Understanding Ajax Applications by Connecting Client and Server-Side Execution Traces

Andy Zaidman*, Nick Matthijssen**, Margaret-Anne Storey***, and Arie van Deursen*

*Delft University of Technology, The Netherlands
a.e.zaidman@tudelft.nl, arie.vandeursen@tudelft.nl
**Delft University of Technology, The Netherlands & University of Victoria, Canada
nick8maal@gmail.com
***University of Victoria, Canada
mstorey@uvic.ca

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. We evaluate FireDetective using (1) a pretest-posttest user study and (2) a field user study. Preliminary evidence shows that the FireDetective tool is an effective aid for web developers striving to understand Ajax applications.

1 Introduction

Over the last decade, web development has evolved from creating static web sites to creating rich and highly interactive web applications. One of the enabling technologies for creating these interactive web applications is Ajax (Asynchronous Javascript and XML), an umbrella term for existing techniques such as JavaScript, dynamic manipulation of the Document Object Model (DOM), and JavaScript's XMLHttpRequest object. Since the term Ajax was coined in 2005 [20], 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" [32]. A well-known 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 web development more complex. Classical web applications are based on a multi-page interface model, in which interactions are based on a page-sequence paradigm [32]. 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" [23]. 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 [8]. However, papers focusing on program understanding specifically for Ajax applications are scarce, as observed by Cornelissen *et al.* [13].

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 [4] 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 run-time. 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.* [2] 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.

This paper builds upon our previous work [29, 30] in which we proposed

FireDetective, and evaluated it using an exploratory pretest-posttest user study experiment. In this paper, we broaden our approach to evaluating FireDetective, as we add a field user study that we conducted with the help of two expert Ajax developers. This evaluation approach allows us to address the following research questions:

RQ1 Which strategies do web developers currently use when trying to understand Ajax applications?

RQ2 Can we use dynamic analysis, as presented through the FireDetective tool, to improve program understanding for Ajax applications?

The rest of this paper is organized as follows. Section 2 describes the design and implementation of FireDetective. Section 3 documents the design of our user study, while Section 4 describes and discusses the findings of this user study. Sections 5 and 6 respectively describe the experimental design and the findings of the field user study that we performed. Threats to validity are covered in Section 7. Section 8 discusses related work. Finally, Section 9 presents our conclusions and identifies future opportunities.

2 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, time-outs, 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.

2.1 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

[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/`.

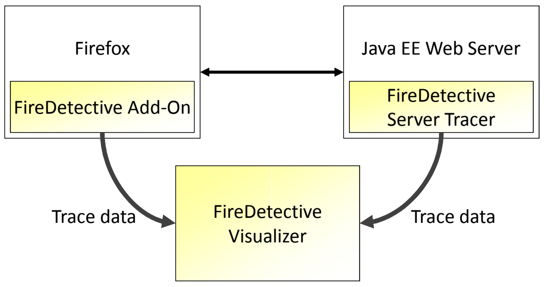


Figure 1: Architecture of FireDetective.

(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 study (see Section 3). However, the same techniques can be applied to Ajax applications with other back-ends, such as PHP or Ruby.

2.2 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. A full-page request is thus the request to load an entirely new HTML (or dynamically generated HTML) page.
- **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. Examples of non-Ajax requests are the loading of additional resources on top of the full-page in the previous category: cascading style sheets (CSS), JavaScript (js) files, etc.
- **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.

- **Time-outs** (in JavaScript) can be set to trigger a time-out handler after a specified time period has elapsed. We link time-outs to the JavaScript traces that were recorded as a result of the time-out handler being invoked, and to the JavaScript calls that started and stopped that particular time-out[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. Since 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, 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?".

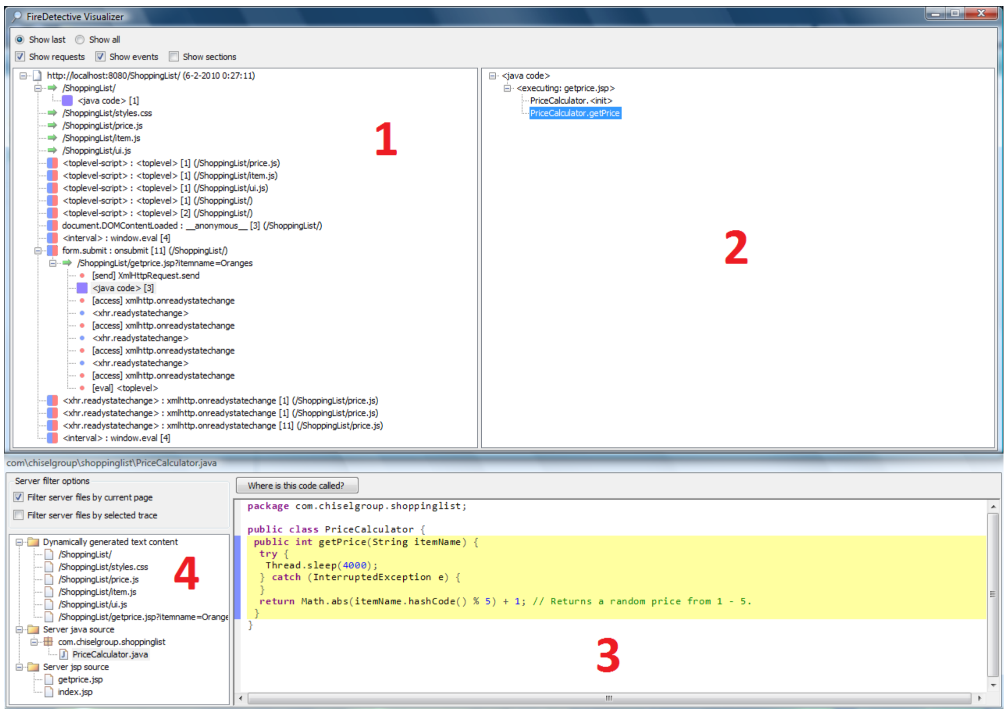
The abstractions were identified through our own experiences as Ajax developers. In Section 4 we offer possible additions to this list. We used different mechanisms for recording and reconstructing these abstractions, and linking them to the relevant traces. These mechanisms are briefly described in Subsection 2.6.

2.3 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 [42]: information visualization tools should allow for creating overviews, zooming, filtering, and providing details on demand. This design correlates with a top-down comprehension strategy [31].

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

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



[✓ Show last] Show all

[✓ Show requests] [✓ Show events] Show sections

```
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

Server filter options

[✓ Filter server files by current page]

Filter server files by selected trace

```
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. Dynamically generated 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.

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 (this also means that different invocations of the same functions are represented separately). 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 [11]. 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 allowing the user to start and stop trace 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.

2.4 Relation to FireBug

FireBug[5] is a popular Firefox plug-in that allows to edit, debug and monitor CSS, HTML, DOM and JavaScript. In essence, it is often used as a *"debugger"* for Ajax web applications, in which capacity it also frequently serves as a program comprehension tool [26].

While FireBug is currently part of the typical web developer's arsenal of standard tools, the goals of FireBug and FireDetective are not quite similar. In contrast to FireBug, which simply displays a list of (Ajax) requests, FireDetective is also able to relate these request to relevant code fragments, both on the browser-side (the JavaScript) and on the server-side (in our case, Java code). FireBug allows to inspect the current values of variables, parameters, and of the `XMLHttpRequest` object. As such, it allows you to see which parameters are sent to the server back-end, but it does not allow

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

[5]See `http://getfirebug.com/`.

you to immediately relate that request to what is happening on the server. Moreover, it is mainly targeted at program comprehension in the small, for understanding situations where you already have a good starting point, which you can mark with a *breakpoint*.

