Understanding Ajax Applications by using Trace Analysis

Master's thesis

Nick Matthijssen

Understanding Ajax Applications by using Trace Analysis

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Nick Matthijssen
born in Breda, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, TU Delft
Delft, the Netherlands
swerl.tudelft.nl

CHISEL Group
Department of Computer Science
University of Victoria
Victoria, BC, Canada
www.thechiselgroup.com

Understanding Ajax Applications by using Trace Analysis

Author: Nick Matthijssen
Student id: 1330195
Email: n.a.matthijssen@student.tudelft.nl

Abstract

Ajax is an umbrella term for a set of technologies that allows web developers to create highly interactive web applications. Ajax applications are complex; they consist of multiple heterogeneous artifacts which are combined in a highly dynamic fashion. This complexity makes Ajax applications hard to understand, and thus to maintain. For this reason, we have created FireDetective, a tool that uses dynamic analysis at both the client (browser) *and* server side to facilitate the understanding of Ajax applications. Using an exploratory pre-experimental user study, we see that web developers encounter problems when understanding Ajax applications. We also find preliminary evidence that the FireDetective tool allows web developers to understand Ajax applications more effectively, more efficiently and with more confidence. We investigate which techniques and features contributed to this result, and use observations made during the user study to identify opportunities for future work.

Thesis Committee:

Chair:	Prof. Dr. Arie van Deursen, TU Delft
University supervisor:	Dr. Andy Zaidman, TU Delft
External supervisors:	Prof. Dr. Margaret-Anne Storey, University of Victoria
	Dr. Ian Bull, University of Victoria
Committee Member:	Prof. Dr. Geert-Jan Houben, TU Delft

Preface

This thesis represents the end result of my Master's project. When I look back at the start of the project roughly a year and a half ago, I'm truly proud of what I learned and achieved over this period. This, however, would not have been possible without the help of many people.

First of all, I would like to thank all participants that took part in my user study. Setting up the study, getting it approved and recruiting participants takes quite a bit of time, but after that, running the sessions and seeing all kinds of interesting data emerge is just amazing.

Next, my project could not have succeeded without my supervisors and their continuous involvement and great feedback. My thanks go out to Andy Zaidman, for his motivation when things did not go as quickly as planned and involvement during the whole project, from finding a research topic to writing the final chapter of this thesis; to Peggy Storey and Ian Bull, for teaching me about doing (empirical) research, helping me greatly with my research and the design of my user study, and for making me feel very at home at the University of Victoria; and finally, to Arie van Deursen, for enabling for me to go to Canada and his involvement in choosing a direction for the project.

Finally, I want to thank all members of the CHISEL group that I had the chance of being a part of for one year, where, from day one, I felt very welcome. The enthusiasm and motivation within the group is fantastic, and I love how easy it is to get feedback on ideas and projects. (DesignFests are amazing!) My thanks go out to my office-mate Sean for helping me with directing my research and setting up the empirical study, Del for helping me out with tracing Java code and the discussions on dynamic analysis, Lars and Christoph for thinking along about my research questions and Trish for her help with filing all the paperwork for the study, but really, I want to thank all group members, since they all contributed to my research by giving good feedback and asking difficult questions (the best kind of question! :-)

Nick Matthijssen
Delft, the Netherlands
April 2, 2010

Table of contents

List of figures

Chapter 1

Introduction

One of the important questions to ask in research is: "So what?" – "Why is this research important?" This chapter defines and motivates the problem that we address and it presents a research plan for doing so.

1.1 Problem statement

Over the past decade, web development has changed its focus from static web sites to rich and highly interactive web applications. The most important technology enabling this shift is Ajax. Ajax is an umbrella term for existing techniques such as JavaScript, DOM manipulation and the XMLHttpRequest object. Ajax is popular: since the term was coined in 2005 [30], 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" [40]. A good example of an Ajax-enabled web application is Gmail, which uses Ajax technologies to update only a part of the page when you open an email conversation and to suggest email addresses of recent correspondents as you type.

Unfortunately, Ajax also makes developing for the web more complex. Classical web applications are based on a multi page interface model, in which interactions are based on a page-sequence paradigm [40]. Ajax changes this by allowing requests to be made after a page has been loaded and by allowing JavaScript code to update parts of the page in the browser, without forcing a full page update.

