Delft University of Technology
Software Engineering Research Group
Technical Report Series

Connecting Traces: Understanding Client-Server Interactions in Ajax Applications

Nick Matthijssen, Andy Zaidman, Margaret-Anne Storey, Ian Bull, Arie van Deursen

Report TUD-SERG-2010-008

TUD-SERG-2010-008

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
`http://www.se.ewi.tudelft.nl/techreports/`

For more information about the Software Engineering Research Group:
`http://www.se.ewi.tudelft.nl/`

Note: Paper accepted for publication in the Proceedings of the 18th International Conference on Program Comprehension (ICPC 2010)

Connecting Traces: Understanding Client-Server Interactions in Ajax Applications

Nick Matthijssen*, Andy Zaidman*, Margaret-Anne Storey†, Ian Bull† and Arie van Deursen*
*Software Engineering Research Group, Delft University of Technology, The Netherlands
Email: nick8maal@gmail.com, {a.e.zaidman, arie.vandeursen}@tudelft.nl
†Department of Computer Science, University of Victoria, Victoria, BC, Canada
Email: mstorey@uvic.ca, irbull@gmail.com

***Abstract*—Ajax-enabled web applications are a new breed of highly interactive, highly dynamic web applications. Although Ajax allows developers to create rich web applications, Ajax applications can be difficult to comprehend and thus to maintain. For this reason, we have created FireDetective, a tool that uses dynamic analysis at both the client (browser) *and* server side to facilitate the understanding of Ajax applications. Using an exploratory pre-experimental user study, we see that web developers encounter problems when understanding Ajax applications. We also find preliminary evidence that the FireDetective tool allows web developers to understand Ajax applications more effectively, more efficiently and with more confidence*.**

I. Introduction

Over the last decade web development has evolved from creating static web sites to creating rich and highly interactive web applications. The most important technology in realizing this shift is Ajax (Asynchronous Javascript and XML), an umbrella term for existing techniques such as JavaScript, DOM manipulation and the XMLHttpRequest object. Ajax is popular: since the term was coined in 2005 [2], a vast amount of Ajax enabled web sites have emerged, numerous Ajax frameworks have been created and "an overwhelming number of articles in developer sites and professional magazines have appeared" [3]. A good example of an Ajax application is Gmail, which uses Ajax technologies to update only a part of the page when you open an email conversation, and to suggest email addresses of recent correspondents as you type.

Unfortunately, Ajax also makes developing for the web more complex. Classical web applications are based on a multi-page interface model, in which interactions are based on a page-sequence paradigm [3]. Ajax changes this by allowing asynchronous requests to be made after a page has been loaded and allowing JavaScript code to update parts of the page in the browser, effectively making delta-updates without reloading the complete page.

Before the dawn of Ajax, Hassan and Holt already noted that "Web applications are the legacy software of the future" and "Maintaining such systems is problematic" [4]. We expect that the interactivity and complexity that Ajax adds will certainly not improve this situation.

Software maintenance starts with building up understanding and subsequently making the necessary modifications. This understanding step is known to be very costly, with Corbi reporting that as much as 50% of the time of a maintenance task is spent on understanding [5]. However, papers focusing on program understanding specifically for Ajax applications are scarce (e.g., [6]).

These observations, together with the rapidly growing number of Ajax enabled web applications, motivated us to examine ways to support web developers in maintaining this new breed of web applications. In particular, in this paper we investigate what kind of problems web developers struggle with when understanding an Ajax application and how we can leverage dynamic analysis to better support web developers in understanding Ajax applications.

Our choice for dynamic analysis is instigated by the fact that specific to Ajax applications is the potential difficulty of following the control flow through an application. This stems from the fact that an Ajax application consists of a collection of heterogeneous resources, such as web templates, client side scripts and server side scripts, which are dependent on each other and all of which contribute to the application. Links between these artifacts are often established at runtime. Next, HTML pages can be generated and updated dynamically, and client side scripts can be generated on the fly and executed. Finally, the languages that are used themselves are highly dynamic, such as JavaScript and server side scripting languages such as PHP. Antoniol *et al.* [7] already argued that static analysis alone is insufficient for web applications. We argue that the even higher degree of dynamicity in Ajax applications makes static analysis insufficient for Ajax applications as well.

In order to facilitate a better understanding of Ajax-based web applications, we have built *FireDetective*, a tool that records execution traces on both the client (browser) and server, and subsequently visualizes them in a combined way.

Next, we used our tool in an exploratory study, which matches the early stage of this research. The study consists of two parts, each of which addresses one of our research questions:

RQ1 Which strategies do web developers currently use

*This work is described in more detail in the MSc thesis of Nick Matthijssen [1].

when trying to understand Ajax applications?

RQ2 Can we use dynamic analysis to improve program understanding for Ajax applications?

The rest of this paper is organized as follows. Section II describes the design and implementation of FireDetective. Section III documents the design of our empirical study. Section IV describes and discusses our findings. Threats to validity are covered in Section V. Section VI discusses related work. Finally, Section VII presents our conclusions and identifies future opportunities.

II. Tool design

FireDetective[1] is a tool that records execution traces of the JavaScript code that is executed in the browser *and* of the server side code on the server. The level of detail that is used is the "call" level: the tool records the names of all functions and methods that were called, and in what order they were called, allowing the tool to reconstruct a call tree representation of each trace. From our own experiences as Ajax developers we realized that *relating* these separate traces to each other would be essential for obtaining a good understanding of the control flow through an Ajax application. Consequently, the tool also records information about abstractions that are specific to the Ajax/web-domain, such as (Ajax) requests, DOM events, timeouts, etc. This is a key element of the tool: it enables us to link the aforementioned execution traces in meaningful ways. Moreover, the abstractions can be used as familiar starting points for program understanding. The tool presents the network of traces and abstractions to the user in a set of interactive views.

