or a maximized state, as

shown in Figure 5.7. When the Info Pane is minimized, it is partly hidden behind the image gallery. This puts emphasis on the large snapshot image of the pet. An arrow image on the right-hand side of the Info Pane is pointing upward, indicating that the pane can be expanded. When the Info Pane is maximized it slides up and obscures the large snapshot image. This allows the user to read the summary information and learn more about the selected pet. In this state, the arrow on the right-hand side is pointing downward, indicating that the pane can be closed.

When transitioning between minimized and maximized states, the Info Pane is animated to slide upward and downward. The user can control the state of the Info Pane by clicking on the arrow image. This toggles the Info Pane from minimized to maximized, and vice versa. Thus, the arrow image is actually a button. For this reason, we denote the arrow as the toggle button.

In our scenario, the developer is assigned to modify the user interaction so that the Info Pane expands to cover the entire snapshot image. In other words, we want to increase the height of the Info Pane when it is maximized. Perhaps some of the pet summaries have long descriptions that require the Info Pane area to be larger when it is maximized.

Using Firebug, the developer can inspect the Info Pane to discover that the corresponding DOM element is a `<div>` tag on the page with an id attribute value of "infopane". Although without FireInsight, the developer again faces the challenge of manually mapping the DOM element to the JavaScript source code. Examining all the included JavaScript source files on the page (i.e. catalog.jsp) reveals that there are 1,542 lines of code. This number excludes the Dojo source code. Manually searching the source code for references to "infopane" in order to understand the logic will be very time consuming.

If the developer uses FireInsight's inspection mode, it becomes immediately apparent which of the attributes for the Info Pane `<div>` are being mutated. Figure 5.8 shows the list of attributes that are affected by DOM mutators. Specifically,

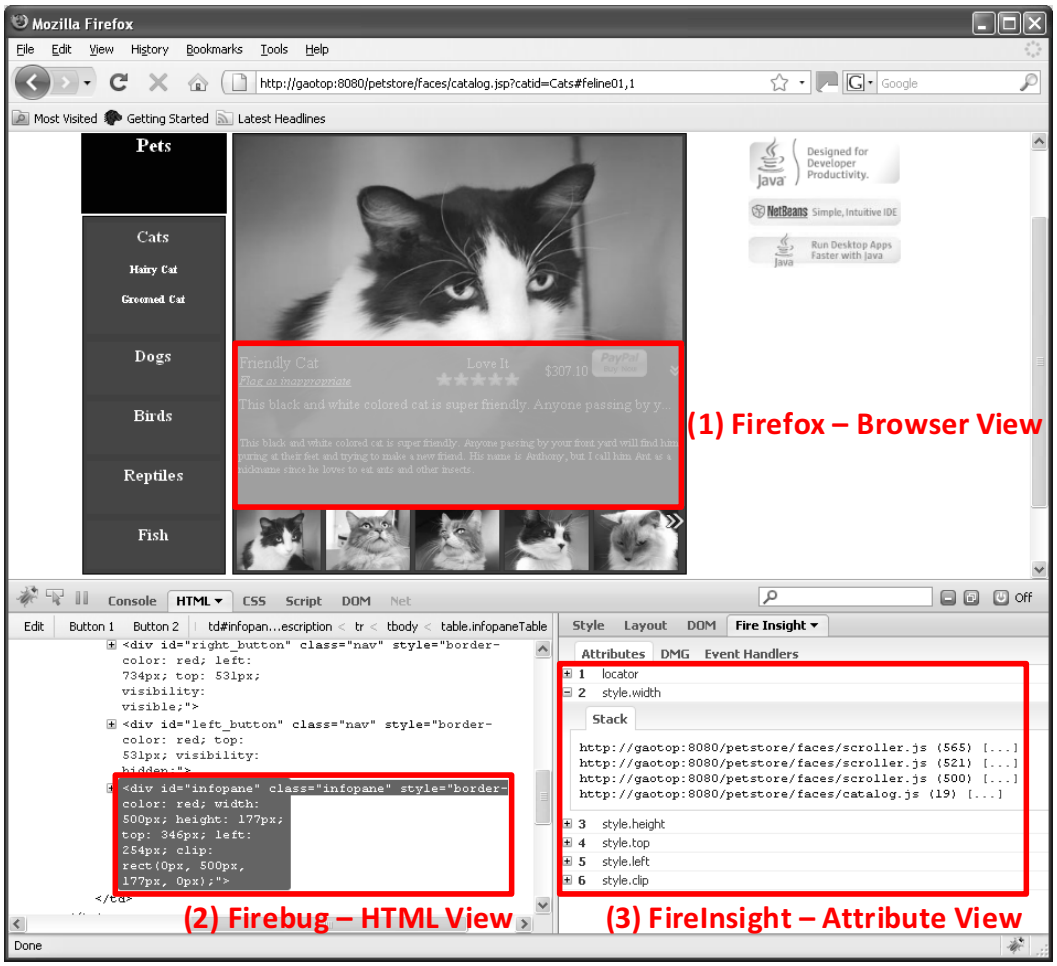


Figure 5.8: The above screenshot shows the Info Pane area being selected using the FireInsight inspection mode. For the Info Pane, the developer can see the association between: (1) the DOM element in the browser view, (2) the HTML code in the HTML view, (3) the element attributes that have been mutated in the Attribute view.

`style.width`, `style.height`, `style.top`, `style.left`, and `style.clip` each get mutated as part of the user interaction. We can see in the figure that the call-stack for `style.width` has been expanded. Examining the call-stacks reveals that all of the DOM mutators occur within the function `createInfoPane()`, which is located in scroller.js. However, this function does not appear to be responsible for the toggle animation between the Info Pane's minimized and maximized state. Similar to the situation we saw with the RSS news feed, there appears to be one event handler responsible for the initialization of the DOM element, and another is responsible for animating it.

Since the Info Pane transitions between two different states the sequence of DOM mutators that get invoked during the user interaction will depend on its current state (i.e. minimized or maximized). Due to the complexity of this behavior, we need to examine the DMG(s) for better program understanding. Since FireInsight records a history of the DOM mutators that have executed so far, the DMG will change in real-time accordingly.

We illustrate how the DMG for the Info Pane behavior evolves over time in Figure 5.9 and Figure 5.10. When the Catalog Browser page first loads in the browser, the Info Pane is minimized and no DOM mutators related to the Info Pane behavior have been executed yet (Figure 5.9 - Step 1). Note in the initial state, FireInsight's Event Handler view does not display the event handler function that we are looking for. We can verify this by exploring the DMG for `initCatalog()` and observing that the code is only responsible for rendering the initial view of the Info Pane. In contrast, we are interested in how the Info Pane is animated from minimized state to maximized state and vice versa.