FireDetective on the other hand, does not offer these low-level inspection features, but instead, focuses on program comprehension in the large, by offering the ability to relate actions in the JavaScript code in the browser to code that is being executed on the server.

As a tool, FireBug is very important as it is currently widely used by web developers. FireBug is also be part of our experiment in Section 3.

For completeness, we also mention DragonFly, an Opera plug-in with functionality similar to FireFox's FireBug plug-in[6], and the Google Chrome Web Developer Tools[7].

2.5 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. Since 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 Firefox FireBug add-on currently[8] 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. "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

[6]For more information, see: `http://www.opera.com/dragonfly/`

[7]For more information, see: `http://code.google.com/intl/nl/chrome/devtools/`

[8]FireBug version 1.5.0.

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.

2.6 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 time-outs (and intervals) are registered by wrapping all related properties and functions (e.g., `XMLHttpRequest.responseXML`, `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. However, 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.

3 Design of the user study

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 [3], yet it allows us to report on facts of real user-behavior, even those observed in under-controlled, limited-sample experiences. In particular, we are using a *one group pretest posttest* design, which entails that there is only an experimental group and no control group. This type of experiment is called pre-experimental because it does not allow to identify an event related to the dependent variable that intervenes between the pretest and the posttest where the effects could be confused with those of the independent variable [3].

In the user study, we observed eight 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 (see Section 3.6 for details). 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 (RQ1).*
- **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 (RQ2).*

Our approach is exploratory as we are still at an early stage in this research project. We focus on observing participants as they work on assigned tasks with and without the FireDetective tool. We asked participants to think aloud during the study, and since the study was conducted in a lab setting we were able to make audio and screen recordings for later analysis. We also gave questionnaires to the participants to determine their perceptions of the benefits of using dynamic analysis both before and after using FireDetective. After each part, participants were subjected to a short interview. In the following sections, these aspects are described in more detail.

3.1 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 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 4), 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 about the tools 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 be stalled with a tool issue for too long. A short interview asking participants about encountered problems concluded part A.

3.2 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 were incompatible at the time of the user study. This incompatibility originates in Firefox' debugger service, which only allows a single listener per notification type, which is a problem because FireBug and FireDetective both listen for the same type of events. Additionally, FireDetective injects extra JavaScript code into each web page that is being viewed. This code is again stripped out in FireDetective's visualizer. However, the injected code still shows up in FireBug, which can lead to confusion. These factors made it non trivial to make FireBug and FireDetective compatible.

As in part A of the study, the focus lies on observing participants as they work on tasks. However, in part B we also collected data on the user's perceptions of the benefits from using the FireDetective tool. We use a pretest-posttest design [6], in which a pretest questionnaire was used to measure the participants' expectations prior to using FireDetective, while a posttest questionnaire measured the participants' experience after using the tool (see [28, p.85]). In particular, we evaluated four attributes:

- **Better understanding.** Will the tool allow web developers to understand Ajax applications more effectively?
- **Quicker understanding.** Will the tool allow users to understand Ajax applications more efficiently?
- **More confident about understanding.** Will use of 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. Will 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 (see Tables 1 and 2 for the pretest and posttests, more information is also available in [28, p.85]).

After a 10-minute introduction in which we showed all features of FireDetective, participants worked on a different set of program understanding

A. Software development experience
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
5 = I've been using it regularly for over two years now
1. Java
2. Java Server Pages (JSP)
3. JavaScript
4. Dojo JavaScript framework
5. Eclipse
6. Firefox
7. FireBug
8. Java PetStore
B. Understanding web applications
In this experiment, you will be using a tool that uses dynamic analysis to show the real-time execution of web applications. It displays the execution of scripts and functions in the browser (JavaScript code), the execution of class methods on the server (Java code), and how the browser and server communicate. *For each of the next statements, please indicate to what extent you agree with them, ranging from 1 (completely disagree) to 5 (completely agree).*
1. Such a tool could allow me to better understand web apps.
2. Such a tool could allow me to be more confident that I really understand the web application that I'm investigating.
3. The value added by such a tool will be minimal.
4. Such a tool could save me time.

Table 1: Pretest questionnaire

tasks for 25 minutes. The decision to have less time for this part was made to keep the complete duration of the study under two hours.

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

3.3 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"[9]. The application consists of 12KLoc, which are written in a variety of languages, such as HTML, CSS

[9]See http://java.sun.com/developer/releases/petstore/, retrieved on November 14th, 2010.

For each of the next statements, please indicate to what extent you agree with them, ranging from 1 (completely disagree) to 5 (completely agree).
A. Tool user experience
1. I found FireDetective easy to use.
2. FireDetective should have been integrated with Eclipse.
B. Tool adequacy
1. There's added value in using dynamic (i.e., runtime) information for analyzing web applications.
2. The value added by a tool like FireDetective is minimal.
3. A tool like FireDetective saves me time.
4. A tool like FireDetective allows me to better understand web apps.
5. A tool like FireDetective makes me more confident that I really understand the web application that I'm investigating.
C. Tool features
Below are a number of features of the FireDetective tool. Please select your top 3 features. Put a "1" next to the best feature, a "2" next to the second best, and a "3" next to the third best feature. *Please mark features that you didnt find useful with an 'X'.*
1. Having an overview of events and the JavaScript functions that handle them.
2. Being able to directly jump from (Ajax) requests to the corresponding server side code.
3. Real-time trace analysis, i.e.: (almost) no delay between capturing traces and analyzing them.
4. Marking sections of a trace using the Firefox add-on, by using Begin mark and "End mark".
5. Being able to easily track xhr (Ajax) requests.
6. Filtering packages and java files based on the current page or trace.

Table 2: Posttest questionnaire

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.

3.4 Task design

The study required the design of two task sets, one for each part of the study. We constructed the tasks ourselves, 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. See Tables 3 and 4 for an overview of the two task sets[10].

[10]The complete task descriptions can be found in [28, p.88].

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 *et al.*'s taxonomy of 9 principal activities [35], 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, (1) to limit the number of tasks, (2) to reduce the risk of our participants becoming fatigued during the study and (3) because these three activities are less atomic and can be composed of several activities that are already captured in the first six activities. For completeness, the other three principle activities from Pacione *et al.* are: A7. Investigating patterns in the systems execution, A8. Assessing the quality of the systems design and A9. Understanding the domain of the system.

Since we were keen to observe how FireDetective would be used on unfamiliar code, we strove to choose tasks for the second set that would involve code not inspected in part A of the study. Nevertheless, a learning effect might be possible due to the fact that the software system remains the same. However, this should not impact our results, as we are not measuring an increase (or decrease) in efficiency from developers using FireDetective, but instead, we are gauging the FireDetective user experience.

3.5 Pilot sessions

Three pilot sessions were conducted to fine tune the study. Two of the three pilot participants were co-workers of the second author of this paper; the third pilot participant was recruited via the same route that we recruited all of the other participants. All pilot sessions were done in a similar way as the actual study, except for the fact that we were particularly interested in whether all tasks were clear, whether the tasks were deemed too difficult and in what other ways could we improve the settings of our experiment.