A. Architecture

The architecture of FireDetective is shown in Figure 1. The tool consists of a Firefox add-on which records JavaScript traces and information about Ajax abstractions, and a server tracer which can be hooked into a Java EE[2] web server. Both of these components forward the data that they record (via sockets) to the visualizer, the third and final component of FireDetective. The visualizer then processes and visualizes the data in real-time. A benefit of this architecture is that it allows users to use Firefox to interact with an Ajax application, as they normally would, and then use the FireDetective visualizer to inspect what is going on "under the hood". The architecture also enables components to run across different machines. Currently, the tool is built for Ajax applications with a Java + JSP back-end, a decision that was influenced by the target application that we chose for our empirical study (see Section III). However, the same techniques can be applied to Ajax applications with other back-ends, such as PHP or Ruby.

[1]FireDetective is open source and can be downloaded from http://swerl.tudelft.nl/bin/view/Main/FireDetective.

[2]Java Platform, Enterprise Edition. See http://java.sun.com/javaee/.

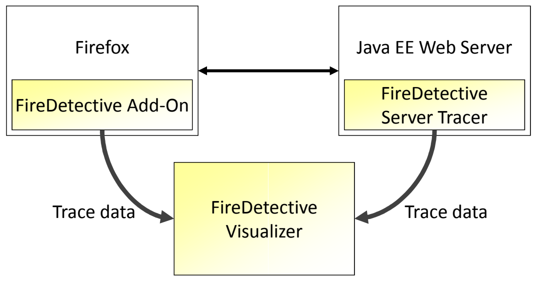


Figure 1. Architecture of FireDetective.

B. Using abstractions to link traces

We use a number of abstractions from the Ajax/web domain to which we link traces or calls within traces. They are listed below.

- **Full page requests** occur when a whole page is loaded. We use a full page request to group all requests and JavaScript traces that take place before the next full page request occurs, into a chronological list.
- **Non-Ajax requests** are contained within a full page request. They are also associated with the server side trace that was recorded for that particular request.
- **Top-level script load invocations** occur when the browser has loaded scripts and executes them. These script loads are linked to the resulting JavaScript trace.
- **DOM events** are events such as "element was clicked" or "page was loaded". They are associated with one or more JavaScript traces that were recorded as a result of event handlers firing for the DOM event in question.
- **Ajax requests**, like other requests, are associated with a single server side trace. They are also linked to the JavaScript call that sent the request and the JavaScript traces that were recorded when handling the response.
- **Timeouts** (in JavaScript) can be set to trigger a timeout handler after a specified time period has elapsed. We link timeouts to the JavaScript traces that were recorded as a result of the timeout handler being invoked, and to the JavaScript calls that started and stopped that particular timeout[3].
- **Web template invocations** are not specific to Ajax, and are used in many web applications. In our case, we are working with JSP templates. Because these templates are compiled prior to use, they do not end up in the trace in their original form. Therefore, we reconstruct JSP invocations from the original trace and link them to the points in the traces where they took place.

Some links between traces/calls and abstractions represent a causal relationship, e.g. some JavaScript call *causes* an Ajax request, which then *causes* a server side and – when the response is received – JavaScript trace to be created. By following these links in one direction, tool users are able to answer "what?" and "how?" questions about the program,

[3]The last two types of links were only implemented after conducting the empirical study.

e.g. "how was this DOM event handled?". Moreover, links can also be followed in the reverse direction, enabling tool users to answer "why?" questions, e.g. "why did this Ajax request occur?".

The abstractions were identified through our own experiences as Ajax developers. In Section IV we offer possible additions to this list. We used different mechanisms for recording and reconstructing these abstractions, and linking them to the relevant traces. These mechanisms are briefly described in Subsection II-E.

C. Interactive visualization

The visualizer displays the collection of traces and abstractions to the user. Its interface is shown in Figure 2. The visualization's design is loosely based on guidelines outlined by Shneiderman [8]: information visualization tools should allow for creating overviews, zooming, filtering, and providing details on demand. This design correlates with a top-down comprehension strategy [9].

Three main views are used, each of which shows a different level of detail. The first view is a high-level view, which shows a tree representation of the aforementioned abstractions (except template invocations). Expandable tree nodes may reveal more detail, e.g. expanding an Ajax request node shows its relation to particular traces and calls, i.e. the life cycle of the request. The second view is a trace view which displays one execution trace at a time, as a call tree. Each tree node represents a single call, with expandable subcalls. The third view is a standard source code view.

The three views are linked: selecting a high-level entity in the first view shows the related trace in the trace view, and selecting a call in the trace view shows the related code. There is also one side view, which contains a tree representation of the resources (e.g. code files) of the Ajax application. Clicking a resource shows the file in the code view. The view can be filtered to only show the files that were used for the current page, which greatly reduces the number of files that are shown, and allows a tool user to quickly see which resources are involved on the current page. The user can also select a block of code (e.g. a JavaScript function) to highlight and cycle through invocations of that block of code in the high-level view and trace view.

A disadvantage of execution traces is that they can quickly grow to massive proportions. In order to reduce the size of traces, we use two simple, well-known trace reduction mechanisms [10]. The first one is to filter out all library calls and only keep calls that are specific to the Ajax application that is being analyzed. Both client side libraries (such as Dojo[4] and server side libraries (such as Java EE server internals) are filtered out. The second mechanism concerns stopping and starting recording. This allows the user to time slice the Ajax application, and, for example, to find out how a particular interaction with the Ajax application is handled.

[4]See http://dojotoolkit.org/.

D. Barriers to comprehension

One caveat regarding JavaScript tracing is that the language allows a developer to define anonymous functions, a mechanism which is commonly used by web developers. Because many trace visualizations (including ours) display the names of invoked functions, this becomes a problem: e.g., a call tree showing "anonymous" functions calling each other is not particularly helpful.

In practice, it turns out that a function is often assigned to exactly one variable, e.g.: `var f = function(...) { ... }`. Therefore, whenever this is the case, we use the name of the variable to identify the function. We parse all JavaScript files and for every anonymous function definition that we encounter, we try to find a variable or instance variable that precedes it. Note that this approach is not always correct: in the example, f could be reassigned another function. However, the approach seems to work well in practice: for example, the popular Firefox FireBug add-on currently[5] uses a similar technique (albeit simpler, based on regular expressions) to "name" anonymous functions.