If we now click on the toggle button in the browser view, the Info Pane will expand and slide upward, causing DOM mutators to execute in order to animate the sliding effect (Figure 5.9 - Step 2). Looking at the Event Handler view, we can see an event handler function called `changeInfoPane()`. If we examine its DMG we

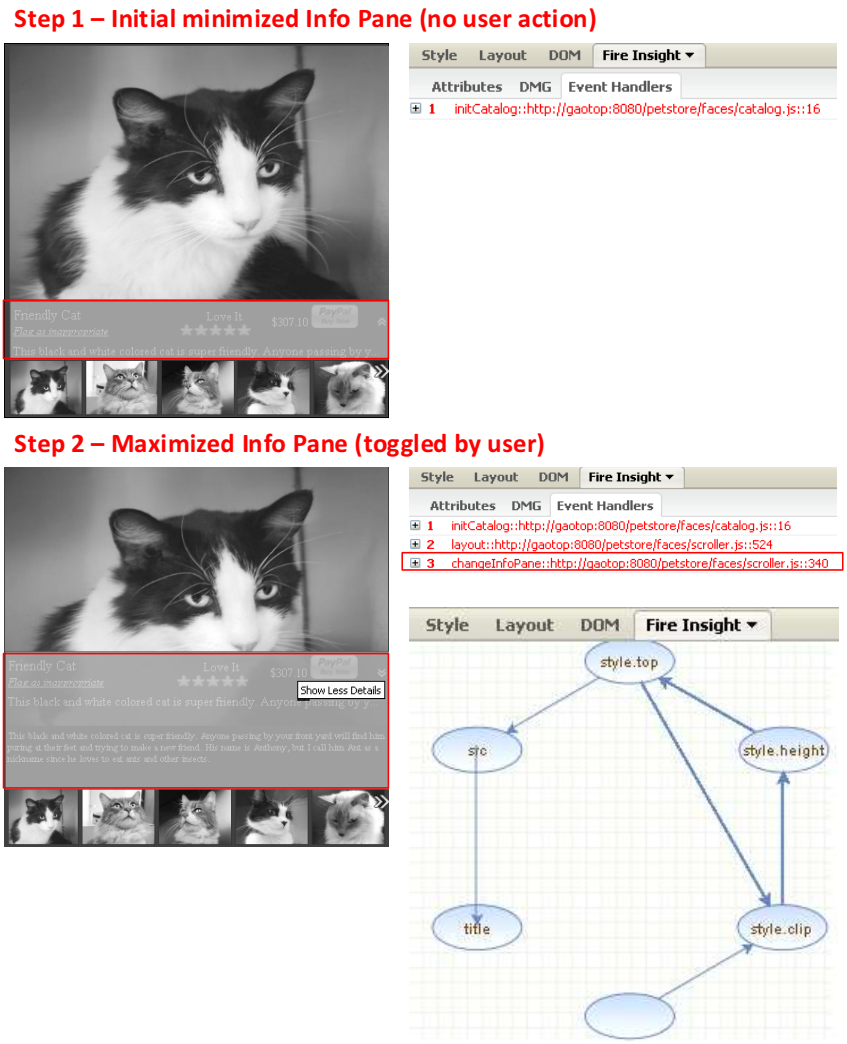


Figure 5.9: The above diagram shows how FireInsight's Event Handler and DMG views evolve in real-time as the user interacts with the Info Pane. (1) The initial state of the Info Pane is minimized; the Event Handler view does not show any function related to animating the sliding of the Info Pane and there is no corresponding DMG. (2) The user toggles the Info Pane to its maximized state; the Event Handler now shows the function changeInfoPane(). A corresponding DMG has been generated.

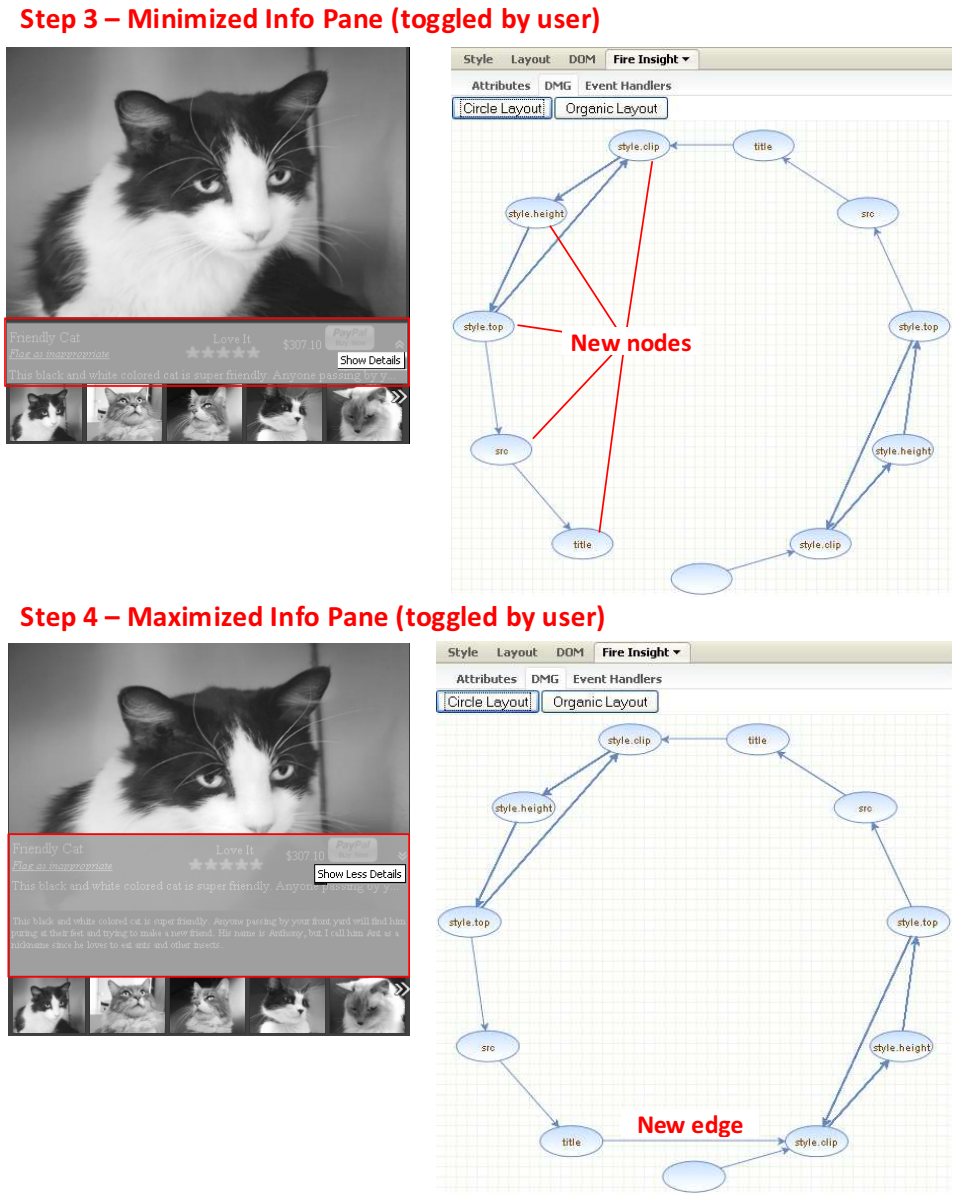


Figure 5.10: The above diagram shows how FireInsight's Event Handler and DMG views evolve in real-time as the user interacts with the Info Pane. (3) The user toggles the Info Pane back to its minimized state; the corresponding DMG has evolved to show additional DOM mutator nodes. (4) The user toggles the Info Pane back to its maximized state; the corresponding DMG has been fully generated.

