**ISBN** 978-981-11-3671-9
**Proceedings of 2017 the 7th International Workshop on Computer Science and Engineering (WCSE 2017)**
*Beijing, 25-27 June, 2017, pp. 209-213*
doi: 10.18178/wcse.2017.06.036

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

Peng Li[1+]
[1]Beijing Institute of Technology, 5 South Zhongguancun Street, Haidian District, Beijing, China

**Abstract.** As Web programming standards and browser infrastructures have matured, the implementation of UIs for many Web sites has seen a parallel increase in complexity. In order to deal with this problem, we are researching ways to bridge the gap between the browser view of a UI and its JavaScript implementation. To achieve this, we propose a novel JavaScript reverse-engineering approach and a Firebug extension called FireInSight. This approach helps to relate the semantically meaningful elements in the browser to the lower-level JavaScript syntax, by leveraging context available during the script execution. The approach uses run-time tracing to build a dynamic, context-sensitive, control-flow model that provides feedback to developers as a summary of tracing information. To demonstrate the applicability of the approach we present a study of an existing open-source Web 2.0 application called the Java Pet Store.

**Keywords:** Reverse-Engineering, Software Maintenance, Rich Internet Applications, JavaScript

## 1. Introduction

The user interface (UI) is a key aspect of most Web sites. As Web browser programming standards such as JavaScript and the W3C Document Object Model (DOM) have matured, the implementation of UIs for many sites have seen a parallel increase in complexity. These rich Web applications have the advantage of providing a seamless and interactive experience for end-users. However, these applications also require more development effort to build and maintain than older Web UI. As the Web has become more interactive and complex, we are researching a more interactive, model-based approach for Web application reverse-engineering and debugging.

Unfortunately, reversing engineering a rich interactive Web page and mapping the appearance or behavior of some element in the Web page to the corresponding implementation can be quite difficult. This is because today's Web UI are stateful and reactive. Their appearance and behavior vary over time based on mutations of state made from JavaScript. This problem is exacerbated by the fact that a developer working on the UI might not have written the original code for all parts of the Web site. In that case, they may need to dig through unfamiliar code to try and reverse-engineer the source. This process is especially difficult since code for some systems on the Web is poorly documented. As described by Hassan et al. [4], "Currently, [code] inquiries can only be answered by scanning the source code for answers using tools such as grep, consulting documentation, or asking senior developers."

In order to deal with this problem, we are researching an interactive approach to bridge the gap between the browser view of a UI and the JavaScript piece of the implementation. This is motivated by the fact that the browser view is usually easy to understand and semantically meaningful, unlike the implementation code. We want to help developers use the live UI as an entry-point into the lower-level implementation details.

To achieve this, we propose a novel JavaScript tracing approach. To a first approximation, when a change is made by a script statement to a visual DOM attribute (e.g. color, height, etc...), we record a link

+ Corresponding author. Tel.: + 8613910468787.
*E-mail address*: lipeng360@bit.edu.cn.

between the effected browser element and the code responsible. The intuition is that a developer can now easily navigate through the code by hyperlinking directly from browser elements.

## 2. Implementation

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 have created a programming tool called FireInsight, which is an extension to Firebug, a popular web development tool built on top of the Mozilla Firefox web browser. 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. Our tool is implemented as a JavaScript front-end which is integrated into the Firebug, to execute within a standard Web browser, and a separate HTTP proxy executable. A developer using our tool will install and point their browser to the HTTP proxy which provides instrumentation of existing JavaScript code.

## 3. Case Study: Catalog Browser Info Pane

Our primary objective is to determine whether or not FireInsight can effectively model the UI-JavaScript mapping within JPS 2.0. We evaluate our tool by presenting one particular case study, the Catalog Browser page of JPS 2.0 in Figure 1.

This 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 this scenario, we show how our tool might improve developer understanding. We utilize a narrative approach when presenting this 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.

This Catalog Browser page allows the user to peruse pet listings stored in the JPS2.0 database, which involves a complex user interaction. 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 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.