Before Ajax existed, Hassan and Holt already noted that "Web applications are the legacy software of the future" and "Maintaining such systems is problematic" [34]. One can imagine that the complexities of Ajax will certainly not improve this situation. Given the growing number of Ajax-enabled web applications, we need ways to support web developers in maintaining these applications.

Maintenance starts with understanding. A developer first needs to understand the relevant parts of a system before he or she can make 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 [14]. The importance of understanding is underlined by the fact that entire conferences have been devoted to the topic of understanding, for example, the International Conference on Program

Comprehension. A variety of papers have been published about how to support the process of understanding for web applications in particular.

However, papers focusing on program understanding specifically for Ajax applications are scarce (e.g. [19]). This is where this thesis aims to contribute. It investigates the strategies that web developers use when understanding an Ajax application and how program understanding techniques, specifically from the area of trace analysis, can be applied to the domain of Ajax applications. By doing so, we hope to better support web developers with understanding Ajax applications.

1.2 Research design

Web developers make up the target population that we investigate in this thesis. Before we look at improving program understanding for Ajax applications, we would like to understand more about how web developers currently go about understanding Ajax applications. Therefore, our first research question is:

- **RQ1: Which strategies do web developers currently use when they try to understand Ajax applications?**

Our initial approach to answering this question is introspection, i.e. we draw from our own web development experience and look at the strategies that *we* follow when trying to understand an Ajax application. This is described in the first section of chapter 3. However, we realize that this might lead to a subjective answer to the research question. Moreover, the answer is also most likely to be incomplete: other developers might use other strategies than the ones identified by us. Therefore, we also use a more thorough approach for investigating this question which is discussed later on in this section.

The next step in our research process is to consider the strategies that we identified, and look for problems with these strategies. We then take a subset of these problems, and transform them into a tool design. This process, from strategies to a fully detailed tool design, is described in chapter 3. The tool that we design uses techniques from the area of trace analysis, which in turn belongs to the area of dynamic analysis. Our choice for using trace analysis is motivated by the problems that we found; this is discussed in detail in the chapter.

Our second research question considers the implementation of the aforementioned tool design.

- **RQ2: Is it feasible to build a tool in which trace analysis techniques are applied to the domain of Ajax applications?**

We attempt to answer this question by actually implementing the tool according to the design. This is the topic of chapter 4. The chapter describes technical barriers and challenges that need to be overcome to create a tool that (a) works and (b) is reasonably fast to be of practical use to web developers. Assuming it is feasible to create such a tool, we can leverage it to find insights into our third, and final research question.

- **RQ3: Can we use trace analysis to improve program understanding for Ajax applications?**

To answer this question we conduct an exploratory empirical study. If we can find evidence that trace analysis is indeed useful for improving understanding of Ajax applications, we can also investigate the factors that contribute to the improved understanding process. This knowledge is useful, because it can aid in decreasing the effort spent on software maintenance for Ajax applications.

Chapter 5 describes the design of the study. Chapter 6 describes our findings. The idea behind our study is to give participants (who are web developers) a set of typical program understanding tasks, and then observe how they accomplish these tasks. We split the study up into two parts. During the first part participants use standard web development tools, whereas in the second part they use our tool. The purpose of the first part is to help answer RQ1: which strategies do developers use? The purpose of the second part is to provide insight into RQ3: can trace analysis improve program understanding? Finally, both parts are expected to be useful for identifying future opportunities for improving program understanding for Ajax applications.

Empirical evaluations in software engineering are quite rare. Sjøberg et al. estimate that only about 20% of all software engineering papers of the past decade report empirical evaluations [52]. If we focus on program comprehension, and specifically, program comprehension through dynamic analysis (of which trace analysis is a subarea), we find that the number is even lower. A survey by Cornelissen et al. [19] shows that only 11 out of 176 reviewed papers carry out an industrial strength evaluation. Furthermore, only 6 out of 176 papers carry out an empirical evaluation involving human subjects. (They do note that three of those were conducted recently, and that this type of evaluation could become more common.) Despite these low numbers, empirical evaluations are important: Sjøberg et al. state that "Software engineering research studies the real world phenomena of software engineering", and "sciences that study real-world phenomena [...] of necessity use empirical methods [...]. Hence, if software engineering research is to be scientific, it too must use empirical methods." [51]

Chapter 7 gives an overview of other efforts concerned with making Ajax applications easier to understand. Finally, chapter 8 concludes our work and lists various ways in which this research can be continued.