will see that it is responsible for animating the maximization of the Info Pane. The DMG shown in Figure 5.9 - Step 2 contains a cycle between the nodes `style.clip`, `style.height` and `style.top`. This corresponds to an execution sequence that mutates the three aforementioned attribute repeatedly in a loop. That is how the JavaScript behavior animates the Info Pane sliding upward. The animation ends with the `title` being set to the string "Show Less Details". The `title` attribute corresponds to text that appears when the user hovers the mouse over the toggle button, which is shown in Figure 5.9 - Step 2.

If we now click the toggle button again, the Info Pane will compress and slide downward. Figure 5.10 - Step 3 shows that the DMG has evolved to contain a new set of DOM mutator nodes. It now has double the number of nodes, excluding the empty start node. The new nodes represent the opposite animation effect, namely to minimize the Info Pane. Notice that this new group of nodes also has a cycle as well, which is the looping of DOM mutators to animate the Info Pane sliding downwards. Looking at the DMG, it is straightforward to see how the control-flow moves from one DOM mutator to the next and understand how this corresponds with the actual user interaction in the browser view. After the second button click, the Info Pane has essentially completed a full cycle of its animation but has not returned back to its initial state yet.

If we click the toggle button for a third time, the event handler will execute the first five mutator nodes in the DMG again, representing an animation of the Info Pane sliding upwards. In order to visit those nodes again we need an additional link in order to transition from the final node in the graph (a DOM mutator to change the `title` attribute) to the beginning of the graph (a DOM mutator to alter the `style.clip` attribute). Figure 5.10 - Step 4 shows this final edge has been added to the DMG after the toggle button is clicked for the third time.

Armed with this new understanding of how the Info Pane user interaction works, our developer can now easily modify the behavior so that the Info Pane

covers the entire snapshot image when it is maximized. Specifically, looking at the first half of the DMG for `changeInfoPane()` we can view the source code, by right-clicking any of the graph nodes and then clicking the top level entry in the call-stack. This will show the function that maximizes the Info Pane is called `maxmizeInfoPane()`[2]. The function animates the Info Pane by incrementing the values for `style.clip`, `style.height`, and `style.top`. Making a delayed call to `changeInfoPane()`, which when executed invokes `maxmizeInfoPane()` again. This cycle only stops when the Info Pane has reached a predefined maximum height, as specified by the variable `INFOPANE_EXPAND_HEIGHT`. Therefore, the developer simply needs to increase the value of `INFOPANE_EXPAND_HEIGHT` to achieve the objective for this scenario. Figure 5.11 shows the final result after the developer has increased the value of `INFOPANE_EXPAND_HEIGHT`.

It is important to mention that our Info Pane case study represents the same scenario Li and Wohlstadter used for evaluating their Script Insight tool. Comparing our full DMG (Figure 5.10 - Step 4) with theirs (Figure 6 of [21]) we can see that our graphs are not identical. The first difference is the ordering of nodes within the cycle that is responsible for animating the sliding effect. In our DMG the cycle has the order `style.clip` - `style.height` - `style.top`, whereas in Figure 6 of [21] the order is `style.height` - `style.top` - `style.clip`. Examining the code for the event handler `changeInfoPane()`, we can see that two auxiliary functions are used to animate the Info Pane. The first is the aforementioned `maxmizeInfoPane()`, which slides the pane upwards. The second is a function called `minimizeInfoPane()`, which slides the pane downwards. The source code for both functions clearly shows that the ordering of the DOM mutator statements is `style.clip` - `style.height` - `style.top`. Therefore, our ordering of the nodes is correct.

The second difference is our DMG includes an additional state which represents the DOM mutator for the `title` attribute. This is missing from the DMG in

[2]Note that the function name is misspelled. This is unfortunately a mistake within the JSP2.0 codebase.