Another potential issue is the "lazy loading" of JavaScript files, a technique that is used in the Dojo library, for example. "Lazy loading" refers to retrieving a script file by means of an Ajax request, and subsequently "eval"-ing it, reducing the initial page load time. However, because of the "eval" call, the link between original filename and code is lost. This can lead to the undesirable situation of having a fragment of code and not knowing where it came from, except that it was dynamically generated at some point.

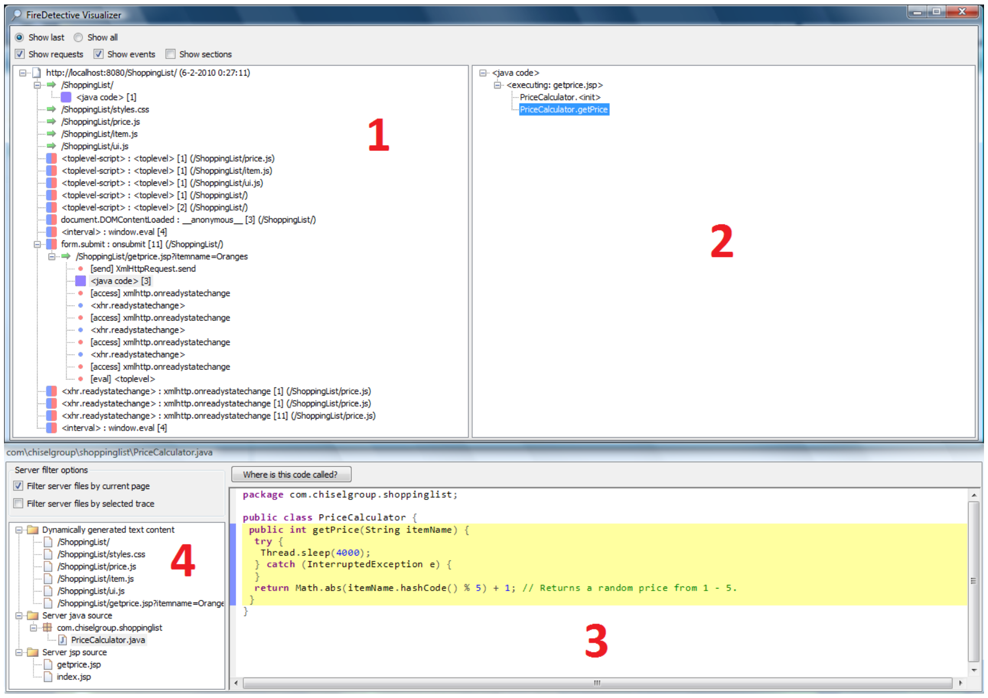
The tool solves this problem by computing a hash code for the response text of every Ajax request, and every "eval"-ed string. When the tool shows a fragment of "eval"-ed code and finds a matching Ajax response text hash, the tool can reconstruct the filename of the "eval"-ed code.

E. Implementation details

JavaScript function calls and Java calls are recorded using Firefox' debugger interface and the Java VM tool interface, respectively. This has the advantage that no code needs to be instrumented, and that the approach also works for JavaScript code that is generated dynamically and "eval"-ed on the fly.

The connection between browser and server is made by appending a custom header `X-REQUEST-ID` containing an id, to every outgoing HTTP request in Firefox. Upon receiving the request on the server side, the id can be detected by the server tracer. DOM events are registered in Firefox by adding event listeners for all possible DOM events, for the `window` and `document` objects. Ajax requests and JavaScript timeouts (and intervals) are registered by wrapping all related properties and functions (e.g., `XMLHttpRequest.responseXML`,

[5]FireBug 1.5.0, see http://getfirebug.com/.



```
http://localhost:8080/ShoppingList/ (6-2-2010 0:27:11)
  /ShoppingList/
    <java code> [1]
  /ShoppingList/styles.css
  /ShoppingList/price.js
  /ShoppingList/item.js
  /ShoppingList/ui.js
  <toplevel-script> : <toplevel> [1] (/ShoppingList/price.js)
  <toplevel-script> : <toplevel> [1] (/ShoppingList/item.js)
  <toplevel-script> : <toplevel> [1] (/ShoppingList/ui.js)
  <toplevel-script> : <toplevel> [1] (/ShoppingList/)
  <toplevel-script> : <toplevel> [2] (/ShoppingList/)
  document.DOMContentLoaded : __anonymous__ [3] (/ShoppingList/)
  <interval> : window.eval [4]
  form.submit : onsubmit [11] (/ShoppingList/)
    /ShoppingList/getprice.jsp?itemname=Oranges
      [send] XmlHttpRequest.send
      <java code> [3]
      [access] xmlhttp.onreadystatechange
      <xhr.readystatechange>
      [access] xmlhttp.onreadystatechange
      <xhr.readystatechange>
      [access] xmlhttp.onreadystatechange
      <xhr.readystatechange>
      [access] xmlhttp.onreadystatechange
      [eval] <toplevel>
  <xhr.readystatechange> : xmlhttp.onreadystatechange [1] (/ShoppingList/price.js)
  <xhr.readystatechange> : xmlhttp.onreadystatechange [1] (/ShoppingList/price.js)
  <xhr.readystatechange> : xmlhttp.onreadystatechange [11] (/ShoppingList/price.js)
  <interval> : window.eval [4]
```

```
<java code>
  <executing: getprice.jsp>
    PriceCalculator.<init>
    PriceCalculator.getPrice
```

com\chiselgroup\shoppinglist\PriceCalculator.java

```
Dynamically generated text content
  /ShoppingList/
  /ShoppingList/styles.css
  /ShoppingList/price.js
  /ShoppingList/item.js
  /ShoppingList/ui.js
  /ShoppingList/getprice.jsp?itemname=Orange
Server java source
  com.chiselgroup.shoppinglist
    PriceCalculator.java
Server jsp source
  getprice.jsp
  index.jsp
```

```java
package com.chiselgroup.shoppinglist;

public class PriceCalculator {
 public int getPrice(String itemName) {
  try {
   Thread.sleep(4000);
  } catch (InterruptedException e) {
  }
  return Math.abs(itemName.hashCode() % 5) + 1; // Returns a random price from 1 - 5.
 }
}
```

Figure 2. The visualizer, showing an analysis of a small sample application. 1. High-level view. An Ajax request is expanded; related traces/calls are shown. 2. Trace view. 3 Code view. 4. Resource list, showing only the files that were used on the current page.