Chapter 2

Background

Several topics are brought together in this thesis, namely: program understanding, dynamic analysis and Ajax applications. This chapter gives an overview of each of these topics by describing relevant previous research efforts.

2.1 Program understanding

Program understanding or program comprehension is the research field that studies how software developers understand programs.

In order to correctly modify software, it is critical for developers to be able to obtain and maintain a good understanding of the (part of the) software system they are working on. However, software systems are generally complex and may be large. Moreover, the complexity of a software system may increase over time, along with the ever-changing environment in which it operates [6]. Cross-cutting concerns [10], single features that should really be in one place in a code base but which are scattered throughout, are just one form through which software complexity manifests itself. Developers usually employ abstractions to deal with complexity and size, but these only partly suffice [54]. Therefore, it comes as no surprise that program understanding is known to be costly. As we already mentioned in the introduction, as much as 50% of the time spent on a maintenance task is spent on understanding [14].

2.1.1 Theories and cognitive models

There exist a number of theories on program understanding, which try to shed light on questions such as: how do developers go about understanding programs? and: which strategies do they follow?

A commonly used definition of program understanding is: program understanding is the act of constructing mental models of software systems. Von Mayrhauser and Vans define a mental model as "an internal, working representation of the software under consideration" [58]. Mental models may contain facts from different levels of abstraction, ranging from low-level facts (e.g. bits of source code) to high-level facts (e.g. architectural overviews). Building mental models is an incremental process – they are constructed bit by bit, as the developer gains more insight into a software system.

There exist different cognitive models which describe how developers build up mental models. These cognitive models are general in the sense that they abstract away several details, such as developer, software system and task characteristics [55]. No two developers are the same, and different developers have different backgrounds and varying skill levels. Next, software systems come in all shapes and sizes: some are easy to understand, while others may be hard to gain insight into. Even the choice of programming language can have influence on how difficult it can be to understand a software system [45]. Finally, program comprehension is never an end-goal in itself: the knowledge obtained by program understanding is always used in subsequent activities, for example maintenance tasks. The type of these tasks affects how a developer goes about understanding a software system.

Despite these generalizations, cognitive models are commonly used to describe the process of understanding a software system. We distinguish three of them.

- **Top-down.** A top-down understanding approach starts with a hypothesis concerning the global nature of a software system [11]. This hypothesis is gradually refined into more concrete hypotheses that can be verified. During this process, developers use programming plans (they look for recurring patterns in software they are familiar with), they use beacons (which are sets of features that indicate a certain hypothesis is correct, such as the `swap` statement in a sorting routine) and rules of discourse (common programming conventions, such as code standards).

- **Bottom-up.** In this approach, developers start the understanding process by examining source code and lower-level details. They construct higher-level facts about the system by chunking and concept assignment [45]. Chunking is grouping low-level facts into higher-level facts. Concept assignment [8] is relating facts from the real world and assigning them to their counterparts in the source code.

- **Combinations of top-down and bottom-up.** According to Letovsky, developers frequently alternate between top-down and bottom-up approaches [37]. Von Mayrhauser and Vans go even further and state that developers actually use them at the same time, to simultaneously construct different types of knowledge at different abstractions levels [58].

For exploring how unfamiliar software systems work, bottom-up approaches are more suitable than top-down approaches; moreover, developers who are understanding a software system in a bottom-up fashion first look at the control-flow, or sequence of operations in a system [45]. They traverse the hierarchy from bottom to top, group lower-level abstractions into higher-level ones, and construct what Pennington calls a *program model*. Once this model is complete, they construct the *situation model*, by looking at data-flow and functional abstractions: the goal-based decomposition of the program. Both models are cross-referenced to further increase understanding of the system.

Developers may follow either a *systematic* or *as-needed* approach [39, 53]. The systematic approach corresponds to following a program's control flow from start to finish and reading and understanding every part of the software system under study.

On the contrary, the as-needed strategy involves understanding only the parts that are necessary at a certain time, for a certain task. An advantage of the systematic approach is that it is thorough and therefore more likely to be correct. Indeed, in a user study where participants are asked to enhance a piece of software, Soloway *et al.* discovered that a systematic approach invariably led their participants to making correct enhancements, whereas the as-needed approach only led to half of the participants making the correct enhancement [53]. However, for a lot of real-world software development tasks it is not necessary to understand the whole system, and the as-needed approach may be more efficient in those cases.