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

Task 1 – The headline bar	
Near the top of most pages of the Java Petstore application is a gray headline bar. The headline text switches from time to time.	
a)	Give the names of the JavaScript functions that are related to the switching of the headline text.
b)	Explain how these functions call each other when the text switches, and how they keep the switching going.
c)	From what web URL does the application get the headlines? Where in the code can you change that? (give file name + line number)
Task 2 – Server code	
The Petstore consists of 6 sub pages: home, seller, search, catalog, maps and tag.	
a)	Which of those sub pages call either directly or indirectly methods of the GeoCoder class? (package: com.sun.javaee.blueprints.petstore.proxy)
b)	Is the class SQLParser (package: com.sun.javaee.blueprints.petstore.search) being called - either directly or indirectly - on the search page?
Task 3 – Seller page	
Navigate to the seller page.	
a)	Clicking on the next button does not trigger the forms validation check. The Java Petstore manager has encountered several users who complained about this. He asks you to change the pet store, such that validation is also performed after clicking the next button. Which function or method do you need to modify? How do you modify it?
b)	The user is required to enter a city and state on the second page of the form. As the user types in these text fields, an auto complete box shows up that allows the user to select cities and states but only US cities are listed. Of course, this is unacceptable! Which parts of the application (e.g. which functions or class methods) need to be modified for Canadian provinces and cities to show up in the auto complete box?
Task 4 – Popup view	
Navigate to the search or tags page. Note that a popup appears when you hover over a pet name with the mouse.	
a)	What JavaScript functions are involved on the client side? (give their names and locations: file name(s) + line numbers)
b)	What Java classes and JSP files are involved on the server side?
c)	How come the popup doesnt appear if you quickly hover over a description?

Table 3: Task set A

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.

Task 1 – Search & tag page	
Navigate to the search page and click "Submit". Clicking the little icons under "Map" allows you to (un)check all checkboxes.	
a)	List the JavaScript functions that are involved in this process.
Navigate to the tags page. Notice how you can click the tags to update the list.	
b)	Does clicking the tags trigger an ajax request? If yes, which JSP file(s) and server class(es) are involved?
Task 2 – Server code	
a)	Is the IndexDocument class (package: com.sun.javaee.blueprints.petstore.search) really being used on the search page? If yes, give a possible chain of events/calls leading to a use of the class (e.g., "user moves mouse" $\rightarrow$ handled by handleEvent $\rightarrow$ etc. $\rightarrow$ calls IndexDocument).
b)	The Petstore consists of 6 sub pages: home, seller, search, catalog, maps and tag. Which of those sub pages make use either directly or indirectly of the EntryFilter class? (package: com.sun.javaee.blueprints.petstore.controller)
c)	What is the purpose of the ImageAction class (package: com.sun.javaee.blueprints.petstore.controller.actions)?
Task 3 – Catalog page	
Navigate to the catalog page.	
a)	The Pet store owner asks you to speed up the scrolling of the filmstrip at the bottom. Which parts of the code do you modify? (give file names(s) + line numbers)
b)	Go to the "fish" and then "small fish" category. Scroll to the right in the bottom bar and try to flag "Nicks goldfish" as inappropriate. This is supposed to delete the pet. Why does this not work? (i.e.: where's the bug?)
Task 4 – Catalog & search page	
Go to the catalog page.	
a)	You can click on the "star strip" to rate a pet. How is the rating computed?
b)	Moving over a category on the left causes the category to expand and show its subcategories underneath. List the JavaScript functions that are involved.
Go to the search page.	
c)	At the moment, searching causes the whole page to refresh. In order to improve the user experience of the pet store, we want to ajaxify this process: i.e., we want search results to appear without refreshing the page. Which parts of the application (client side + server side) would we need to modify?

Table 4: Task set B

3.6 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.

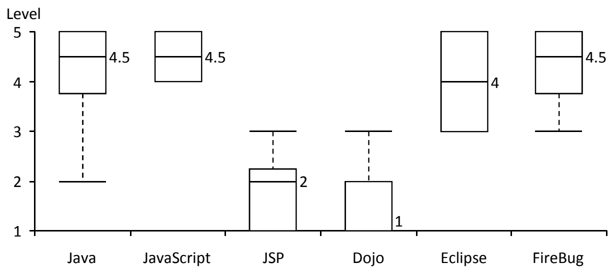


Figure 3: Box plots of participants' experience with relevant technologies and tools. The values on the 5-point scale (vertical axis) correspond to 1 = "Never used it", 2 = "Used it for a couple of hours or less", 3 = "Used it for one or two projects", 4 = "I use it regularly" and 5 = "I've been using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

This turned out to be the case for 8 out of 9 participants that we recruited. One participant had no web development experience and this was reflected in the results: the participant was only able to complete the most basic tasks. Since the participant was clearly not representative of the target population we excluded this data. As such, only 8 participants are considered in the analysis from now on.

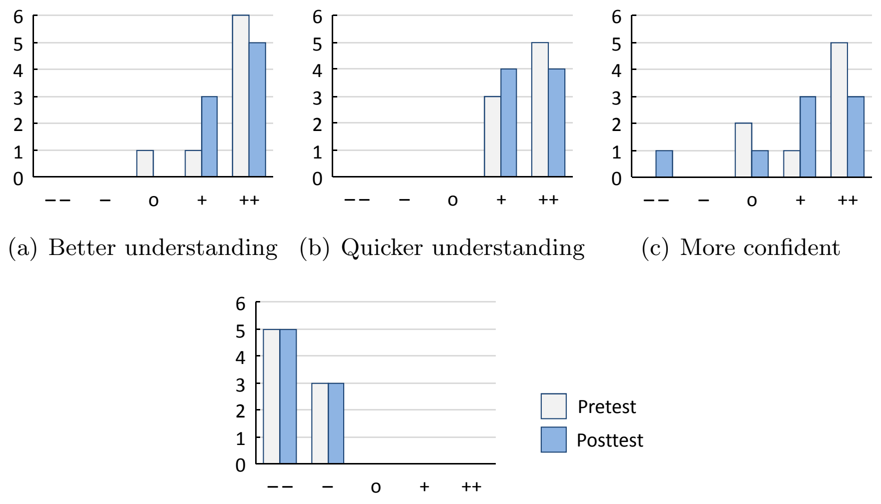
Our 8 participants represent our target population quite well. Five had a professional web development job: one full-time and four part-time. Two others had a professional software development job: one full-time and one part-time. Both of these participants indicated that they worked on web development projects for at least a part of their jobs. Except for the two full-time developers, the six other participants were either computer science or software engineering students: four undergraduate and two 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 low number. However, technologies like Ajax have not been around for that long: at the time of our study, the term Ajax had been coined less than 5 years ago [20]. 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 5-point Likert 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 7, 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.

4 Findings and discussion of the user study

Our findings cover the Ajax understanding strategies currently used (part A, Section 4.1), as well as the way in which dynamic analysis in general and FireDetective in particular can support this understanding (part B, Section 4.2).

Although our focus in this study was not to assess 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 is 6 (min. 4, max. 8) for part A, and 7 (min. 5, max. 9) for part B. Roughly two thirds of these attempts led to the correct answer in both parts of the study (for details see Tables 5 and 6).



(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.

	Mean	Min	Median	Max
Attemped	6	4	6	8
Correct	4	2.5	3.75	6.5
Fraction correct	0.66	0.5	0.65	0.81

Table 5: Correctness scores for part A of the user study (0.5 scores were given in case of a partially correct solution).

	Mean	Min	Median	Max
Attemped	6.75	5	7	9
Correct	4.69	3.5	4.75	6
Fraction correct	0.71	0.39	0.71	0.9

Table 6: Correctness scores for part B of the user study (0.5 scores were given in case of a partially correct solution).

4.1 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 [37], 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 "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 (at the time of the study) 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 in the DOM tree (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) a 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, our findings support our argument that dynamic analysis would be beneficial in tool support.