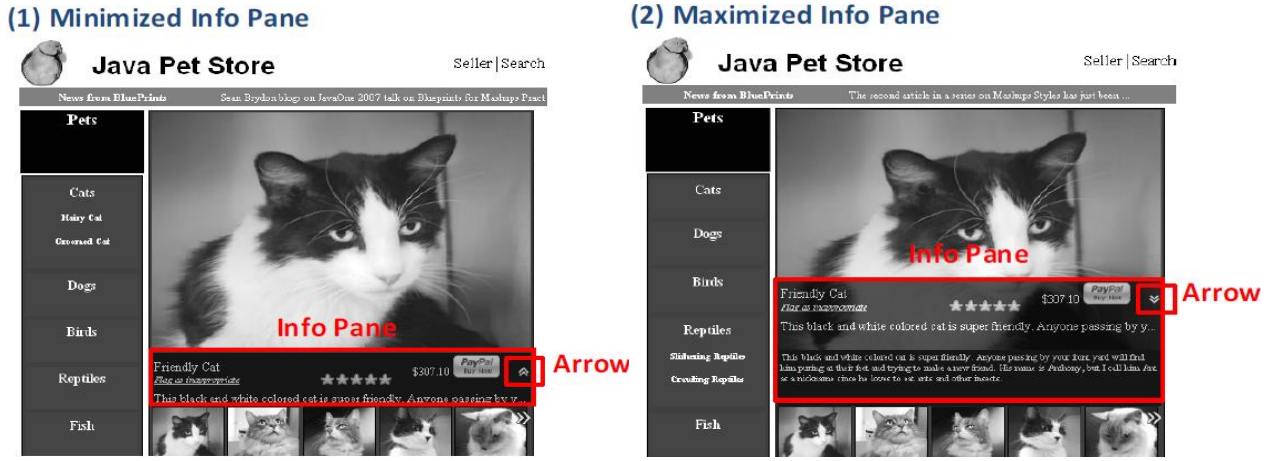


Fig. 1: The above screenshot shows the JPS2.0 Catalog Browser page. In particular it shows the two states of the Info Pane area: (1) minimized, and (2) maximized.

The Info Pane area can be in a minimized state or a maximized state, as shown in Figure 1. 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  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. Manually searching the source code for references to "infopane" in order to understand the logic will be very time consuming.

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 behaviour, we need to research the use of a custom control-flow model, the *DOM mutation graph* (DMG), that developers can use to leverage their understanding of these causal relationships, seen in the browser view, in mapping from the browser view to script source code. 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 2. 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 2 - 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 2 - Step 2). Looking at the Event Handler view, we can see an event handler function called changeInfoPane(). If we examine its DMG we will see that it is responsible for animating the maximization of the Info Pane. The DMG shown in Figure 2 - 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 2 - Step 2.

If we now click the toggle button again, the Info Pane will compress and slide downward. Figure 2 - 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.