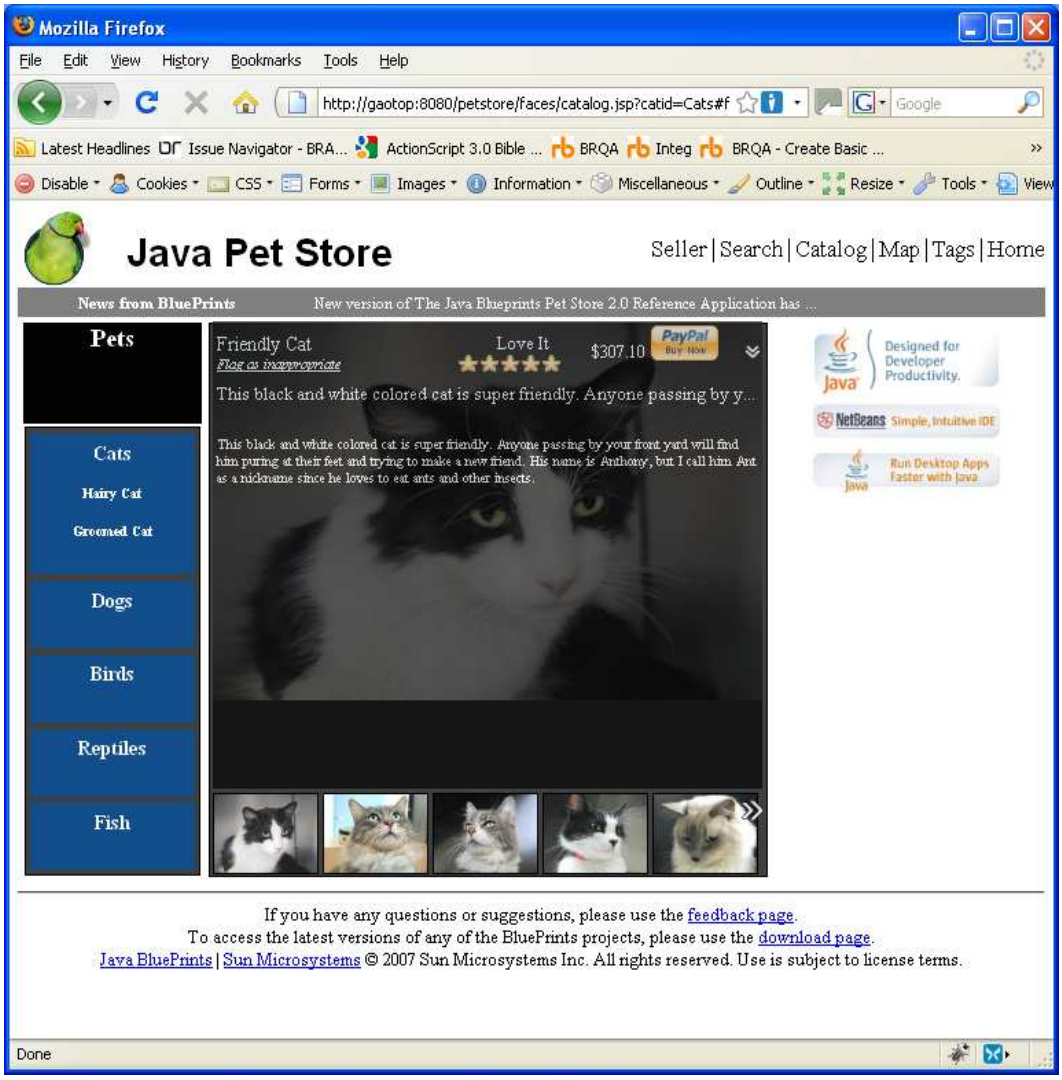


Figure 5.11: The above screenshot shows the final result after the developer has finished modifying the maximum allowable height for the Info Pane.

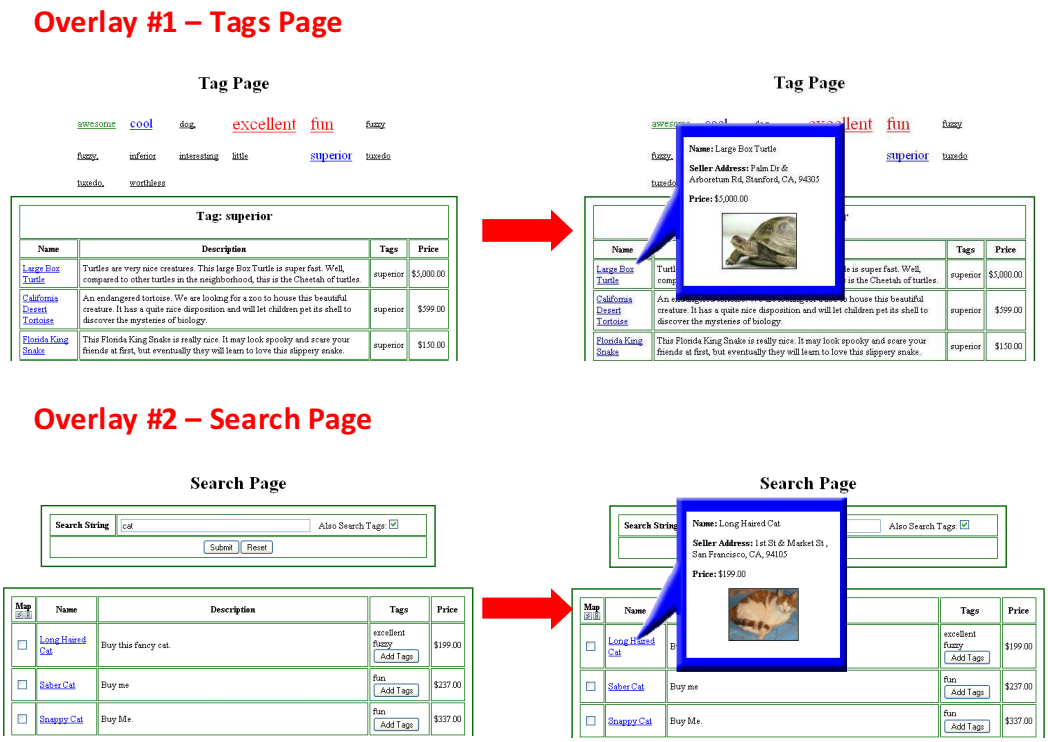


Figure 5.12: The Overlay Pop-up behavior is present on both the Search and Tags pages. The behavior produces a pop-up window over the existing page, which displays additional information regarding a pet listing.

Figure 6 of [21]. Once again examining the code verifies that the `title` is indeed mutated, thus our version of the DMG is correct.

5.1.3 Case Study: Pet Info Overlay Pop-up

For our final evaluation we explore a case study involving Overlay Pop-ups, which are used to display additional details for a pet listing. The Overlay Pop-up behavior is found on both the Tags page (tag.jsp) and the Search page (search.jsp), as shown in Figure 5.12. The Tags page displays a listing of results when the user performs a tag query, which involves selecting a tag value from a set of preexisting tags. For example, clicking on the "excellent" tag will return all pet listings associated with that tag descriptor. Similarly, the Search page displays a listing of results when

the user performs a keyword query. In each case, the returned results represent pet listings. Both pages display the name, description, associated tags (if any), and price for each pet listing. The name field contains a URL link which directs the user to the Catalog Browser page and automatically loads the particular pet that the user clicked. All other fields for a result are static text.

The overlay interaction is triggered by a user event. Specifically, when the mouse pointer is moved over the name field for a pet listing an overlay window pops up in front of the search results. The purpose of the pop-up is to display some additional information about the pet, although some fields are repeated. The overlay window displays the pet name, address of the seller, the price, and most importantly an image of the pet. It is important to indicate that the overlay behavior is triggered by a mouse over event not a mouse click event, i.e. the mouse is placed over the name field but does not click the name. If the user clicks the name field, the user will be hyperlinked to the Catalog Browser page and the browser reloads the page.

In our scenario, the developer is assigned to fix a bug that has been discovered in the existing overlay behavior. The problem is the Overlay Pop-up contains a misplaced arrow image, which overlaps the pop-up and obscures the pet listing details. An additional obstacle in this scenario is the bug occurs only in a specific situation. Figure 5.13 shows there are two types of overlay windows. Type (A) has a downward pointing arrow, while type (B) has an upward pointing arrow. We can see from the figure that it is type (B) that contains the logic bug.

We assume our developer is not familiar with the code base or at least does not have an understanding of the entire overlay behavior. Since the overlay behavior is present on both the Search and Tags pages, we can use either one to reproduce the bug and investigate the issue. The Search page has 866 lines of JavaScript code, while the Tags page has 839 lines of code; neither line count includes the Dojo source code. Since both are relatively similar in terms of code size, we make an arbitrary choice and pick the Search page to use for debugging.