`window.setTimeout`) and callbacks. JSP invocations are reconstructed by recognizing certain calls that occur within the JSP engine, which works well for our target application, although it fails to scale up to bigger applications with multiple JSP files with the same name, but in different directories. One possible solution would be to instrument JSP files prior to analysis, which has the additional benefit of not depending on implementation details of the JSP engine.

III. Study design

We used an exploratory pre-experimental user study to address our research questions: which strategies do web developers currently use, and, can dynamic analysis improve program understanding for Ajax applications? The type of experiment is called pre-experimental to indicate that it does not meet the scientific standards of experimental design [11], yet it allows to report on facts of real user-behavior, even those observed in under-controlled, limited-sample experiences.

In the study, we observed 8 participants working on a number of program understanding tasks. Two were full-time software developers; the other six were computer science or software engineering students, of which five had a part-time software development job. Each participant's session consisted of two distinct parts:

- **Part A: Observing current understanding strategies**. Participants used a standard set of web development tools: Eclipse and Firefox with the popular FireBug add-on. *The purpose of this part is to provide insight into which strategies web developers use when trying to understand Ajax applications, and whether these strategies are sufficient.*
- **Part B: Support through dynamic analysis**. Participants used Eclipse and Firefox with FireDetective. *The purpose of this part is to provide insight into whether dynamic analysis techniques as provided through FireDetective can improve understanding, and if so, how.*

Our approach is exploratory and the focus lies on observing participants as they work on tasks. We asked participants to think aloud during the study, and because the study was conducted in a lab setting we were able to make audio and screen recordings for later analysis. After each part, participants were subjected to a short interview. Additionally, some quantitative data was collected during the study, mainly through two short questionnaires. In the following sections, these aspects are described in more detail.

A. Design of Part A: Observing current understanding strategies

Part A started with a background interview and questionnaire, to gauge the development experience of the participant. This was followed up by a 10-minute introduction to the tools used in this part of the study: Eclipse and FireBug. Since participants were likely to have experience with these tools (this was indeed the case, see section IV), the introduction served mostly to refresh the participants' memory.

After the introduction, participants worked on a set of program understanding tasks for 35 minutes. We emphasized that they could use any feature they wished, to minimize bias towards using the features that we had shown them. Participants were informed that they could move on to the next task if they failed to make progress on their current task, and that they could ask questions at any time (questions about the target application itself were not answered, for obvious reasons). Also, if the experiment leader noticed that a participant was struggling with a particular tool feature, the participant would be given a short explanation of the feature. Since our goal was to find out as much as we can about the strategies that participants use, we did not want them to get stuck for too long. A short interview asking participants about encountered problems concluded part A.

B. Design of part B: Support through dynamic analysis

After a short break, participants continued with part B, during which they used Eclipse and FireDetective. Ideally, we would have included FireBug as well, but unfortunately, FireDetective and FireBug are currently incompatible.

As in part A of the study, the focus lies on observing participants as they work on tasks. However, part B also contains a quantitative component, for which a pretest-posttest design was used [12]. The pretest measured participants' expectations prior to using FireDetective, while the posttest measured participants' experience after using the tool. In particular, we evaluated four attributes:

- **Better understanding.** Does the tool allow web developers to understand Ajax applications more effectively?
- **Quicker understanding.** Does the tool allow to understand Ajax applications more efficiently?
- **More confident about understanding.** Does the tool make web developers more confident about their understanding of an Ajax application?
- **Minimal value.** This attribute is inversely related to the above attributes. Does the tool provide value?

For the pretest, participants were given a short abstract description of a tool like FireDetective. To avoid influencing participants' expectations by exposing them to part A of the study, the pretest was conducted during the beginning of part A (after the background questionnaire). The posttest took place after working with FireDetective. In both the pretest and posttest, each of the four attributes was tested via a multiple choice question for which we used a 5-point Likert scale, ranging from strongly disagree to strongly agree.

After a 10-minute introduction to FireDetective, participants worked on a different set of program understanding tasks, for 25 minutes. This choice was made to keep the complete duration of the study under two hours. Since our intent is not to compare the effectiveness of FireDetective to FireBug, this difference in timing is not a concern in our study design. We decided to allocate more time to Part A as understanding how developers use existing tools was more important to us at this stage of our research. The target application was the same as in part A.

Working on the tasks was followed by the posttest questionnaire. We also asked participants to rate their top 3 features. Finally, another short interview was conducted, asking about encountered problems, least and best liked parts of the tool and suggestions for improvement.

C. Target application

To gain real world insights, we required a target application that was representative of a real world Ajax application and written using languages and technologies that our participants were familiar with. The Java Pet Store satisfied these requirements. It is a reference application, "designed to illustrate how the Java Enterprise Edition 5 Platform can be used to develop an AJAX-enabled Web 2.0 application"[6]. The application consists of 12KLoc, which are written in a variety of languages, such as HTML, CSS and JavaScript on the client side, and Java and JSP on the server side. All of these files were made available in an Eclipse workspace.

The Java BluePrints library is used extensively in the Pet Store, and we found that not including its client side code limited us in the task design. Moreover, this code would show up in FireBug and FireDetective anyway. Hence, we made sure that all client side code that was potentially visible in FireBug and FireDetective could also be found in Eclipse. This amounted to +6KLoc for BluePrints and +97KLoc for Dojo, respectively.

D. Task design

The study required the design of two task sets, one for each part of the study. We constructed the tasks ourselves, by drawing from our own experience with the Pet Store. Each task set consisted of 4 tasks, divided into 2 or 3 subtasks each, adding up to a total of 10 subtasks per task set[7].