4.2 Part B: Support through dynamic analysis

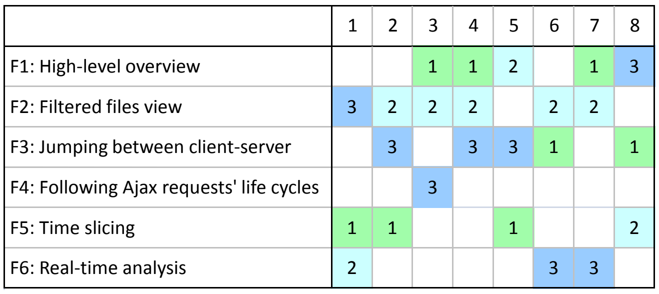
Central to this part of the study is our second research question: "Can dynamic analysis improve program understanding for Ajax applications?" We explore this question by considering whether dynamic analysis as provided by the FireDetective tool can be used to improve understanding of Ajax web applications. Furthermore, if this is indeed 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 questionnaires, the questionnaire about feature usefulness, observations of participants using the tool and the final interview.

4.2.1 Pretest–posttest

The results of the pretest and posttest are shown in Figure 4. From the results we can see that the pretest and posttest results are fairly similar: the participants did not completely switch their opinions before and after using the tool. The pretest results are quite positive, which suggests that users feel they could benefit from tool support. The posttest results are positive as well, and show that participants were not disappointed with FireDetective.

In particular, participants indicate that the tool can help them to understand web applications more effectively (Figure 4(a)) and more efficiently (Figure 4(b)). Participants also seem convinced that the tool helps them to be more confident about their understanding of the web application they are investigating (Figure 4(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 acknowledged that the tool adds value (Figure 4(d)).

Discussion. While these are preliminary findings, we consider them encouraging. They suggest that FireDetective, which leverages dynamic analysis techniques, is indeed capable of improving program understanding for



	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.

Ajax applications.

4.2.2 Features

We asked the participants' opinion on six features of FireDetective (see Figure 5) 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 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 preferred 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 are actually useful.

4.2.3 Observations and interview

Besides the expected learning curve and usability issues (see [28]), 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. Since 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 a first indication that features of FireBug and FireDetective are considered 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 in this initial study 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 clear 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.

5 Design of the field user study

The user study we describe in Sections 3 and 4 helps us understand how a tool such as FireDetective can aid in understanding Ajax applications. However, the time that each participant spent with FireDetective was limited to 25 minutes. Moreover, the study was also conducted in a controlled lab setting using mainly student programmers (although most had industrial experience) and doing preassigned tasks. What we lack is a more in-depth look at how FireDetective can be used in more open ended tasks. Furthermore, we also wanted to gauge whether professional web developers with more background in Ajax and related technologies would have a different opinion on the strengths and weaknesses of FireDetective. This section describes the experimental setup of a user study conducted in a field setting with these goals in mind. This field user study is meant to strengthen and broaden our insights into RQ2 in which we investigate whether dynamic analysis, as presented through the FireDetective tool, can improve program understanding for Ajax applications.

5.1 Field setting

In order to get a more in depth perspective of how FireDetective would be used by seasoned and experienced professional programmers, we set up a field user study with two professional web developers from a company called Mendix[11]. Mendix was founded in 2005 as a spin-off from the Delft University of Technology and the Erasmus University of Rotterdam. In 5 years, the company has grown from a start-up to a company with 75 employees. Their core business is to rapidly develop business applications that can easily be integrated into existing IT environments. Mendix is at the forefront of technology, as they are actively using relatively new technologies from the realms of model-driven engineering (MDE) and Ajax. Mendix was kind enough to be willing to cooperate in evaluating FireDetective and two experienced web developers volunteered to participate in our field user study[12].

5.2 Experimental setup

The field user study consisted of a single half-day session with the two developers from Mendix. The study was again centered around the Java Pet Store (see Section 3.3). This provided us with the benefit that we already had a reasonable level of knowledge of the application, while the two developers that participated in the field user study had no experience with it.

Step 1: Demo of FireDetective. We started the session with a short demo of FireDetective, in which we showed the two developers all features of FireDetective on a very small toy Ajax application.

Step 2: Free exploration. In this step we asked the two developers to freely explore the Java Pet Store application for around 1.5 hours. We gave

[11]More information: `http://www.mendix.nl`

[12]One of these two developers was an undergraduate student in the Software Engineering Research Group of the Delft University of Technology a few years back.

them the goal of getting a good understanding of the implementation of the major functionality in the Java Pet Store, and told them that we would discuss the implementation details of this functionality later on during the session. We provided technical assistance during the user study. The two developers were allowed to work in pair-programming style and we asked them to think aloud, which allowed us to gain insight into their way of working with FireDetective.

Step 3: Questionnaire. When the two developers were satisfied with their reconnaissance mission of the Java Pet Store, we gave them the same same set of questions that we also used in the posttest of our user study (see Section 4.2). This step only took a few minutes and was mainly done to be able to compare between both groups of subjects.

Step 4: Contextual interview. While we already gained quite a lot of information during the free exploration phase, we intensified the interview once the developers felt they were comfortable with their understanding of the Java Pet Store. In particular, we used a contextual interview [24]. This contextual interview already started during the second step, where we observed how the developers explored the system. In particular, we took note of which questions they were asking and how they were using FireDetective to answer these questions. Subsequently, we continued the interview and we aimed to further explore the possibilities of FireDetective and identify circumstances in which FireDetective can be of benefit. In order to steer this conversation, we used the work by Sillito *et al.* [43], who identified a set of 44 typical questions developers have when maintaining a piece of software. In order to save the participants' time, we focused on questions that could benefit from the dynamic analysis of web applications. The eliminated questions could either easily be answered using other static analysis features in an IDE such as "Where does this type fit in the type hierarchy?" or they were not pertinent for web applications. See Table 8 for the 25 questions that we asked the developers. For each of the questions that we discussed, we asked the two developers to rate the usefulness of FireDetective to answer the question. For this, we used a 5-point Likert scale that ranged from "totally disagree" (score 1) to "totally agree" (score 5). This interview took close to two hours and during the interview, the developers frequently went back to FireDetective to get a good appraisal of FireDetective on specific aspects that are highlighted by Sillito *et al.* 's questions.

5.3 Participant profile

Both Mendix developers can be considered experienced software engineers, with a broad experience in web technology. One developer has a university background and 4 years of professional experience, the other developer has 10 years of development experience and holds a college degree in IT. In a similar manner to Section 3.6, we gauged their experience with some key web application development technologies. Both developers have been working for more than 2 years with technologies such as Java, JavaScript, the Dojo JavaScript Framework, Eclipse, Firefox and FireBug on an almost daily basis. They consider their experience with Java Server Pages (JSP) to be rather limited, but they have previously come in contact with it. Besides the aforementioned technologies, they have experience with other JavaScript frameworks such as JQuery[13] and Prototype[14]. Other technologies that they are familiar with are HTTP proxies for analyzing the traffic between client and server, various JavaScript debuggers and profilers (e.g., Venkman[15]), HTML, and Cascading Style Sheets (CSS).

6 Findings and discussion of the field user study

We already indicated in Section 5 that the developers first had the opportunity to freely explore the Java Pet Store application for 1.5 hours. During this free exploration they formed and refined hypotheses. When they were satisfied with an initial hypothesis on the implementation of a particular feature, they started investigating the application itself in order to verify their hypothesis.

For understanding the Java Pet Store, they typically investigated the HTML and/or JavaScript code and/or used the FireDetective tool. During the exploration, they continued to think aloud and continuously shared their thoughts and told each other which direction they should follow during their reconnaissance.

After the free exploration of FireDetective, we presented the two Mendix developers with a short questionnaire (Section 6.1), and we continued our contextual interview with the aim of getting more in-depth feedback on tool requirements (Section 6.2).