2.1.2 Cognitive support

Program understanding theories can serve as an excellent starting point improving the program understanding process. Indeed, once we know how developers go about understanding, and which steps they find difficult, we can use this information to guide the design of tools that are able to support developers.

Storey *et al.* propose a list of design elements that we should consider when building tools [55]. They divide them into two main categories.

- **Improve program comprehension.** We should support the cognitive models that developers follow, i.e. the ways developers construct mental models of the system. For top-down comprehension, we need to support goal-driven comprehension and provide overviews of the system at different levels of abstractions. For bottom-up comprehension we need to indicate relations between elements and allow developers to follow these relations. Moreover, tools should be able to show how cross-cutting concerns are spread throughout the code, and must offer ways for a developer to keep information about abstractions after he or she has reconstructed them. For combinations of top-down and bottom-up, it is important that a tool allows the construction of multiple mental models and allows cross-referencing between them [55].

- **Reduce cognitive overhead.** Due to the complexity and size of a software system it is easy for a developer to get overwhelmed by the large amount of available information. Therefore, tools that can reduce the cognitive load on a developer can be very helpful. Tools can do this by allowing easy navigation of the software system's information space. Tools may facilitate *directional navigation* (a developer looks for specific information) and *arbitrary navigation* (a developer is exploring the system). When a developer is navigating, it is important that enough orientation cues and navigation context are provided to prevent the developer becoming disoriented. Recent user studies confirm this [20]. Tools should indicate the developer's current focus, how he got there, and should also offer suggestions for reaching new nodes. Finally, tool user interfaces should be easy to use and visual representations software and layouts of diagrams should be carefully chosen [55].

2.2 Dynamic analysis

Ball [4] defines dynamic analysis as "the analysis of the properties of a running system". Note that this is a broad definition; many different aspects of the run-time state may be analyzed.

2.2.1 Strengths and weaknesses

Dynamic analysis' counterpart is static analysis. Static analysis refers to the analysis of static artifacts of a software system (e.g. source code or architecture diagrams). Static and dynamic analysis are complementary and have different advantages and disadvantages.

A strength of dynamic analysis is that it allows developers to peek "under to hood" of a program, and see what is going on at run-time, instead of trying to predict what is going to happen. Also, dynamic analysis can reveal relations that static analysis cannot show. Consider relationships established by late (run-time) binding. These are very common in dynamically typed languages, but also occur in static languages, in the form of polymorphism in object-oriented languages and event-like constructs as in C#, for example. These kinds of relationships cannot be revealed by static analysis because they are formed only when a program runs. Finally, dynamic analysis allows to try out different program inputs and immediately link them to internal behavior and program outputs, without having to understand the entire program, i.e. it enables a *goal-oriented strategy* [19].

A weakness of dynamic analysis is its incompleteness. The quality of the analysis depends on how the execution scenarios are chosen. When one is unfamiliar with a software system, it can be hard to determine what exactly constitutes a good execution scenario. Next, dynamically analyzing a system might cause the system to behave differently from the way it normally does. The program that is being analyzed generally runs slower because of the instrumentation that is required to record the run-time state data. For example, this may then cause the OS' scheduler to schedule threads differently, which may uncover never-seen-before concurrency issues, or may cause existing issues to mysteriously disappear. This problem is called the *probe effect* or *observer effect* [1]. Finally, dynamic analysis may generate large amounts of data. For instance, Reiss and Renieris report on an experiment where one gigabyte of information was generated for every 2 seconds of executed C/C++ code or for every 10 seconds of executed Java code [47]. When dealing with larger systems and larger execution scenarios these scalability issues become even more apparent.

2.2.2 Extracting run-time properties

Before we look at specific dynamic analysis techniques, we look at the practical side of dynamic analysis: before run-time properties of a system can be analyzed, we first need to capture them in some way. Different approaches may be used.