For the generalizability of the study it is important to make sure that the tasks are realistic and that they accurately represent a significant part of the program understanding task domain. Therefore, we used open-ended questions rather than multiple choice questions. Moreover, we designed tasks using Pacione's taxonomy of 9 principal activities [13], and strove for coverage of the first 6 principles he suggests: A1. Investigating the functionality of (a part of) the system; A2. Adding to or changing the system's functionality; A3. Investigating the internal structure of an artifact; A4. Investigating dependencies between artifacts; A5. Investigating runtime interactions in the system; A6. Investigating how much an artifact is used. We did not cover the last three principles, to limit the number of tasks and reduce the risk of our participants becoming fatigued during the study.

Since we were keen to observe how FireDetective would be used on unfamiliar code, we strove to choose tasks for

[6]See http://java.sun.com/developer/releases/petstore/, retrieved on December 14th, 2009.

[7]The task descriptions can be found in [1].

the second set that would involve code not inspected in part A of the study.

E. Pilot sessions

Three pilot sessions were conducted to fine tune the study.

The first pilot session did not use think aloud, and it turned out to be hard to reconstruct the participant's thinking steps. As a result, we switched to think aloud with audio and screen recordings. Also, the questionnaires were reduced in size, with more emphasis on participant interviews. To keep the total length of the study under 2 hours, the duration of the second part (during which participants use FireDetective) was reduced from 35 to 25 minutes.

During the second pilot we found that the tasks were too difficult, so they were altered to make them slightly easier. To reduce pressure on participants, we decided to give out tasks one at a time. Also, at the beginning of the study we made it clear that if participants were unsure what to do next, they could indicate this and move on to the next task. Finally, FireDetective's user interface was improved and simplified.

The third pilot session ran without major problems and only a few minor adjustments were made afterwards. In particular, we altered the introduction to Eclipse to exclude explanations of Eclipse features (such as "Call hierarchy") as such explanations may bias participants towards using these features. Also, some of the task descriptions were adjusted to make them clearer.

F. Participant profile

All participants in the user study were required to have web development experience. Since the term "web development experience" can be interpreted quite broadly, we specifically asked for basic Java and JavaScript experience. We assumed that when people had experience with these two languages, especially the latter, they would also have experience with web development. This turned out to be the case for 8 out of 9 participants that we recruited. One participant indicated to have 0 years of web development experience and this was reflected in the results: the participant was only able to complete the most basic tasks. Because the participant was clearly not representative of the target population we excluded this data. As such, the total number of participants is 8.

Our 8 participants represent our target population quite well. 5 had a professional web development job: 1 full-time and 4 part-time. 2 others had a professional software development job: 1 full-time and 1 part-time. Both of these participants indicated that they worked on web development projects for at least a part of their jobs. Except for the 2 full-time developers, the 6 other participants were either computer science or software engineering students: 4 undergraduate and 2 PhD students. Participants' median number of years of web development experience was 2 years (min. 1 year, max. 5 years); it can be argued that this is a

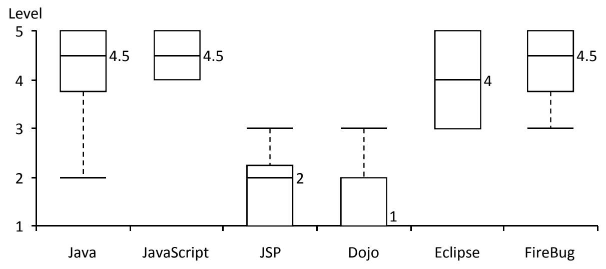


Figure 3. Box plots of participants' experience with relevant technologies and tools. The values on the 5-point scale (vertical axis) correspond to 1 = "Never used it", 2 = "Used it for a couple of hours or less", 3 = "Used it for one or two projects", 4 = "I use it regularly" and 5 = "I've been using it regularly for over two years now".

low number. However, technologies like Ajax have not been around for that long: at the time of writing, the term Ajax has been coined less than 5 years ago [2]. Moreover, the median number of years of software development experience was 5.5 years (min. 2 years, max. 10 years), which shows that participants *did* have general software development skills.

Participants' rated their experience with particular technologies that were relevant to the study. They did so on a custom 5-point scale; the results are shown in Figure 3. We can clearly see that participants have a good understanding of Java, JavaScript, Eclipse and FireBug, yet, we can also see that they are not familiar with JSP and the Dojo library. The impact of this on the generalizability of the study is discussed in Section V, which covers threats to validity. Participants did not have a prior understanding of the Pet Store or BluePrints library, which we could see from observing participants working on the tasks.

IV. Findings and discussion

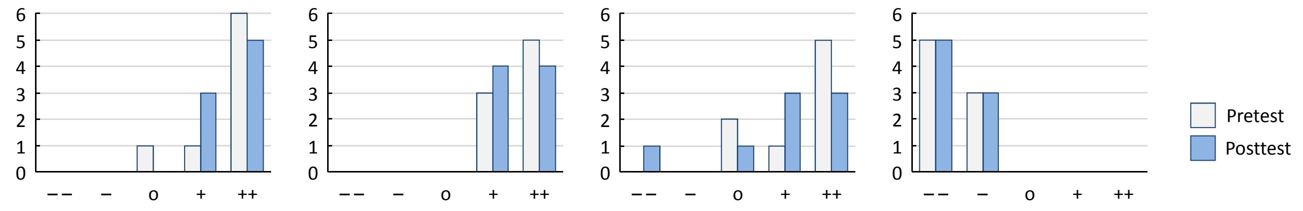
Although our focus in this study was not on the number of completed tasks, but rather the strategies used for solving them, it is interesting to note that the median number of subtasks worked on for part A is 6 (min. 4, max. 8), for part B this is 7 (min. 5, max. 9). Roughly two thirds of these attempts led to the correct answer, in both parts of the study. In the following, we report our observations.

A. Part A: Observing current understanding strategies

Central to the first part of the study is our first research question: "which strategies do web developers currently use when trying to understand Ajax applications?" While participants were working with Eclipse and FireBug, we were able to make a number of observations.