6.1 Questionnaire

About FireDetective as a tool. Both developers indicated that they found FireDetective easy to use (and gave it a score of 4 on a 5-point Likert scale ranging from “Totally disagree” to “Totally agree” — also see Table 7). Both developers gave a score of 5 to the statements that there is added value in using dynamic analysis for *analyzing* and *understanding* web applications. They also gave a score of 5 for the statement that FireDetective allows them

[13]`http://jquery.com/`
[14]`http://www.prototypejs.org/`
[15]`http://www.mozilla.org/projects/venkman/`

to better understand web applications. One score of 4 and one of 5 was given to the statement that "FireDetective makes me more confident that I really understand the web application", while "FireDetective is likely to save me time" got a score of 5 from both developers.

	Developer 1	Developer 2
I found FireDetective easy to use	4	4
FireDetective should be integrated into Eclipse (or another IDE)	1	3
There's added value in using dynamic analysis for *analyzing* web applications	5	5
There's added value in using dynamic analysis for *understanding* web applications	5	5
A tool like FireDetective is likely to save me time	4	4
A tool like FireDetective allows me to better understand web applications	5	5
A tool like FireDetective makes me more confident that I really understand the web application that I'm investigating	4	5

Table 7: Results to the questionnaire about the FireDetective tool on a scale of 1 (totally disagree) to 5 (totally agree).

While both developers filled in their questionnaire forms individually, their scores agree on almost all statements. When we then compare their answers to Figure 4, subgraphs (a), (b) and (c) we see that these two experienced developers are equally positive (if not slightly more) about FireDetective than the participants from the user study were.

Features of FireDetective. In a similar style to the user study, we also asked the two developers to provide their opinion on the key features of FireDetective (also see Figure 5). We asked them to indicate their top 3 most-liked features. Both developers liked the overview of events and the JavaScript functions that handle them the most, with both developers stating that other tools like FireBug allow them to place breakpoints during debugging, but failing to give them a chronological overview, something which FireDetective does provide. The second-best feature of FireDetective is the fact that FireDetective allows them to track Ajax requests. For the third feature, one of the developers indicated the fact that you can mark sections of a trace, which helps to avoid cluttering, while the other developer indicated the fact that FireDetective allows to jump directly from Ajax requests to the corresponding server-side code, avoiding the need for context-switches when going from one tool to another.

Number	Question	Developer 1	Developer 2
Q2	Where in the code is the text in this error message or UI element?	4	4
Q3	Where is there any code involved in the implementation of this behavior?	5	4
Q12	Where is this method called or type referenced?	2	2
Q13	When during the execution is this method called?	2	2
Q14	Where are instances of this class created?	2	1
Q15	Where is this variable or data structure being accessed?	2	2
Q18	What are the arguments to this function?	2	2
Q19	What are the values of these arguments at runtime	2	1
Q20	What data is being modified in this code?	3	3
Q21	How are instances of these types created and assembled?	3	2
Q22	How are these types or objects related?	1	2
Q23	How is this feature or concern implemented?	5	5
Q25	What is the behavior these types provide together and how is it distributed over the types?	4	4
Q26	What is the "correct" way to use or access this data structure?	2	3
Q27	How does this data structure look at runtime?	1	3
Q28	How can data be passed to (or accessed at) this point in the code?	5	5
Q29	How is control getting (from here to) here?	5	5
Q30	Why isn't control reaching this point in the code?	4	2
Q31	Which execution path is being taken in this case?	5	5
Q32	Under what circumstances is this method called or exception thrown?	2	2
Q33	What parts of this data structure are accessed in this code?	4	4
Q37	What is the mapping between these UI types and these model types?	4	5
Q39	Where in the UI should this functionality be added?	4	4
Q40	To move this feature into this code, what else needs to be moved?	1	1
Q41	How can we know this object has been created and initialized correctly?	1	2

Table 8: The 25 questions that we selected from [43] and that we asked the developers, including the score of the developers with 1 indicating "Totally disagree" and 5 indicating "Totally agree". The question-numbering is taken from [43].

6.2 Contextual interview

As we already discussed in 5.2, part of the contextual interview is based on the typical maintenance questions that were presented by Sillito *et al.* [43]. The responses to each of these questions can be seen in Table 8. While we will not touch upon all 25 questions in detail in our discussion, we will highlight those questions that sparked the most interesting discussions. For the remainder of this section, we will adhere to the question numbering as used by Sillito *et al.* [43].

We first gave the two Mendix developers the aforementioned set of questions and asked them to think about the questions and rate the usefulness of FireDetective for answering each of the questions, on a 5-point Likert scale. In a subsequent step, we compared the responses of the developers in group and tried to bridge differing opinions. In all cases where we identified differing opinions – 2 cases to be precise –, this was due to a different interpretation of the question. In addition, we did not only compare notes, we also collected anecdotes on how FireDetective was useful for answering a question. In particular, when we were discussing the questions in a group, the developers were frequently referring to using some FireDetective functionality to investigate the implementation of a particular feature. In the next paragraph, we highlight some of the more interesting anecdotes. Important to note as well is that during the entire interview, FireDetective was available to the developers so that they could check up on certain ideas or opinions.

Interview highlights. We will now discuss some of the highlights of the interview.

Q2 *Where in the code is the text in this error message or UI element?* Both developers agreed here that FireDetective is useful (score: 4) and that finding the origins of an error message in the trace is actually more efficient than using a textual search tool, which would sometimes need to be applied on both the client and server-side code in order to find the origins of an error message.

Q3 *Where is there any code involved in the implementation of this behavior?* Many times during the exploration of the Java Pet Store case, the two developers formed a hypothesis that a particular functionality was purely implemented on the client side, or both on the client and the server-side. FireDetective helped them to verify their hypotheses. As an example, the two Mendix developers hypothesized that when the Java Pet Store puts a marker on the Google Map when clicking on the address of a physical shop, this functionality is purely implemented on the client-side. Investigation with FireDetective confirmed their initial hypothesis. The developers valued FireDetective's ability to answer this question with scores of 4 and 5.

Q23 *How is this concern or feature implemented?* Both developers strongly agreed that FireDetective comes in very handy for answering this question and both gave a score of 5 for FireDetective's ability to answer this question. Both developers really appreciated the trace marking feature of FireDetective, which effectively enabled them to do feature location [47].

Q28 *How can data be passed to or accessed at this point in the code?* Both developers immediately indicated that FireDetective makes it very easy to see how data can be passed to a point in code, but how it is accessed is not always clear. For the first option, both developers decided to give a score of "totally agree" (5) for FireDetective's capabilities to support this question. What they particularly liked about FireDetective in this regard, is its ability to connect the client and the server-side code, with the added benefit of having the possibility to navigate both the client and the server side code from within FireDetective. This effectively allows them to follow the control flow, which helps to understand how data can be passed.

Q29 *How is control getting (from here to) here?* Again, both developers indicated that FireDetective supports this question very well (score of 5 by both). Furthermore, the developers stressed that the FireDetective allowed them to get a really good overview of everything that is happening at the client-side, and while they appreciated the connection to the server, they thought that the main strength of FireDetective was its way of visualizing calls and interactions that occur within the client-side (browser). At this point, they also compared FireBug with FireDetective, stating that for understanding purposes FireDetective is superior because it provides a better overview of what is going on, compared to the breakpointing facility offered by FireBug.

Q31 *Which execution path is being taken in this case?* FireDetective's facilities to answer this question were rated with a score of 5 by both developers.