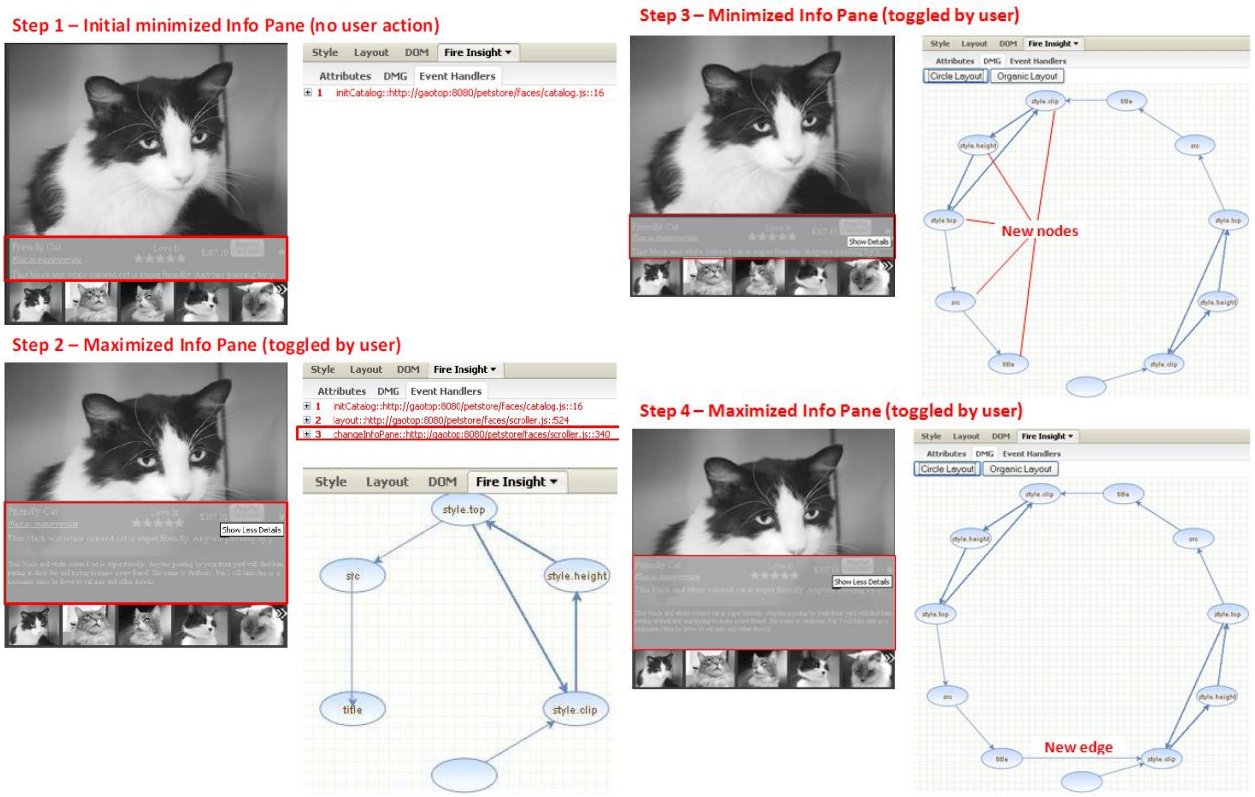


Fig. 2: The above diagram shows how FireInsight's Event Handler and DMG views evolve in real-time as the user interacts with the Info Pane. (1) The initial state of the Info Pane is minimized; the Event Handler view does not show any function related to animating the sliding of the Info Pane and there 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.(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.

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 2 - 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(). 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.

## 4. Related Work

In [11], McMaster et al. present how to use calling context information collected during a GUI program's execution to solve the GUI test suite reduction problem (i.e. finding a minimal satisfactory test set). Their research considers two GUI test cases to be equivalent if they generate the same set of call stacks after execution. This new call-stack coverage criterion can be used to address the challenges for GUI-

intensive applications, which are difficult to be handled by some other criteria such as statement or branch coverage. Similar with their research, we also use calling context to distinguish two artifacts. However, our research is used to resolve the ambiguity of the different UI changes instead of GUI test cases, for example, accordion row expanding and deflation.

In [17], Michail introduced a tool to provide GUI-guided browsing of source. Their objective was to allow developers to find where in the code a feature was implemented, based on how code was related to the GUI. For example, to find "spell checking" code, they could locate the code which executed when the spell-checking menu was selected. Similar to our approach, they use a GUI as an entry-point into the lower-level implementation details. However, they user the GUI to understand its relation to other program features and not the GUI implementation itself.

## 5. Acknowledgements

This project is funded by Frontier and interdisciplinary innovation program of Beijing Institute of Technology.

## 6. References

[1] Yu, J., Benatallah, B., Saint-Paul, R., Casati, F., Daniel, F., and Matera, M.: A framework for rapid integration of presentation components. In: Proc. of the International Conference on the World-Wide Web (2007)

[2] Trigueros, M.L., Preciado, J.C., Sánchez-Figueroa, F.: A Method for Model Based Design of Rich Internet Application Interactive User Interfaces. In: Proc. of the International Conference on Web Engineering, pp:226-241 (2017)

[3] Valderas, P., Pelechano, V., Pastor, O.: Introducing Graphic Designers in a Web Development Process. In: Proc. of International Conference on Advanced Information Systems Engineering,pp: 395-408 (2007)

[4] Hassan, A. and Holt, R.: Architecture recovery of web applications. In: Proc. of the International Conference on Software Engineering (2012)

[5] Dojo JavaScript Toolkit, http://dojotoolkit.org/

[6] jQuery JavaScript Library, http://jquery.com/

[7] Appert, C., Beaudouin-Lafon, M.: SwingStates: adding state machines to Java and the Swing toolkit. Softw., Pract. Exper. 38(11): 1149-1182 (2016)

[8] Shehady, R.K., Siewiorek, D.P.: A Methodology to Automate User Interface Testing Using Variable Finite State Machines. In: Proc. of the International Symposium on Fault-Tolerant Computing, pp:80-88 (1997)

[9] Java Pet Store, Sun Microsystems, http://java.sun.com/developer/releases/petstore/

[10] Rhino JavaScript compiler framework. Mozilla, http://www.mozilla.org/rhino/

[11] McMaster, S., Memon, A.M.: Call Stack Coverage for GUI Test-Suite Reduction. In: Proc of the International Symposium on Software Reliability Engineering, pp:33-44 (2006)

[12] Ricca, F., Tonella, P.: Analysis and Testing of Web Applications. In: Proc. of the International Conference on Software Engineering, pp: 25-34 (2001)

[13] FireBug, http://getfirebug.com/.

[14] Mesbah, A., Bozdag, E., Deursen, A.V.: Crawling AJAX by Inferring User Interface State Changes. In: Proc. of the International Conference on Web Engineering, pp: 122-134 (2008)

[15] Rossi, G., Urbieta, M., Ginzburg, J., Distante, D., Garrido, A.: Refactoring to Rich Internet Applications. A Model-Driven Approach. In: Proc. of the International Conference on Web Engineering, pp: 1-12 (2008)

[16] Meliá, S., Gómez, J., Pérez, S., Díaz, O.: A Model-Driven Development for GWT-Based Rich Internet Applications with OOH4RIA. In: Proc. of the International Conference on Web Engineering, pp: 13-23 (2008)

[17] Michail, A. Browsing and searching source code of applications written using a GUI framework. In: Proc. of the International Conference on Software Engineering (2002)