- **Debugger interfaces.** Quite a few programming language platforms offer a debugger, profiling or tool interface. Typically, via such an interface notifications

for all kinds of run-time events can be received, such as (JIT[1]) compilation of functions, control flow entering or leaving a function, exceptions being thrown, memory (de)allocations, etc. These events can then be tracked and captured. Some interfaces may also allow to suspend and resume execution, and allow to inspect the run-time state of the program. Advantages of the approach are that notifications can be turned on and off at run-time, and the original source code of the program does not have to be modified. A disadvantage is that the approach may be slow, especially when many notifications are being generated, e.g. for every function call. Some examples of interfaces are the JVMTI (Java)[2], Firefox' `jsdIDebuggerService` component (JavaScript)[3] and the `bdb` module (Python)[4].

- **Code instrumentation.** This approach is based on directly inserting instrumentation statements into the code of the program before it runs. Instrumentation can be inserted at source, byte or machine code level. Aspect-oriented programming [35] provides a relatively easy way to do basic code instrumentation. Alternatively, code can be transformed via other approaches, e.g. a library or a custom implementation. Because statements are directly inserted into the code, code instrumentation is typically faster than debugger interfaces. However, the approach is less flexible. Instrumentation cannot be turned on and off at run-time. The approach does not work for dynamically generated code. Moreover, when code is run in a limited security context (e.g. a web browser) this forces its security restrictions upon the instrumentation code as well, which might cause problems (e.g. it might prevent the instrumentation code from writing to a file).

- **Combined approaches.** The approaches may be combined to counter some of their disadvantages. For instance, function compilation notifications may be used to instrument code as needed, when the program is running. In fact, some tool interfaces use this approach "under the hood" to speed up the notifications[5]. This approach also enables dynamically generated code to be instrumented. Also, Tanter *et al.* [29] suggest a technique called *dynamic behavioral reflection* to add and also remove instrumentation at run-time.

The granularity of the measurements can be adjusted. Measurements can be taken at the statement level (after every statement), call level (to show function calls), class level (to show interactions between classes) and architectural level (to show interactions between architectural components), for example. There is a trade-off: higher granularities produce more detailed information, but also generate larger traces which are thus harder to handle. The approaches discussed above may somewhat limit this choice: for example, Aspect-oriented programming and debugger interfaces are not

[1] Just-in-time.

[2] Java Virtual Machine Tool Interface. See `http://java.sun.com/javase/6/docs/technotes/guides/jvmti/`.

[3] See `http://www.oxymoronical.com/experiments/apidocs/interface/jsdIDebuggerService`.

[4] See `http://docs.python.org/py3k/library/debug.html`.

[5] An example is the .NET profiling API. See `http://msdn.microsoft.com/en-us/library/ms404386(VS.100).aspx`.

usually suited for statement level instrumentation, and can only work with call level instrumentation and up.

The recording process produces a program trace, or trace for short. A trace is a collection of run-time states that a program went through. Execution traces are the most commonly occurring variety: they contain information about a program's control flow. Other types of traces are traces that contain memory events (allocation, deallocation) or object lifelines (create, read, update, delete of individual object instances). Of, course, it is also possible to combine different types of traces, e.g. execution traces that are enriched with parameter values and perhaps variable values.

2.2.3 Trace analysis

Cornelissen *et al.* [19] divide the field of dynamic analysis into a number of subfields, namely: *trace analysis*, *design and architecture recovery*, the study of *behavioral aspects* and *feature analysis*. The first one, trace analysis, is type of dynamic analysis that we focus on in this research. For a description of the other subfields we refer to [19].

Trace analysis concerns itself with visualizing traces to provide insight into the workings of the program. Since traces may quickly grow to massive proportions, we need ways to deal with their size. We consider two common ways to do so: trace reduction and trace visualization, which are often combined.

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 [15].

- **Ad-hoc methods.** Straightforward ways to reduce the size of traces are: defining start and end points within the code, extracting time slices from a trace, and sampling of traces [13, 27]. These approaches do not consider the contents of the trace.

- **Language-based filtering.** 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 [16, 31]. We can also filter elements of the program or its libraries, i.e. calls to specific components, classes, methods, etc.

- **Metric-based filtering.** Metrics 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 [23] or below a specific depth [16]. 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 [33].

- **Trace summarization.** The idea behind trace summarization is to find patterns within traces that can be compacted. 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" [23]. Examples are methods based on string matching [56] run-length encoding or grammars [47], techniques that are borrowed from the signal processing field [36] and approaches that use information from source code [41]. 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 [23]. 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) [23].