Additional insights obtained from the contextual interview. In this section we will briefly discuss some of the additional insights that we gained during the contextual interview with the two developers. These additional insights could not be directly mapped to one of the comprehension questions in the previous part.

One of the developers suggested to incorporate profiling functionality into FireDetective, to provide more obvious insights into the performance of the application. This would effectively mean integrating some of the functionality of DynaTrace, a tool both developers knew (also see Section 8), into FireDetective. There was actually no consensus on the benefits of integrating all functionality (for understanding and for profiling) in one tool versus using two separate tools, while each separately would possibly be more powerful in its own right.

With regard to FireBug, the two Mendix developers would appreciate the ability to investigate the actual values of parameters, variables and return values in FireDetective, a feature that FireBug currently does have. Intuitively, one of the developers wanted to start up FireBug during the exploration phase, but quickly abandoned this route, when he realized that

FireDetective and FireBug were unable to be used on the same installation of Firefox.

They also added that they did not see any benefit from using the Java IDE to inspect the application under study, as most of the material they needed could be inspected right from within FireDetective. They did add however, that they would prefer to see the whole JavaScript file instead of only snippets of JavaScript code. Technically, FireDetective always tries to provide users with a context, instead of just showing a single function. However, currently FireDetective cannot reconstruct a context for most types of dynamically generated code (e.g., `eval('alert(123)')`), which is what the developers were alluding to in their remarks.

They finished the interview by stating that they felt that FireDetective excelled at giving the developer a feeling of confidence of his understanding, which they rated as more important than any time-gain from using FireDetective.

6.3 Discussion

The two experienced web developers from Mendix that we recruited for this user study have provided us with additional indications of the perceived usefulness of FireDetective as a program comprehension tool for understanding complex Ajax web applications. Their opinions are similar to the results that we obtained from our first user study.

In particular, if we compare the results from the posttest of the user study (see Figure 4) to the opinion of the two Mendix developers, we see a similar trend: web developers are convinced that FireDetective can help them in understanding applications. From the insights that we gained from interacting with the two expert developers, however, we gathered that while FireDetective might speed up the comprehension process, the ultimate benefit they see is that FireDetective increases their confidence in their understanding.

Also of interest to note is that the two expert developers found FireDetective's features to investigate client-side interactions to be the most useful, because it is at the client side that things often become difficult to follow. They still appreciated the fact that they also had the opportunity to investigate the server-side behavior, without having to make a context-switch to an IDE or another tool.

Furthermore, the hypothesis-driven approach that the developers followed for understanding the Java Pet Store gives an indication that they were following a top-down comprehension strategy, where they created a hypothesis, executed part of the application under study, and started investigating the behavior drilling down to the source code level to accept or reject their hypothesis. This is similar to what we saw in our user study.

7 Threats to validity

In this section we identify factors that may jeopardize the validity of our results and the actions we took to reduce or alleviate the risk [41, 48].

7.1 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. For both user studies, we mitigated this concern by indicating to participants that only honest answers were valuable. Nevertheless, we recognize that such a bias possibility still exists.

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 of the user study 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 desired. For part B of the user study, we made sure to explain *all* features of FireDetective, so that participants would not be biased towards using any feature in particular. For the field user study, we showed all features of FireDetective to the two developers.

The tasks of the user study might have been too easy or too difficult. However, through three pilot sessions we adjusted the task difficulty level accordingly. Also, participants of the user study 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. For the field user study, we indicated that the free exploration period would take approximately 1.5 hours, but that if the developers felt the need, they could continue beyond this time frame.

7.2 External validity

A concern regarding the generalizability of the results of the user study is that most participants of the user study were students. However, as shown in Section 3.6 a lot of these participants had a relevant part-time job. Participants of the user study 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. Additionally, for the field user study that we performed, we recruited two experienced web developers and while they did not execute the same tasks, their opinion of FireDetective does not differ much.

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., [27]. While the two experienced web developers that we recruited for the field user study had no prior knowledge of the Java Pet Store, they actually acknowledged that this project contains many of the typically (Ajax) idioms that you would also find in industrial-strength Ajax web applications, which is an extra argument for the representativeness of the Java Pet Store application.

The tasks for the user study 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 [35] to make sure that the tasks are realistic and cover a significant portion of the program comprehension spectrum.

Similarly, the questions that we discussed with the two Mendix developers might not have been realistic. In order to mitigate this threat, we reused the list of questions that was previously identified by Sillito *et al.* [43].

Both the user study that we describe in Section 3 and the industrial field study from Section 5 let the participants deal with a software system that they are not familiar with; both experiments deal with a situation in which the participants are considered *software immigrants* [44], i.e., developers that are getting to know the domain of the case study, in this case the Java Pet Store. As such, the findings that we report upon are based on developers trying to find their way in a previously unknown software system and do not reflect situations were developers are already familiar with the system. We acknowledge that a follow-up study should also investigate the usefulness of FireDetective in situations where the developers are already familiar with the domain.

8 Related work

Reverse engineering approaches can generally by categorized into static, dynamic or hybrid (combining static and dynamic) analysis techniques. This section provides a brief overview of approaches that use dynamic analysis [13]. We start by discussing some general trace analysis techniques, after which we focus specifically on techniques for reverse engineering and understanding web applications.

8.1 Trace analysis

Trace analysis concerns itself with with sequences of run-time events and how these sequences can be used to gain insight into the workings of the program. Since traces may quickly grow to massive proportions [49, 50, 53], we need ways to deal with their size [13]. We consider two common ways to do so: *trace reduction* and *trace visualization*, which are often combined.

Trace reduction Trace reduction, or trace compaction, refers to the act of transforming traces such that they become smaller. Most techniques are automatic, i.e., they require no user intervention. Techniques may be divided into several categories [11]:

- **Ad-hoc methods** like (1) defining start and end points within the code, (2) extracting time slices from a trace, and (3) sampling of traces [7, 19].

- **Language-based filtering methods** in which particular kinds of programming constructs can be omitted from a trace without sacrificing too much of the information the trace conveys. Examples are getters and setters that are called from within a class (when called between classes, getter and setter accesses can indicate important relationships!), and constructors and destructors of unimportant or unused objects [9, 21]. We can also filter elements of the program or its libraries, i.e., calls to specific components, classes, methods, etc.

- **Metric-based filtering methods** can be used to determine which parts to keep and which parts to discard from a trace. Examples are: using stack depth as a metric, i.e., filtering all calls above a specific depth [36] or below a specific depth [9]. Hamou-Lhadj and Lethbridge put forward a utilityhood metric that indicates the probability that a specific method is a utility method, which is based on fan-in and fan-out analysis, and use a threshold value to filter parts of the trace [22]. A similar technique for finding important classes has been proposed by Zaidman and Demeyer [52].

- **Trace summarization** is meant to find patterns within traces to compact these patterns. Typically, there are a lot of those patterns, since programs often contain repetitions, and “execution patterns of iterative behavior rarely justify the space they consume” [36]. Examples are methods based on string matching [45], run-length encoding or grammars [39], techniques that are borrowed from the signal processing field [25, 51] and approaches that use information from source code [33]. A question that arises when identifying patterns, is how far we should go with generalizing parts of traces to patterns. Seldomly will we see many exact recurrences of a pattern. Instead, each recurrence often differs by a slight amount [36]. De Pauw *et al.* propose various measures to decide which parts can be considered equivalent, such as: class identity (the same classes are being called), message structure identity (the same methods are being called) and repetition identity (different numbers of repetition are considered the same) [36].