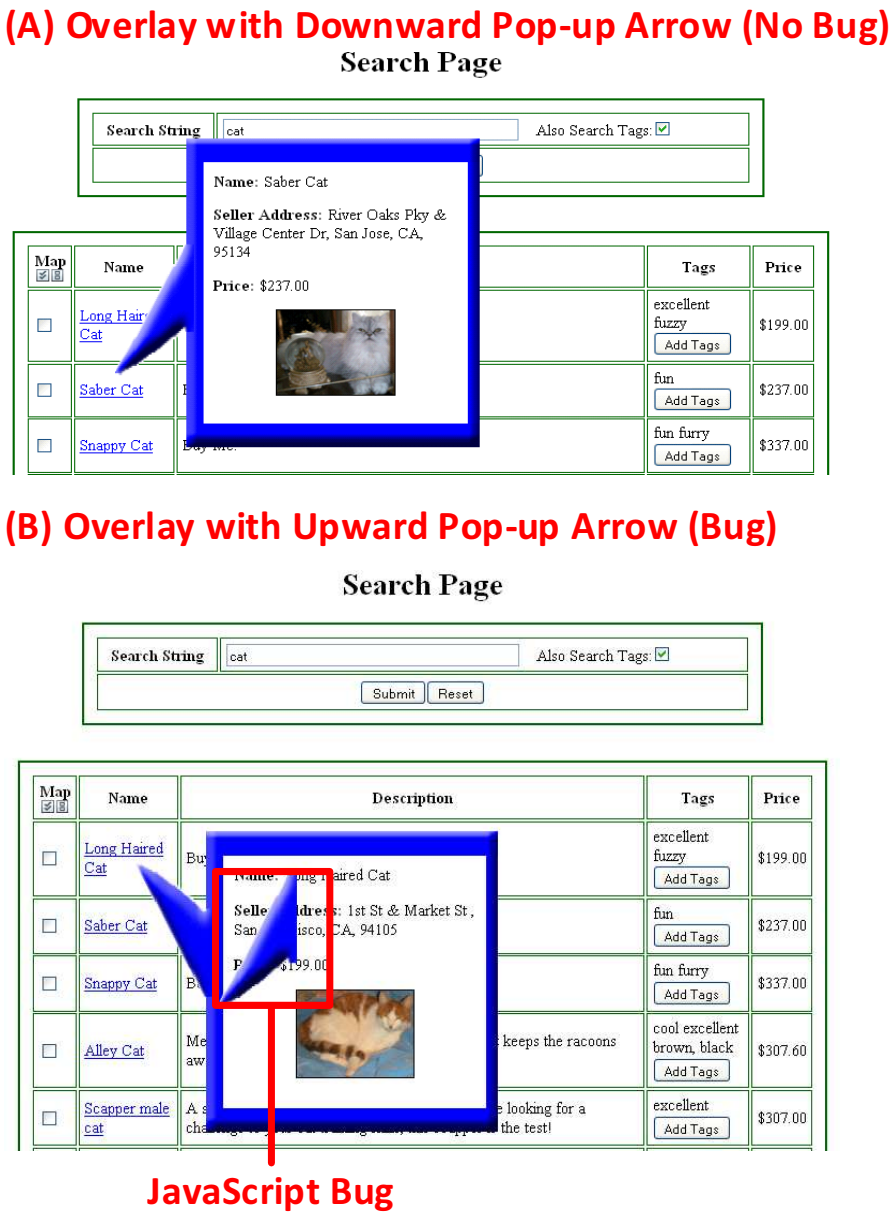


Figure 5.13: There are two possible types of overlay windows: (A) An overlay with a downward pointing arrow, (B) An overlay with an upward pointing arrow. The JavaScript bug only occurs for type (B).

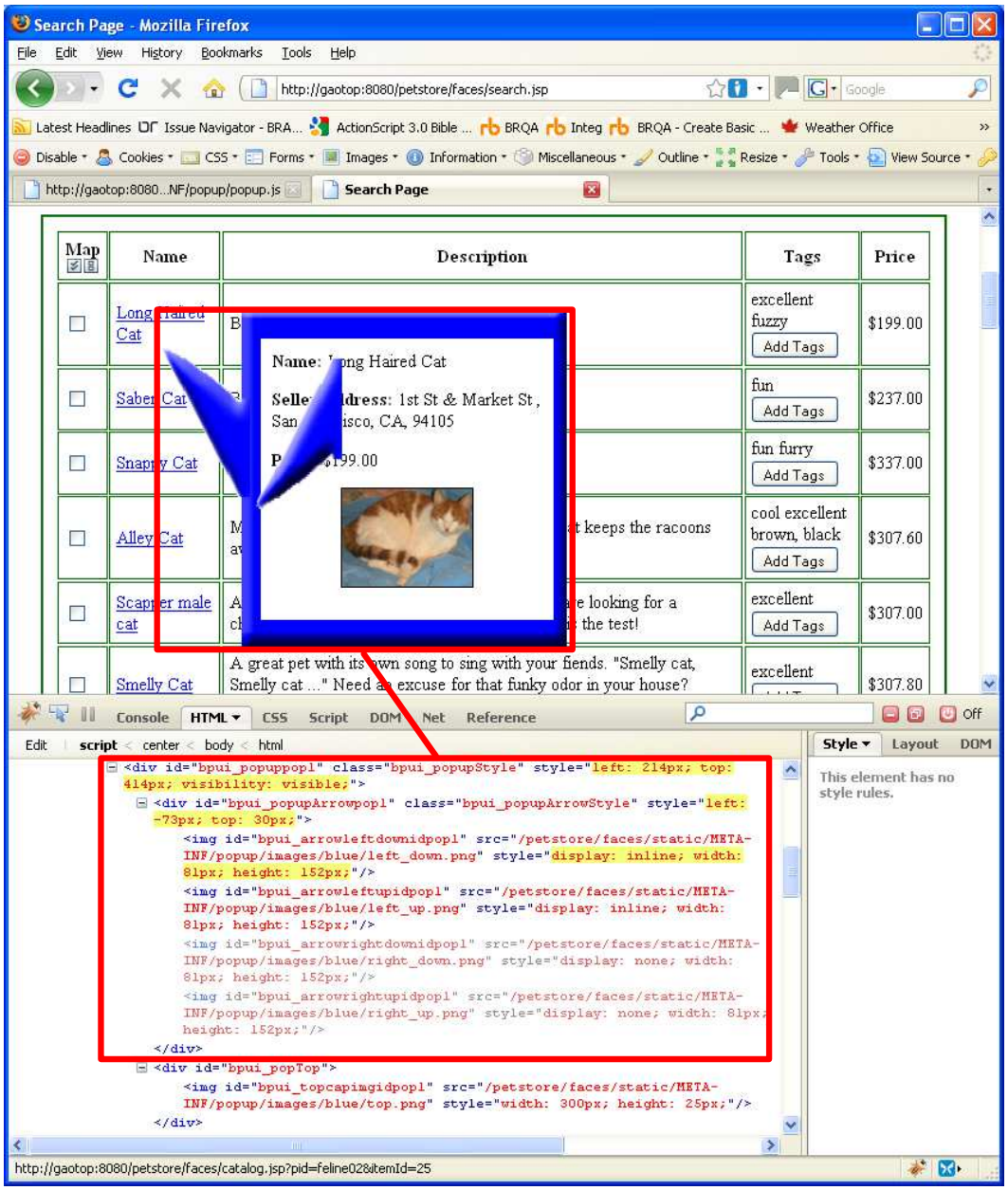


Figure 5.14: Since the Overlay Pop-up behavior is triggered by a mouse over event and not a mouse click event, we cannot select the overlay DOM elements using inspection mode in the standard way. Instead of using the standard point and click to select the overlay area, we must use Firebug's inspection highlighting feature. We can see in the above screenshot the mutated DOM elements are highlighted in yellow.