First of all, participants relied almost solely on bottom-up comprehension strategies, i.e. starting at the lowest level – e.g. code fragments – and trying to piece the fragments that they found together. Participants mainly focused on exploring control flow relationships [14], i.e. finding definitions and/or occurrences of functions, methods and classes.

In order to explore these control flow relationships, all participants made heavy use of text search. While Eclipse provides functionality for exploring control flow, e.g., the



(a) Better understanding (b) Quicker understanding (c) More confident (d) Minimal value (e) Legend

Figure 4. Distributions of participants' expectations before (pretest, light gray) and experiences after (posttest, blue) using FireDetective. Horizontal axes: 5-point Likert scale, ranging from strongly disagree ("– –") to strongly agree ("++"). Vertical axes: number of participants.

"Open Declaration" and "Call Hierarchy" functions, these functions were only occasionally used by participants (far less than text search). A possible reason for this might be that these functions (currently) do not always work as expected for web applications: for instance, opening the "Call hierarchy" of a Java method does not show calls made from a JSP file, and "Open Declaration" does not always work well with JavaScript's anonymous functions.

Another use of text search, specific to web applications, was mapping an id of an element (usually found through the FireBug element inspector) to where the id was used in the code. We also noticed more *ad hoc* uses of text search, such as searching for (part of) an URL or searching for some text of the web page, used both successfully and unsuccessfully by participants to get an idea of where a particular element or URL was generated on the server.

Text search leads to a number of problems. Important results are sometimes missed because of cluttering of the search results window or choosing the wrong search scope. The biggest problem is that text search only allows the user to explore one control flow link at a time, making it easy to lose track. During a task when participants were required to follow a small but branching call tree, participants quickly lost track of which branches they had already explored, causing them to make mistakes: only two participants were able to provide a correct answer.

Discussion. From this we conclude that the strategies that web developers currently use can be improved. Participants rely mostly on looking at code and text search, which can be better supported by tools. Since following control flow constitutes a fairly big chunk of participants' actions, supporting this process seems useful. Considering the incompleteness of static analysis and the highly dynamic nature of web applications, we argue that dynamic analysis support would be beneficial in tool support.

B. Part B: Support through dynamic analysis

Central to this part of the study is our second research question: "Can dynamic analysis improve program understanding for Ajax applications?" If this is the case, we would also like to learn more about *how* this works, and what we can do to further improve understanding. We obtained insights into these questions via four different routes: the pretest-posttest, the questionnaire about feature usefulness, observing participants using the tool and the final interview.

Pretest–posttest. The results of the pretest and posttest are shown in Figure 4. From a first look at the results we can see that the pretest and posttest results are fairly similar: it is interesting to know that participants did not completely switch their opinions before and after using the tool. The pretest results are quite positive, which confirms the need for tool support that we found in part A. The posttest results are quite positive as well, and show that participants were quite pleased with FireDetective.

In particular, participants indicate that the tool can help them to understand web applications more effectively (a) and more efficiently (b). Participants also seem convinced that the tool helps them to be more confident about their understanding of the web application they are investigating (c), although their answers are somewhat more distributed compared to the other questions. One participant answered "strongly disagree" during the posttest, as can be seen from the figure. Interestingly enough, when asked why this was, the participant answered that the tool made some tasks almost too easy: "It seemed like I caught [the answer] a lot quicker than I was expecting, so that questioned how much I really trusted the results that I came up with." Finally, participants acknowledge that the tool adds value (d).

Discussion. While these are preliminary findings, we think they are very encouraging. They show that FireDetective, which leverages dynamic analysis techniques, is indeed capable of improving program understanding for Ajax applications.

Features. We asked participants' opinion on 6 features of FireDetective that we wanted to investigate in more detail: the high-level view (F1), the files view which is filtered and only shows the files that were used on the current page (F2), the ability to jump between client and server traces (F3), the ability to follow the life cycle of an Ajax request (F4), time slicing the analysis by starting and stopping tracing (F5), and the fact that the analysis is real-time (F6). By looking at the screen recordings we were able to reconstruct feature use; feature usefulness was measured by asking participants to indicate their top 3 features in the final questionnaire.

All participants used the first three features (F1, F2, F3). This is not too surprising, since these features are central

	1	2	3	4	5	6	7	8
F1: High-level overview			1	1	2		1	3
F2: Filtered files view	3	2	2	2		2	2	
F3: Jumping between client-server		3		3	3	1		1
F4: Following Ajax requests' life cycles			3					
F5: Time slicing	1	1			1			2
F6: Real-time analysis	2					3	3	

Figure 5. Participants' top 3 features. Each column represents one participant. The 1's, 2's and 3's indicate the participant's best liked, second best and third best liked features respectively.

to the tool. 6 out of 8 participants used the time slice feature (F5) and 4 participants briefly explored the life cycle feature (F4). (use of F6 is implicit). Participants' subjective preferences towards features are shown in Figure 5. We can see that there is no clear winning feature. However, we *can* observe some trends, which may give us some insight into how FireDetective helped improve program understanding.

The high-level overview (F1) and time slicing (of the high-level view) (F5) seem to be popular with three #1 votes each, as well as jumping between client and server (F3) – two #1 votes. A possible explanation for this popularity could be that these three features all play a role in enabling a more top-down understanding process, which, as we could see from part A of the study, participants did not previously use. Rather than starting with low-level code, participants can now look at abstractions such as Ajax requests and DOM events and use them as starting points to explore the code. The filtered files view (F2) has the largest number of votes in general, and may play a similar role. From part A of the study, we saw that participants often did not know all of the files that were relevant to a certain page of the Pet Store: the filtered files view provides an initial overview of these relevant files, such that participants have a better starting point for investigation.

Discussion. It is hard to determine exactly which elements of FireDetective are the main contributors to its usefulness. Some features are untestable via a questionnaire, such as "code view" and "naming of anonymous functions" (automatic): these features are used all the time, but because of that it can be hard for participants to determine whether these features were actually useful.

Observations and interview. Besides the expected learning curve and usability issues (see [1]), participants encountered a number of issues when working with FireDetective.