Trace visualization Trace visualization is a popular research area: many techniques have been suggested. Sequence diagrams - and variations of them - are the most common way to visualize execution traces. Bennett *et al.* investigate the importance of several features of sequence diagrams, and provide a survey of different approaches [5]. Rather than mentioning every trace visualization technique that has been proposed over the years, we mention several techniques that, in our opinion, are among the more interesting and novel ones. Reiss [38] puts forward a real-time visualization of program activity in the form of real-time-box views. Such a view consists of a grid in which every square represents information about a single problem element (e.g., class, method, etc.). Ducasse *et al.* take this idea a step further by introducing polymetric views, a more general version of the former views [18]. For example, instead of squares in a grid, they use nodes in a graph to represent program elements. Cornelissen *et al.* describe the idea of circular bundle views, in which a systems components are shown on the boundary of a circle, and bundles within the circle represent relationships between components [10, 15]. In subsequent research, they also investigated the effectiveness of their circular bundle views in a controlled experiment [14].

8.2 Dynamic analysis for understanding web applications

Early web application reverse engineering efforts were mainly focused on architecture reconstruction, e.g., [2, 17, 23, 40, 46]. Static analysis alone does not suffice because of the dynamic nature of web applications [2, 46], 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.* [16] 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 [27] 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 [34] 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*[16]. DynaTrace Ajax and FireDetective are quite similar: they both record execution traces, they both use abstractions from the Ajax domain to link traces, and they both combine client-side and server-side data. However, DynaTrace is primarily focused on performance analysis, whereas FireDetective is primarily focused on improving understanding. FireDetective lacks performance analysis features, but instead has features that aid the program understanding process, such as showing code in its original context.

A recent development is DynaRIA, which is a tracing tool for Rich Internet Applications (RIAs) [1]. DynaRIA is similar to FireDetective in the sense that it records JavaScript traces and visualizes them. However, it does not record any server-side information.

9 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 and an in-depth field user study to answer our two research questions:

RQ1 *Which strategies do web developers currently use when trying to understand Ajax applications?* In the user study we gauged how the user study participants work with traditional web development tools. We witnessed that participants mainly use a bottom-up approach, and heavily rely on text search. This strategy is *ad hoc* and problematic for understanding Ajax applications, of which the logic is spread over

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

the client and server-side. It is our argument that tool support can improve on this situation.

RQ2 *Can dynamic analysis be used to improve program understanding for Ajax applications?* Participants of the user study indicated that FireDetective – which uses dynamic analysis, in particular trace 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.
During the field user study that we performed with two experienced Ajax web developers, we witnessed a hypothesis-driven top-down understanding strategy. When understanding Ajax applications, the developers found that visualizing the client-side interactions was actually the most beneficial part of FireDetective tool. On some occasions, they also investigated the server-side information that FireDetective provides, and they appreciated the fact that they could see all this information integrated into a single tool. Finally, the two experienced developers also noted that they feel that while they are not sure whether FireDetective would save them time when trying to understand Ajax applications, they felt that FireDetective does make them more confident in their understanding.

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

- We have designed and implemented 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, in particular the trace analysis capabilities of FireDetective, can improve their understanding.
- We have carried out a field user study with two experienced Ajax developers that gave us additional insight into how experienced Ajax developers use FireDetective for understanding complex web applications.

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.

In order to strengthen our evaluation, it is our aim to pursue two distinct routes, namely a *longitudinal study* and a *controlled experiment.* The longitudinal study would explore the long term effects of using FireDetective in a real web development environment, while the controlled experiment would give us additional insight into the actual effectiveness of FireDetective in terms of improvements in speed and correctness of maintenance tasks performed with or without FireDetective, similar to [12].

Additionally, both experiments also gave us input for a number of technical challenges ahead. Some of those challenges are:

- Incorporating some FireBug features into FireDetective, e.g., the inspection of values of parameters and return values.
- Extending the server tracing facilities to encompass other platforms besides Java EE.
- Creating a client-side plug-in for the WebKit[17] browser development platform, which would also enable the investigation of Ajax web applications developed for mobile platforms such as iOS or Android.

In a somewhat different direction for future work, it is our aim to investigate whether understanding rich client GUI applications suffers from the same difficulties as understanding Ajax web applications. Furthermore, if our assumption of seeing the same difficulties in that domain is true, we might be able to reuse some of the ideas of FireDetective to also help developers understand these rich client GUI applications.

Acknowledgements

We would like to thank Ian Bull for providing valuable comments on earlier versions of this paper. Additionally, our gratitude goes out to all volunteers that participated in our user study. We also want to thank Johan den Haan, Michiel Kalkman and Michel Weststrate from Mendix for enabling the field user study.

This work has been sponsored by the Center for Dependable ICT (CeDICT), an initiative of NIRICT, the Netherlands Institute for Research on ICT.

[17]http://webkit.org/

References

[1] Amalfitano, D., Fasolino, A.R., Polcaro, A., Tramontana, P.: Dynaria: A tool for ajax web application comprehension. In: Proceedings of the International Conference on Program Comprehension (ICPC), pp. 46–47. IEEE Computer Society (2010)

[2] Antoniol, G., Di Penta, M., Zazzara, M.: Understanding web applications through dynamic analysis. In: Proceedings of the International Workshop on Program Comprehension (IWPC), pp. 120–129. IEEE Computer Society (2004)

[3] Babbie, E.: The practice of social research, 11th edn. Wadsworth Belmont (2007)

[4] Ball, T.: The concept of dynamic analysis. In: Proceedings of the 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE), pp. 216–234. Springer-Verlag (1999)

[5] Bennett, C., Myers, D., Storey, M.A., German, D.M., Ouellet, D., Salois, M., Charland, P.: A survey and evaluation of tool features for understanding reverse-engineered sequence diagrams. Journal of Software Maintenance and Evolution: Research and Practice **20**(4), 291–315 (2008)

[6] Campbell, D., Stanley, J., Gage, N.: Experimental and quasi-experimental designs for research. Rand McNally Chicago (1963)

[7] Chan, A., Holmes, R., Murphy, G.C., Ying, A.T.T.: Scaling an object-oriented system execution visualizer through sampling. In: Proceedings of the International Workshop on Program Comprehension (IWPC), pp. 237–244. IEEE Computer Society (2003)

[8] Corbi, T.: Program understanding: Challenge for the 1990s. IBM Systems Journal **28**(2), 294–306 (1989)

[9] Cornelissen, B., van Deursen, A., Moonen, L., Zaidman, A.: Visualizing testsuites to aid in software understanding. In: Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR), pp. 213–222. IEEE Computer Society (2007)

[10] Cornelissen, B., Holten, D., Zaidman, A., Moonen, L., van Wijk, J.J., van Deursen, A.: Understanding execution traces using massive sequence and circular bundle views. In: Proceedings of the 15th International Conference on Program Comprehension (ICPC), pp. 49–58. IEEE Computer Society (2007)

[11] Cornelissen, B., Moonen, L., Zaidman, A.: An assessment methodology for trace reduction techniques. In: Proceedings of the International Conference on Software Maintenance (ICSM), pp. 107–116. IEEE Computer Society (2008)

[12] Cornelissen, B., Zaidman, A., van Deursen, A.: A controlled experiment for program comprehension through trace visualization. IEEE Transactions on Software Engineering **37**(3), 341–355 (2011)

[13] Cornelissen, B., Zaidman, A., van Deursen, A., Moonen, L., Koschke, R.: A systematic survey of program comprehension through dynamic analysis. IEEE Transactions on Software Engineering **35**(5), 684–702 (2009)

[14] Cornelissen, B., Zaidman, A., van Deursen, A., Van Rompaey, B.: Trace visualization for program comprehension: a controlled experiment. In: Proceedings of the International Conference on Program Comprehension (ICPC), pp. 100–109. IEEE Computer Society (2009)

[15] Cornelissen, B., Zaidman, A., Holten, D., Moonen, L., van Deursen, A., van Wijk, J.J.: Execution trace analysis through massive sequence and circular bundle views. Journal of Systems and Software **81**(12), 2252–2268 (2008)

[16] De Pauw, W., Lei, M., Pring, E., Villard, L., Arnold, M., Morar, J.F.: Web services navigator: visualizing the execution of web services. IBM Systems Journal **44**(4), 821–845 (2005)

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

[18] Ducasse, S., Lanza, M., Bertuli, R.: High-level polymetric views of condensed run-time information. In: Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR), pp. 309–318. IEEE Computer Society (2004)

[19] Dugerdil, P.: Using trace sampling techniques to identify dynamic clusters of classes. In: Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research, pp. 306–314. ACM (2007)

[20] Garrett, J.J.: Ajax: A new approach to web applications (2005). `http://www.adaptivepath.com/ideas/essays/archives/000385.php`, retrieved on December 14th, 2010.