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 [7].

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 [46] 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 [26]. For example, instead of squares in a grid, they use nodes in a graph to represent program elements. Next, Richner and Ducasse propose *collaboration diagrams*, generalized versions of sequence diagrams in which the time part is abstracted away [49]. Cornelissen *et al.* describe the idea of *circular bundle views*, in which a system's components are shown on the boundary of a circle, and bundles within the circle represent relationships between components [17].

Note these techniques are just a small subset of all approaches that exist. Surveys of trace visualization approaches can be found in [32, 43]. A more recent overview of trace visualization techniques can be found in [18].

2.3 Ajax

Ajax is an umbrella term for a number of existing technologies, as we described in the introduction chapter of this thesis. Ajax can be used to create interactive web applications, or Ajax applications, as we will call them. Examples include well known applications such as Gmail, Google Maps and Facebook. Ajax incorporates technologies such as DOM manipulation, asynchronous data retrieval using XMLHttpRequest, and finally, JavaScript, to bind everything together [30].

2.3.1 How does it work?

How is Ajax different from traditional techniques? Figure 2.1 shows a simplified model of a web application that does not use Ajax technologies. Typically, a browser sends of a request for a page, a web server handles the request and an HTML page is

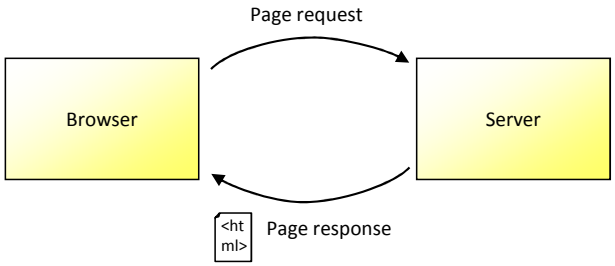


Figure 2.1: Model of a traditional web application (simplified).

sent back to the browser. This HTML page might include references to other content, like style sheets and images, which the browser can fetch from the server by sending a request for each one of them (not shown in figure). After that, no further actions occur until the user interacts with the page. For example, the user can click a link, in which case the process is repeated for the next page.

When Ajax comes into play, things change quite a bit. Figure 2.2 shows a simplified model of an Ajax-enabled web application. The interaction starts off in the same way: the browser sends a page request to the server, which responds with an HTML page. However, this page may include JavaScript code by means of `<script>` tags. These pieces of JavaScript code can then in turn modify the page. They are able to do so by accessing a tree representation of the current page which is called the DOM[6]. Moreover, they may use the `XMLHttpRequest` object to send an asynchronous request to the web server. The server sends back a response, which can be in XML format but need not be: other languages like JSON[7] or HTML are popular choices as well. Upon receiving a response, another piece of JavaScript code can be invoked, which can then update the page via the DOM manipulation mechanism described above. Scripts are also able to set up event handlers for DOM events, such as "the entire page has been loaded" or "the user has clicked an element", and set up timeout handlers (not shown in figure).

Note that the Ajax programming model is quite different from the stand-alone application programming model. For example, a stand-alone Java application has a single-entry point called `main` which determines how control flow through the code. Upon exiting `main`, the application terminates. In contrast, Ajax applications employ a programming model that is highly event-driven. Browser and server may operate concurrently, and control may flow back and forth between browser and server multiple times.

2.3.2 Strengths and weaknesses

The obvious advantage of Ajax over traditional web technologies is that it allows web applications to be much richer and more interactive. However, Ajax applications are also more complex which makes them harder to develop and understand. Ajax applications consist of a collection of heterogeneous artifacts, such as client side scripts,

[6]Document Object Model. See `http://www.w3.org/DOM/`.
[7]JavaScript Object Notation. See `http://json.org/`.

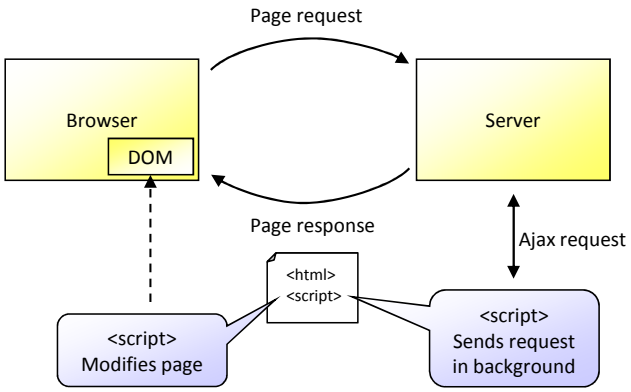


Figure 2.2: Model of an Ajax-enabled web application (simplified).

server side scripts and web templates, which are dependent on each other and all of which contribute to the application. The degree of dynamicity is high: artifacts are often linked dynamically and dynamic programming languages are used.