Because the overlay behavior occurs only on a mouse over event and not a mouse click event, we cannot use the inspection mode from Firebug/FireInsight to directly select the name field to see the corresponding code. In fact, since the overlay simulates a pop-up window, it has its own DOM elements that exist separately from the name field[3]. Thankfully, there is a solution to this problem using Firebug. As shown in Figure 5.14, we can use Firebug's inspection mode highlighting to spot the DOM elements that are related to the overlay behavior. When a DOM element is mutated on the page Firebug automatically highlights the corresponding HTML elements in the HTML code view. This allows our developer to observe which section of the HTML code is mapped to the overlay in the browser view. That said, again there is no mapping back to the JavaScript code.

Using FireInsight, the developer can select the DOM elements corresponding to the overlay and view the generated DMG(s). There are two event handlers of interest, both of which are anonymous functions. The first function occurs on line 7 of source file popup.js and is responsible for hiding the overlay window. The function is assigned to the variable `bpui.popup.hide`. The second function occurs on line 51 of source file popup.js and is responsible for showing the overlay. The function is assigned to `bpui.popup.showInternal`. We can map these two functions back to the HTML code; more specifically to the name field DOM element. Figure 5.15 shows how both of the functions are registered for event handling on the name field element; `bpui.popup.hide` is registered to handle the mouse out event and `bpui.popup.showInternal` is registered to handle the mouse over event.

For this scenario, we know that the bug occurs when the overlay is displayed, not when it is hidden. Consequently, we are interested in `bpui.popup.showInternal` exclusively. But there are two different versions of the Overlay Pop-up and only type (B) exhibits the incorrect behavior, we must ensure that we interact with the type

[3]We say that the overlay simulates a pop-up window, because it is not actually a separate browser window. The overlay simply displays a piece of HTML on top of the existing elements on the page.

Style Layout DOM Fire Insight

Attributes DMG Event Handlers

1 {anonymous}::http://gaotop:8080/petstore/faces/static/META-INF/popup/popup.js::7

2 {anonymous}::http://gaotop:8080/petstore/faces/static/META-INF/popup/popup.js::51

```
<tr>
        <td class="itemCell">
                <input type="checkbox" name="mapSelectedItems" value="25" />
        </td>
        <td class="itemCell">
                <a  href="/petstore/faces/catalog.jsp?pid=feline02&itemId=25"
                    onmouseover="bpui.popup.show('pop1', event, '25')"
                    onmouseout="bpui.popup.hide('pop1')">
                    Long Haired Cat
                </a>
        </td>
        <td class="itemCell">Buy this fancy cat.</td>
        <td class="itemCell">
                <span id="ITEMID_TAGS_25">excellent     fuzzy</span>
                <br />
                <input type="button" value="Add Tags"
                        onclick="addTags(event, 'Long Haired Cat', '25')" />
        </td>
        <td class="itemCell">$199.00</td>
</tr>
```

Figure 5.15: The Overlay Pop-up behavior has two event handlers. The first is a function (bpui.popup.hide) is registered to handle mouse out events. The second function (bpui.popup.showInternal) is registered to handle mouse over events.

(B) overlay. Otherwise, FireInsight will capture execution context data for the type (A) overlay and generate a DMG model that contains DOM mutator nodes that we are not interested in.

Figure 5.16 shows the generated DMG for the type (B) overlay behavior. For illustration purposes we have mapped the first four nodes directly to their corresponding mutator statements in the source code (popup.js). The other nodes in the DMG are related to attributes which affect positioning but not visibility. Therefore our developer should be able to deduce that the bug is related to one of the first four nodes in the DMG. Through some testing by making changes to the four statements and seeing the resulting behavior in browser view, our developer should be able to discover which of the four nodes produces the error. The developer should easily discover that setting an object's `style.display` attribute to "inline" causes it to become visible. Setting it to "none" causes the object to become hidden. We can see from Figure 5.16 that the second DOM mutator is the erroneous statement.

5.2 Challenges and Drawbacks

5.2.1 JavaScript Frameworks

One significant challenge we faced when identifying event handler functions in JPS2.0 was dealing with Dojo function calls. JavaScript frameworks have become common place in web development. In this regard, JPS2.0 is an accurate representation of the complexity found in user interfaces for real-world web applications. The difficulty with modeling control-flow in applications that integrate with JavaScript frameworks is isolating event handlers specific to the application code. Our technique for detecting event handlers involves looking at the bottom of the call-stack. The bottom function represents the initial entry point into the current execution context and is commonly the event handler function we are interested in modeling.