[21] Hamou-Lhadj, A., Lethbridge, T.C.: Techniques for reducing the complexity of object-oriented execution traces. In: Proceedings of the 2nd IEEE International Workshop on Visualizing Software for Understanding and Analysis, pp. 35–40 (2003)

[22] Hamou-Lhadj, A., Lethbridge, T.C.: Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. In: Proceedings of the International Conference on Program Comprehension (ICPC), pp. 181–190. IEEE Computer Society (2006)

[23] Hassan, A.E., Holt, R.C.: Architecture recovery of web applications. In: Proceedings of the International Conference on Software Engineering (ICSE), pp. 349–359. ACM (2002)

[24] Holtzblatt, K., Jones, S.: Conducting and analyzing a contextual interview (excerpt). In: R.M. Baecker, J. Grudin, W.A.S. Buxton, S. Greenberg (eds.) Human-computer interaction, pp. 241–253. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1995)

[25] Kuhn, A., Greevy, O.: Exploiting the analogy between traces and signal processing. In: Proceedings of the International Conference on Software Maintenance (ICSM), pp. 320–329. IEEE Computer Society (2006)

[26] Lerner, R.: At the forge: Firebug. Linux Journal **2007**, 8– (2007). URL `http://portal.acm.org/citation.cfm?id=1243931.1243939`

[27] Li, P., Wohlstadter, E.: Script InSight: Using models to explore JavaScript code from the browser view. In: Proceedings of the International Conference on Web Engineering (ICWE), pp. 260–274. Springer (2009)

[28] Matthijssen, N.: Understanding Ajax applications by using trace analysis. Master's thesis, Delft University of Technology (2010). URL `http://repository.tudelft.nl/assets/uuid:6afeebc9-b574-453d-ac21-5682f57686bc/MScThesisNickMatthijssen.pdf`

[29] Matthijssen, N., Zaidman, A.: Firedetective: Understanding ajax client/server interactions. In: Proceedings of the International Conference on Software Engineering (ICSE), pp. 998–1000. ACM (2011)

[30] Matthijssen, N., Zaidman, A., Storey, M.A., Bull, I., van Deursen, A.: Connecting traces: Understanding client-server interactions in ajax applications. In: Proceedings of the International Conference on Program Comprehension (ICPC), pp. 216–225. IEEE Computer Society (2010)

[31] von Mayrhauser, A., Vans, A.M.: Program comprehension during software maintenance and evolution. IEEE Computer **28**(8), 44–55 (1995)

[32] Mesbah, A., van Deursen, A.: A component- and push-based architectural style for ajax applications. Journal of Systems and Software **81**(12), 2194–2209 (2008)

[33] Myers, D., Storey, M.A., Salois, M.: Utilizing debug information to compact loops in large program traces. In: Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR), pp. 41–50. IEEE Computer Society (2010)

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

[35] Pacione, M., Roper, M., Wood, M.: A novel software visualisation model to support software comprehension. In: Proceedings of the Working Conference on Reverse Engineering (WCRE), pp. 70–79. IEEE Computer Society (2004)

[36] Pauw, W.D., Lorenz, D., Vlissides, J., Wegman, M.: Execution patterns in object-oriented visualization. In: Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems, pp. 219–234. USENIX Association (1998)

[37] Pennington, N.: Stimulus structures and mental representations in expert comprehension of computer programs. Cognitive Psychology **19**(3), 295–341 (1987)

[38] Reiss, S.P.: Visualizing java in action. In: Proceedings of the Symposium on Software Visualization (SoftVis), pp. 57–65. ACM (2003)

[39] Reiss, S.P., Renieris, M.: Encoding program executions. In: Proceedings of the 23rd International Conference on Software Engineering (ICSE), pp. 221–230. IEEE Computer Society (2001)

[40] Ricca, F., Tonella, P.: Analysis and testing of web applications. In: Proceedings of the International Conference on Software Engineering (ICSE), pp. 25–34. IEEE Computer Society (2001)

[41] Runeson, P., Höst, M.: Guidelines for conducting and reporting case study research in software engineering. Empirical Software Engineering **14**(2), 131–164 (2009)

[42] Shneiderman, B.: The eyes have it: A task by data type taxonomy for information visualizations. In: Proceedings of the Symposium on Visual Languages (VL), pp. 336–343. IEEE Computer Society (1996)

[43] Sillito, J., Murphy, G.C., De Volder, K.: Questions programmers ask during software evolution tasks. In: Proceedings of the International Symposium on Foundations of Software Engineering (FSE), pp. 23–34. ACM (2006)

[44] Sim, S., Holt, R.: The ramp-up problem in software projects: a case study of how software immigrants naturalize. In: Proceedings of the International Conference on Software Engineering (ICSE), pp. 361–370. IEEE Computer Society (1998)

[45] Systä, T., Koskimies, K., Müller, H.: Shimba - an environment for reverse engineering java software systems. Software - Practice & Experience **31**(4), 371–394 (2001)

[46] Tonella, P., Ricca, F.: Dynamic model extraction and statistical analysis of web applications. In: Proceedings of the International Workshop on Web Site Evolution (WSE), pp. 43–52. IEEE Computer Society (2002)

[47] Wilde, N., Scully, M.C.: Software reconnaissance: mapping program features to code. Journal of Software Maintenance: Research and Practice **7**(1), 49–62 (1995)

[48] Yin, R.K.: Case Study Research: Design and Methods, 3 edition. Sage Publications (2002)

[49] Zaidman, A., Adams, B., De Schutter, K., Demeyer, S., Hoffman, G., De Ruyck, B.: Regaining lost knowledge through dynamic analysis and aspect orientation — an industrial experience report. In: Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR), pp. 91–102. IEEE Computer Society (2006)

[50] Zaidman, A., Calders, T., Demeyer, S., Paredaens, J.: Applying webmining techniques to execution traces to support the program comprehension process. In: Proceedings of the 9th European Conference on Software Maintenance and Reengineering (CSMR), pp. 134–142. IEEE Computer Society (2005)

[51] Zaidman, A., Demeyer, S.: Managing trace data volume through a heuristical clustering process based on event execution frequency. In: Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR), pp. 329–338. IEEE Computer Society (2004)

[52] Zaidman, A., Demeyer, S.: Automatic identification of key classes in a software system using webmining techniques. Journal of Software Maintenance and Evolution: Research and Practice (JSME) **20**(6), 387–417 (2008)

[53] Zaidman, A., Du Bois, B., Demeyer, S.: How webmining and coupling metrics improve early program compehension. In: Proceedings of the 14th International Conference on Program Comprehension (ICPC), pp. 74–78. IEEE Computer Society (2006)