Ajax is a quite recent technology: the term Ajax was coined less than 5 years ago [30]. As a result, tool support is not as mature as for other programming platforms, but this seems to be improving. Development of Ajax applications is complicated by browser incompatibilities and certain technical problems, e.g. care needs to be taken to make sure that back buttons and bookmarks do not break in Ajax applications. JavaScript frameworks exist that can mitigate these disadvantages: examples include jQuery[8] and Dojo[9].

2.3.3 Server side technologies

Various languages, frameworks and libraries can be used to develop the server side of Ajax applications. While these techniques are, in their application, often not limited to Ajax applications, we discuss some of them here, because they are frequently used in Ajax applications in practice.

A technique that is often used in both non-Ajax web applications and Ajax applications is the dynamic generation of HTML pages. Rather than serving a static HTML file which is the same for every user that requests it, different pages can be served depending on the (user) context. An important key component in getting this to work is the *template engine*. Upon receiving a page request by some client, it takes a web template (written in a language such as PHP, Ruby, JSP, etc.) and executes it (letting the template collect all of the pieces of information that need to be included on the page). Most web template languages allow a mixture of HTML and code. The output is an HTML document that can be served to the client that requested the page.

While the template abstraction makes it easier to use dynamically generated pages, higher-level frameworks exist which allow developers to specify their web applications at a more abstract level. Many enforce the use of the *model-view-controller* design

[8] See `http://jquery.com/`.
[9] See `http://www.dojotoolkit.org/`.

pattern. Examples of frameworks include JSF (built on Java + JSP)[10], CakePHP (built on PHP)[11], Rails (Ruby)[12], Django (Python)[13], etc. These web application frameworks allow developers to define a data model for which default views and controllers can be generated automatically.

[10] See `http://java.sun.com/javaee/javaserverfaces/`.
[11] See `http://cakephp.org/`.
[12] See `http://rubyonrails.org/`.
[13] See `http://www.djangoproject.com/`.

Chapter 3

From strategies to a tool design

Our first research question is: *which strategies do web developers use when they want to understand Ajax applications?* This chapter starts with a brief look at our own web development experience to find an initial answer to this question (later, in chapter 6 the question is investigated in more detail). We also look for problems that are associated with these strategies. We then use the identified problems to guide us in the design of a tool that is aimed at improving understanding of Ajax applications.

3.1 Identifying strategies

This section can be regarded as a mini case study of program understanding strategies of web developers.

Web developers can use a variety of tools for understanding Ajax applications. Tools that we have used ourselves include all kinds of text editors and IDEs (e.g. Eclipse, Visual Studio), mostly for inspecting and modifying code. We also regularly use add-ons and extensions for various browsers, such as the FireBug[1] and Web Developer[2] add-ons for Firefox. We believe such a set of tools to be representative of the tool set of the average web developer. We limit this case study's scope to the understanding of small to medium-sized Ajax applications, since most of our experience concerns these kinds of applications. Moreover, we only consider the process of understanding unfamiliar code, e.g. a new code base of a new application.

In our experience, following the control flow through an application is often an essential part of beginning to understand an application's inner workings. Hence, our understanding strategy frequently consists of picking a starting point (a visible feature, or an interesting class/function) and exploring the code from that point by following "calls" and "called by" relationships. During this process, we mostly rely on browsing through code. In some cases, the IDE offers functionality for jumping to definitions and finding occurrences. At other points, we use text search or briefly scroll through source files in case functions can be found in the same file.

There are several problems with this strategy, specifically when applied to Ajax applications. First of all, Ajax applications consist of a collection of heterogeneous artifacts, such as client side scripts, server side scripts and web templates, which are

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

[2] See `https://addons.mozilla.org/en-US/firefox/addon/60`.

dependent on each other and all of which contribute to the application. This can make following control flow quite challenging. This is further complicated by the high degree of dynamicity of Ajax applications. Links between the aforementioned artifacts are often established at run-time. HTML pages can be dynamically generated and updated, and scripts can be generated and executed on the fly. Finally, the languages themselves are highly dynamic, such as JavaScript and some server side scripting languages such as PHP, Python and Ruby.