However, this technique breaks down when a framework such as Dojo is added

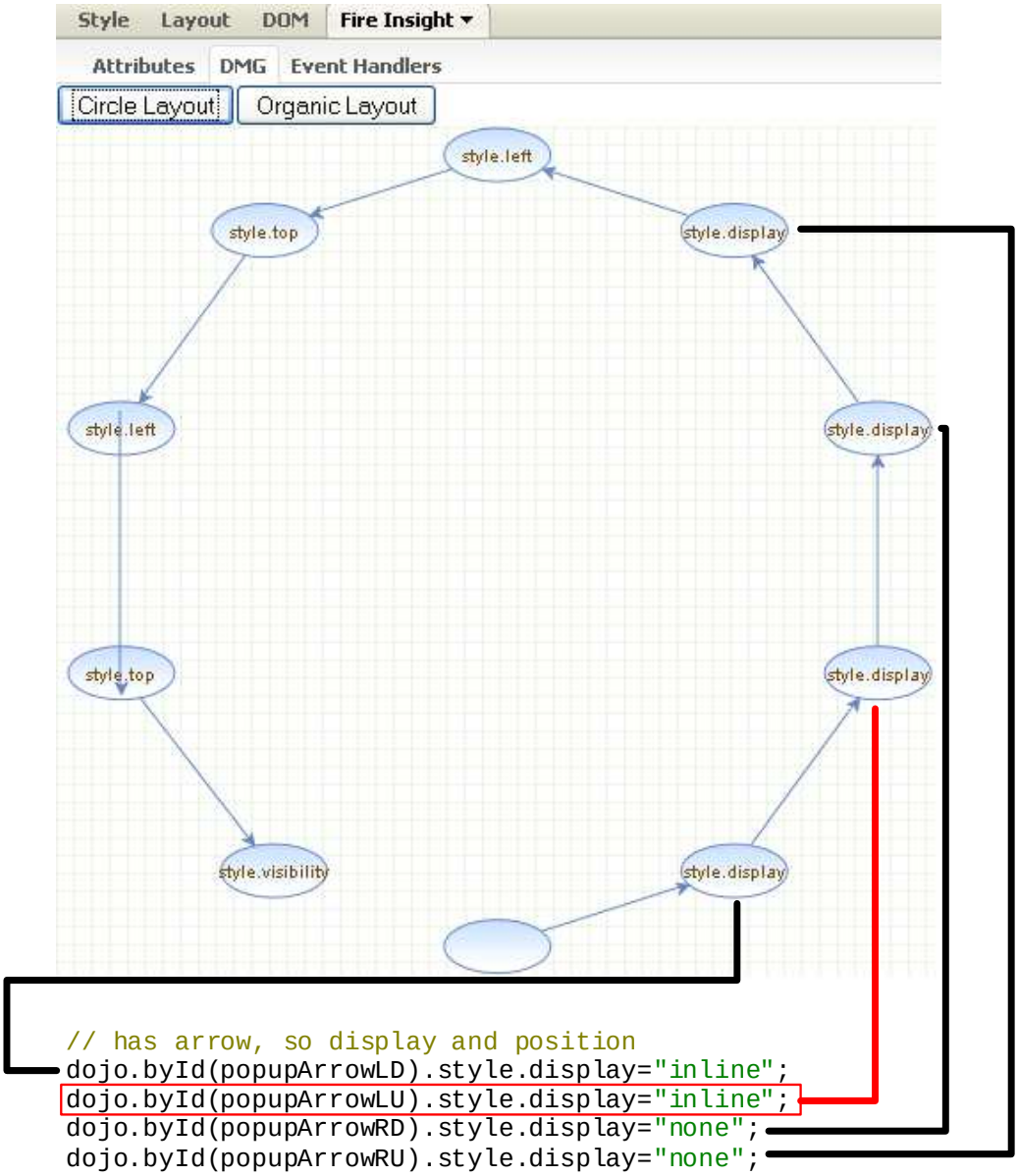


Figure 5.16: Since the type (B) overlay is the only one that contains incorrect logic, we are only interested in inspecting that particular behavior. The above diagram shows the generated DMG for the type (B) overlay. The first four DOM mutator nodes have been mapped back to the exact DOM mutator statements in the source file popup.js. For illustration purposes we have marked the statement that caused the bug.

Application Function Calls

```
createInfoPane()@http://.../petstore/faces/scroller.js:628
initLayout()@http://.../petstore/faces/scroller.js:572
{anonymous}()@http://.../petstore/faces/scroller.js:517
initCatalog()@http://.../petstore/faces/catalog.js:18
{anonymous}([object Event])@http://.../petstore/faces/catalog.jsp?catid=Cats#feline01,1:94
{anonymous}([object Event])@http://.../petstore/faces/static/META-INF/dojo/bpcatalog/dojo.js:3384
{anonymous}([object Event])@http://.../petstore/faces/static/META-INF/dojo/bpcatalog/dojo.js:3277
        .
        .
        .
{anonymous}([object Event])@http://.../petstore/faces/static/META-INF/dojo/bpcatalog/dojo.js:3390
{anonymous}([object Event])@http://.../petstore/faces/static/META-INF/dojo/bpcatalog/dojo.js:3277
```

Dojo Function Calls

Figure 5.17: The above call-stack represents an execution context related to animating the info pane for the Catalog Browser. Some of the entries in the call-stack have been omitted for illustration purposes. The call-stack is divided into two portions: (1) Application function calls, and (2) Dojo function calls. The Dojo function calls all involve dojo.js and are located at the bottom of the stack. This indicates that Dojo is responsible for initiating this execution context.

to the application. This is because Dojo provides its own system for registering application specific event handlers to listen for events through the browser. Dojo provides a layer of abstraction between the application's event handler functions and the browser. Dojo assumes responsibility for wiring the application code together and invoking application logic. Therefore, the bottom of the call-stack is always a Dojo function. This can obscure the event handlers that we are interested in.

Figure 5.17 shows an example call-stack where the bottom level function is a Dojo function. In fact, a significant portion of the call-stack involves Dojo function calls (i.e. references to dojo.js). We can see there is a clear separation between the Dojo function calls, and the application specific function calls. At some point in the execution context, control is handed over from Dojo to the application. However, Dojo is responsible for initiating the execution.

To model application specific event handlers, we would need to ignore the Dojo function calls and isolate the application specific portion. Although, to accomplish this we would need a generic algorithm for determining the division be-

tween application function calls and framework function calls in the call-stack. We currently do not have a generic strategy. As mentioned in Chapter 4, our best workaround is to ignore the Dojo source code during JavaScript instrumentation. We do this by configuring an ignore list of JavaScript source files to explicitly ignore. The source file dojo.js is currently on that list. The problem with this workaround is that it prevents the developer from seeing the entire call-stack. There may be times when it is important to examine Dojo function calls. By ignoring dojo.js during instrumentation we effectively avoid analyzing the Dojo source code.

5.2.2 JavaScript Instrumentation

There are a number of DOM mutator types which are unaccounted for in our current implementation. Unfortunately, our implementation for JavaScript instrumentation required modifying the Rhino JavaScript parser directly. This technique was error prone and time consuming. As a result we were unable to get the Rhino parser to capture all the DOM API methods that we had hoped to cover.

Besides `createElement()` there are many additional API methods that mutate the DOM, such as `appendChild()` and `setAttribute()`. While the function `createElement()` is attached to the global `document` object and therefore easier to replace with our own method, both `appendChild()` and `setAttribute()` are invoked at the individual DOM-element level. Consequently, finding a generic method to capture all calls to `appendChild()` And `setAttribute()` proved to be quite difficult.

Modifying our JavaScript parser to take these additional implementation details into consideration will increase the number of DOM mutators that FireInsight discovers during JavaScript execution. An improved detection rate for DOM mutators will increase the accuracy of FireInsight's DMGs. It would be very interesting to see how many additional DOM mutator nodes are added to the DMGs as a result of capturing more DOM mutator types. If there are too many additional nodes in

the graph, then we also lose some clarity because of the increased complexity in understanding the control-flow represented by the DMG.

Chapter 6

Conclusions and Future Work

In this thesis, we attempted to address the problem of program understanding for JavaScript behavior within web application user interfaces. We argued that program understanding within the domain of front-end development was a challenge of mapping the page elements in the UI to the implementation in the source code. We proposed an approach to improve program understanding by modeling the mapping between the UI behavior in the browser view and the application logic in the code view. The key idea was UI behavior is dictated by mutations to the DOM elements on the page. Therefore, we needed to map the changes in the state on the page to the exact JavaScript statements that caused those changes.