One interesting issue that several participants encountered had to do with Java servlet *filters*, server side classes defined by the web application that process requests. Because the tool records calls to all methods, it also shows calls made to filter classes. However, it cannot show *why* these calls occur, since the internal server logic that calls the filters is hidden from view, and even if the tool were to show these internal calls, it would produce a distorted picture, since the real cause of the filter being called is a binding specified in an XML file. During the study, several participants encountered this problem. They were wondering why the EntryFilter class of the PetStore was invoked, but the tool was unable to give them this information.

Another problem occurred during a task in which participants had to examine a bug, caused by a click handler that contained a syntax error. Participants, still unaware of the cause of the bug, would trigger the click event and search for it in the high-level view of the tool. However, the click event handler did not show up because it failed to compile. Since the tool did not capture information about JavaScript compilations, it was unable to show the reason for the event handler not being called. When participants noticed the syntax error (mostly by hovering over the element, causing Firefox to show the associated script in the status bar), they wanted to find where the event handler was set. Most participants said they would have liked to use FireBug at this point, to use the element inspector to find the id of the element, and look through the code for that id. They essentially wanted to link DOM (element) mutations to code, something which FireDetective cannot currently do since it does not record information about the *DOM mutation abstraction*.

Finally, participants were slightly confused by the way the tool presents full-page requests. The high-level view was filtered to show only the last full-page request. However, participants did not always notice this, causing them to think that they were dealing with an Ajax request, while it was actually a full-page request. This confused them because they were looking for an Ajax request that did not exist.

When asked about potential tool improvements, participants often indicated integration with FireBug, providing evidence for the fact that FireBug and FireDetective are complementary. Participants also asked for mechanisms to reduce the amount of visible information: they were sometimes overwhelmed by the information shown. Since we used only basic trace visualization and reduction techniques, this was to be expected. Participants asked for particular static analysis techniques, such as full text search, possibly because they are attached to their old way of working, but probably because static and dynamic analysis are complementary techniques.

Discussion. From the observations that we made we can extract three ways in which this work can be continued:

- **Other types of abstractions**. The absence of certain abstractions in the tool hampered the understanding process. Our first suggestion is to record information about various types of XML bindings and link them to traces. Candidates include the aforementioned *filters*, servlet mappings and *taglibs* (custom JSP tags, which are linked to their implementation via XML). XML bindings in general represent connecting information,

and hence, they can be very helpful for improving understanding. Other abstractions that we found evidence for being useful are the JavaScript script parsing process and the errors that occur during it and DOM mutations.
- **Different kinds of visualizations.** FireDetective's visualizations are straightforward representations of the recorded abstractions and traces. Only simple trace reduction techniques were used, which – expectedly – caused participants to be overloaded with information on various occasions. We should investigate how to visualize the connected network of abstractions, traces and code in better ways.
- **Integration with existing tools.** From the study it became obvious that FireDetective and FireBug are complementary tools. It could be interesting to investigate how these tools exactly complement each other and how they can be integrated more tightly.

V. Threats to validity

A. Internal validity

Participants might have been inclined to rate the tool more positively than they actually value it, because they might have felt this was the more desirable answer. We mitigated this concern by indicating to participants that only honest answers were valuable.

Next, the introduction sessions might have biased participants towards using the features that we showed them. We tried to neutralize this threat in the following way. During the introduction session for part A we only showed participants basic information on where they could find the different parts (i.e. server side code, client side code) within the Eclipse project, and the basic FireBug views. Explanations of other features were not included and participants were told they could use any feature they liked. For part B, we made sure to explain *all* features of FireDetective.

The tasks might have been too easy or too difficult. However, through pilot sessions we adjusted the task difficulty level accordingly. Also, participants might have felt time pressure, causing them to behave differently. We minimized this problem by telling them that the number of tasks completed was not important and by handing out tasks one at a time, without revealing how many there were to come.

B. External validity

A concern regarding the generalizability of the results is that most participants were students. However, as shown in Section III-F a lot of these participants had a relevant part-time job. Participants were not familiar with two of the technologies used in the study, JSP and Dojo. We admit that the learning curve involved has likely impacted the results. Yet, we also think that this impact is limited because both JSP and Dojo are technologies that are very similar to rivaling technologies. Moreover, participants were given a brief introduction to JSP, and were allowed to ask questions about the technologies involved at any time.

The Java Pet Store, our target application, is a showcase application. This might cause one to question whether this application is representative of a real-world Ajax application. However, the application represents the state-of-the-practice and manual inspection of the application shows that it uses Ajax on most of its pages and is clearly more than just a "toy example". Moreover, the application has been used in previous program understanding research efforts, e.g. [15].

Finally, the tasks might not have been representative of real-world tasks. Because of the limited time frame tasks are likely to be shorter than real-world tasks, and they might not have covered all program understanding aspects. We tried to mitigate this threat by using Pacione's framework of principal comprehension activities [13] to make sure that the tasks are realistic and cover a significant portion of the program comprehension spectrum.

VI. Related work

Early web application reverse engineering efforts were mainly focused on architecture reconstruction, e.g. [4], [7], [16]–[18]. Static analysis alone does not suffice because of the dynamic nature of web applications [7], [18], so in most cases the static analysis is complemented by dynamic analysis. However, many client side aspects that are common in Ajax applications are not taken into account.

De Pauw *et al.* [19] present the *Web Services Navigator*, a tool that offers insight into message and transaction flows in systems of multiple web services. The tool combines multiple web service event logs to reconstruct meaningful abstractions in the web service domain and has some similarities with FireDetective, albeit applied to a different domain.

Recent efforts have focused on understanding only client side aspects of Ajax applications. Li and Wohlstadter [15] present a tool named *Script InSight*, which uses dynamic analysis to record DOM mutations and relate them to the JavaScript functions that caused them. This allows a web developer to map a DOM element on the page to locations in the code where the element was modified.

Oney and Myers [20] present *FireCrystal*, which enables a user to view a timeline of DOM events and DOM modifications, and view code coverage per DOM event.

Our approach differs from these last two approaches in a number of ways. First, our approach visualizes execution traces. Second, it combines client and server side information to show a complete picture of an Ajax application. Third, it uses a different and larger set of abstractions from the Ajax/web domain to link traces together (in contrast to only DOM mutations and DOM events).