We provided a way to visually inspect the DOM elements in browser view and link those elements back to the DOM mutators in the source code. We accomplished this by recording call-stack information about each DOM mutator during execution of behavior in the browser. The context information was captured in real-time and compiled into an execution history for the page. We provided insights into UI behaviors by modeling the DOM mutators as a control-flow graph. The DMG model provided semantic meaning about an arbitrary execution history by organizing DOM mutators based on the event handlers responsible for invoking them. We used JavaScript instrumentation to capture execution context data. That is,

we injected our own analysis code into existing JavaScript source files as they were delivered to the browser.

We presented a programming tool called FireInsight to demonstrate our approach. To address interoperability we integrated FireInsight with the Mozilla Firefox browser as well as the Firebug web development tool. We evaluated our approach by applying our tool to a reference web application called JPS2.0. Using a series of detailed case-studies we showed how FireInsight could be used to improve developer understanding of the JavaScript behavior in the user interface of JPS2.0.

Thanks to the positive results from our evaluation of FireInsight, we feel there are a number of interesting directions to take our research. First, we need to investigate how to effectively detect the application-level event handlers when the source code integrates with JavaScript frameworks. As we saw for JPS2.0, FireInsight could not adequately detect application-level event handlers when Dojo source code was included in the JavaScript instrumentation. We bypassed the problem by explicitly ignoring Dojo source code during instrumentation. Today's Ajax-driven web applications commonly use JavaScript frameworks or libraries to manage complexity. Therefore, it is crucial that FireInsight be able to handle analyzing code from JavaScript frameworks.

Second, we need to explore ways to make our JavaScript instrumentation procedure configurable, so that a developer can define what JavaScript statements will be captured by our analysis code. Currently we use a custom JavaScript parser that is implemented to look for very specific DOM mutators, such as calls to `createElement()` or direct assignment statements. However, we do not account for all the possible methods for mutating DOM elements. Additional DOM API methods exist that can mutate page elements, such as `appendChild()` and `setAttribute()`. This should serve as motivation to evolve our system for JavaScript instrumentation so that FireInsight can be configured to analyze any arbitrary JavaScript statement.

Bibliography

[1] Adobe. web design software, HTML editor — Adobe Dreamweaver CS4. http://www.adobe.com/products/dreamweaver/index.html.

[2] Aptana. Aptana Studio - The Developer's Choice. http://www.aptana.org/studio/.

[3] T. A. Corbi. Program understanding: challenge for the 1990's. *IBM Syst. J.*, 28(2):294–306, 1989.

[4] Douglas Crockford. JSLint: The JavaScript Code Quality Tool, 2002. http://www.jslint.com/lint.html.

[5] Douglas Crockford. *JavaScript: The Good Parts*. O'Reilly Media, Inc., 2008.

[6] Facebook. Facebook. http://www.facebook.com.

[7] David Flanagan. *JavaScript: The Definitive Guide, Fifth Edition*. O'Reilly Media, Inc., New York, NY, USA, 2006.

[8] Apache Software Foundation. HttpCore - HttpComponents HttpCore Overview. http://hc.apache.org/httpcomponents-core/index.html.

[9] Mozilla Foundation. Firefox web browser — Faster, more secure, & customizable. http://www.mozilla.com/en-US/firefox/firefox.html/.

[10] Mozilla Foundation. Main page - MDC. https://developer.mozilla.org/En.

[11] Mozilla Foundation. Statistics Dashboard :: Add-ons for Firefox. https://addons.mozilla.org/en-US/statistics.

[12] Jesse James Garrett. Ajax: A New Approach to Web Applications. http://www.adaptivepath.com/ideas/essays/archives/000385.php.

[13] Firebug Working Group. Extensions. http://getfirebug.com/extensions/index.html.

[14] Firebug Working Group. Firebug: Web Development Evolved. `http://getfirebug.com/`.

[15] Ross Harmes and Dustin Diaz. *Pro JavaScript Design Patterns*. Apress, 2007.

[16] JGraph. mxGraph - the AJAX diagramming solution. `http://www.jgraph.com/mxgraph.html`.

[17] Emre Kiciman and Ben Livshits. Ajax View - Microsoft Research. `http://research.microsoft.com/en-us/projects/ajaxview/`.

[18] Emre Kiciman and Benjamin Livshits. Ajaxscope: A platform for remotely monitoring the client-side behavior of web 2.0 applications. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 17–30, New York, NY, USA, 2007. ACM.

[19] Haruka Kikuchi, Dachuan Yu, Ajay Chander, Hiroshi Inamura, and Igor Serikov. Javascript instrumentation in practice. In *APLAS '08: Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, pages 326–341, Berlin, Heidelberg, 2008. Springer-Verlag.

[20] J. F. Kurose and K. W. Ross. *Computer Networking - A Top-Down Approach Featuring the Internet*. Addison-Wesley Pearson, Boston, MA, USA, 2005.

[21] Peng Li and Eric Wohlstadter. Script insight: Using models to explore javascript code from the browser view. In *Web Engineering, 9th International Conference, ICWE 2009, San Sebastián, Spain, June 24-26, 2009, Proceedings*, pages 260–274, 2009.

[22] Mozilla. Rhino: JavaScript for Java. `http://www.mozilla.org/rhino/`.

[23] Stephen Oney and Brad Myers. Firecrystal: Understanding interactive behaviors in dynamic web pages. In *VL/HCC 2009: Proceedings of 2009 IEEE Symposium on Visual Languages and Human-Centric Computing*, 2009.

[24] Leslie M. Orchard, Ara Pehlivanian, Scott Koon, and Harley Jones. *Professional JavaScript Frameworks: Prototype,YUI, ExtJS, Dojo and MooTools*. Wrox Press Ltd., 2009.

[25] siliconforks.com. JSCoverage - Code coverage for JavaScript. `http://siliconforks.com/jscoverage/`.

[26] Inc. Sun Microsystems. Java 2 Platform Standard Ed. 5.0 - java.util.concurrent API. http://www.j2ee.me/javase/6/docs/api/java/util/concurrent/package-summary.html.

[27] Inc. Sun Microsystems. Java BluePrints. http://java.sun.com/reference/blueprints/.

[28] Inc. Sun Microsystems. Java Pet Store 2.0 Reference Application. http://java.sun.com/reference/blueprints/.

[29] Inc. Sun Microsystems. Java SE Downloads - Previous Release - J2SE 5.0. http://java.sun.com/javase/downloads/index_jdk5.jsp.

[30] Twitter. Twitter. http://www.twitter.com.

[31] Wikipedia.org. Adobe Dreamweaver - Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Dreamweaver.

[32] Wikipedia.org. Web application. http://en.wikipedia.org/wiki/Web_application.

[33] Nicholas C. Zakas. *Professional JavaScript for Web Developers, Second Edition*. Wrox Press Ltd., Birmingham, UK, UK, 2009.