Finally, there is one commercial tool of interest: *DynaTrace Ajax*[8]. While conceptually quite similar to FireDetec-

[8]See http://ajax.dynatrace.com/. DynaTrace Ajax Edition was released in September 2009, after we built FireDetective.

tive, DynaTrace Ajax focuses on performance analysis.

VII. Conclusions and future work

In this paper we have introduced FireDetective, a dynamic analysis tool for analyzing Ajax applications. FireDetective records execution traces on both the browser and server, captures information about Ajax/web abstractions, and presents this information in a linked way.

We conducted an exploratory user study to provide insight into our two research questions:

RQ1 *Which strategies do web developers currently use when trying to understand Ajax applications?* Participants mainly use a bottom-up approach, and heavily rely on text search. This strategy is *ad hoc* and problematic for understanding Ajax applications; tool support should be improved.

RQ2 *Can we use dynamic analysis to improve program understanding for Ajax applications?* Participants indicated that FireDetective – which uses dynamic analysis – allows them to understand Ajax applications more effectively, more efficiently and with more confidence. A possible explanation could be that the tool offers the option to switch to a more top-down way of understanding. From the observations and interviews conducted during the user study we identify three different ways to further support the understanding process: incorporating information about additional abstractions (such as various kinds of XML bindings and JavaScript parsing errors), exploration of other kinds of visualizations and integration with existing tools, such as Firefox' FireBug add-on.

Contributions. In this paper, we have made the following contributions:

- We have built FireDetective, a dynamic analysis tool for understanding Ajax applications.
- We have shown how to employ abstractions in the Ajax/web domain to link execution traces.
- We have carried out a preliminary user study that showed us (1) how developers traditionally go about understanding Ajax applications and (2) that dynamic analysis techniques can improve their understanding.

Future Work. An interesting avenue for future work is to explore ways to further improve program understanding of Ajax applications. At the same time we must carefully evaluate empirically how individual aspects and techniques affect the understanding process. Furthermore, a longitudinal study which explores the long term effects of such techniques in a real web development environment is our next step.

Acknowledgments. We would like to thank all volunteers that participated in our user study.

References

[1] N. A. Matthijssen, "Understanding Ajax applications by using trace analysis," Master's thesis, Tech. Univ. Delft, 2010.

[2] J. J. Garrett, "Ajax: A new approach to web applications," 2005, http://www.adaptivepath.com/ideas/essays/archives/000385.php, retrieved on December 30th, 2009.

[3] A. Mesbah and A. van Deursen, "A component- and push-based architectural style for ajax applications," *Journal of Systems and Software*, vol. 81, no. 12, pp. 2194–2209, 2008.

[4] A. E. Hassan and R. C. Holt, "Architecture recovery of web applications," in *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 2002, pp. 349–359.

[5] T. Corbi, "Program understanding: Challenge for the 1990s," *IBM Systems Journal*, vol. 28, no. 2, pp. 294–306, 1989.

[6] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke, "A systematic survey of program comprehension through dynamic analysis," *IEEE Transactions on Software Engineering*, vol. 35, no. 5, pp. 684–702, 2009.

[7] G. Antoniol, M. Di Penta, and M. Zazzara, "Understanding web applications through dynamic analysis," in *Int'l Workshop on Program Comprehension*. IEEE, 2004, pp. 120–129.

[8] B. Shneiderman, "The eyes have it: A task by data type taxonomy for information visualizations," in *Proc. Symposium on Visual Languages (VL)*. IEEE, 1996, pp. 336–343.

[9] A. von Mayrhauser and A. M. Vans, "Program comprehension during software maintenance and evolution," *IEEE Computer*, vol. 28, no. 8, pp. 44–55, 1995.

[10] B. Cornelissen, L. Moonen, and A. Zaidman, "An assessment methodology for trace reduction techniques," in *Int'l Conf. on Software Maintenance (ICSM)*. IEEE, 2008, pp. 107–116.

[11] E. Babbie, *The practice of social research*, 11th ed. Wadsworth Belmont, 2007.

[12] D. Campbell, J. Stanley, and N. Gage, *Experimental and quasi-experimental designs for research*. Rand McNally Chicago, 1963.

[13] M. Pacione, M. Roper, and M. Wood, "A novel software visualisation model to support software comprehension," in *Working Conf. Rev. Engineering*. IEEE, 2004, pp. 70–79.

[14] N. Pennington, "Stimulus structures and mental representations in expert comprehension of computer programs," *Cognitive Psychology*, vol. 19, no. 3, pp. 295–341, 1987.

[15] P. Li and E. Wohlstadter, "Script InSight: Using models to explore JavaScript code from the browser view," in *Int'l Conf. Web Engineering (ICWE)*. Springer, 2009, pp. 260–274.

[16] F. Ricca and P. Tonella, "Analysis and testing of web applications," in *Proceedings of the 23rd International Conference on Software Engineering (ICSE)*. IEEE, 2001, pp. 25–34.

[17] G. Di Lucca, A. Fasolino, F. Pace, P. Tramontana, and U. de Carlini, "WARE: A tool for the reverse engineering of web applications," in *Proc. Conf. on Software Maintenance and Reengineering (CSMR)*. IEEE, 2002, pp. 241–250.

[18] P. Tonella and F. Ricca, "Dynamic model extraction and statistical analysis of web applications," in *Proc. Int'l Workshop on Web Site Evolution (WSE)*. IEEE, 2002, pp. 43–52.

[19] W. De Pauw, M. Lei, E. Pring, L. Villard, M. Arnold, and J. F. Morar, "Web services navigator: visualizing the execution of web services," *IBM Systems Journal*, vol. 44, no. 4, pp. 821–845, 2005.

[20] S. Oney and B. Myers, "FireCrystal: Understanding interactive behaviors in dynamic web pages," in *Proc. of the Symposium on Visual Languages and Human-Centric Computing (VLHCC)*. IEEE, 2009, pp. 105–108.

TUD-SERG-2010-008
ISSN 1872-5392