Antoniol *et al.* [2] already argued that static analysis is insufficient for web applications. Since Ajax applications are a more dynamic type of web applications, static analysis must be insufficient for Ajax applications as well. Given the aforementioned problems, we can see why this is the case. For example, let's consider dynamically dispatched calls. If we only look at code, say a fragment of JavaScript code consisting of a set of calls, it can be hard and in some cases it can be impossible to find out which functions are actually being called. We can use tricks to mitigate this particular problem, such as (in our experience not uncommon) setting breakpoints in functions that we expect to be called, or inserting a bunch of "print" statements at those locations and see which ones are actually executed. However, these approaches are crude and may take (a lot of) time. The problem of following control flow largely remains.

We mentioned that our strategy often starts with picking a starting point for investigation. A starting point could be an interesting class of function that we come across, or a DOM element, for example. A technique that we often use to map DOM elements to code, is to inspect the DOM element, look at its id, and then search for uses of that id in the code. However, this technique fails when the id is generated dynamically.

Frequently, we find ourselves "diving" into the code and looking for interesting fragments, a process that can become complicated when the application's architecture and design are not perfectly straightforward.

Therefore, we would like to be able to use additional types of starting points. For example, the FireBug add-on displays a list of (Ajax) requests, but relating these requests to relevant code fragments is not possible. Similarly, we can use add-ons to see which DOM events are being fired, but we cannot easily find out which code is being executed as a result of the events firing. Finally, seeing DOM mutations and being able to map them to code would be useful, but is not possible[3].

Summing up, we think picking a starting point and exploring the code by using control flow relationships is a useful strategy when understanding Ajax applications. However, a lot of manual effort is involved which makes the strategy time consuming and error prone. We expect that these negative aspects can be mitigated by supporting the strategy with (better) tools. A tool should be able to cope with a fragmented control flow between heterogeneous artifacts and across machines (i.e. browser and server), and should also focus on enabling a more "top-down" [58] way of exploring, rather than the current "bottom-up" approach.

[3]These observations were made when we started our research, in early 2009. Things have improved since then, and certain new add-ons have appeared. (An example is EventBug, which can show a DOM element's event handlers. See `http://getfirebug.com/releases/eventbug/`.) See chapter 7 for an overview of related work.

3.2 Tool design

The observations in the previous section led us to create a design for a tool that is aimed at improving understanding of Ajax applications. In the following, we outline some of the major design decisions and explain how the tool should work.

The tool that we design uses dynamic analysis. The choice for dynamic analysis is instigated by the high degree of dynamicity in Ajax applications, and the fact that static analysis does not suffice for analyzing them, as described in the previous section. More specifically, the tool uses trace analysis. Our reason for choosing trace analysis is that it is arguably one of the most straightforward (conceptually) applications of dynamic analysis: i.e. recording traces and visualizing them to the tool user.

Since one of our goals is to make control flow easier to follow, the tool records execution traces. This goal is also a deciding factor in choosing the level of detail of the trace. We opt for 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, and providing sufficient information for tool users to follow the control flow. Note that the tool records traces on both the client (i.e. browser) and the server side, to offer tool users a complete picture of an Ajax application.

In the previous section we already noted that control flow is fragmented between heterogeneous resources and machines. Hence, without any further tool requirements we would get a tool that records a collection of separate traces, which the tool user has to piece together manually. This, of course, would be disadvantageous for understanding. Moreover, the tool would also lack additional starting points for exploring an Ajax application.

For these reasons, the tool also records information about higher-level abstractions that are specific to the Ajax/web-domain, such as (Ajax) requests, DOM events, timeouts and intervals, etc. This is a key element of the tool: it enables linking the execution traces with each other and with higher-level abstractions. Incidentally, these abstractions can also be used as starting points for program understanding. The tool presents the network of traces and abstractions to the user in a set of interactive views.

3.2.1 Using abstractions to link traces

The tool uses a number of different abstractions from the Ajax/web-domain to which it links execution traces or calls within execution traces. Figure 3.1 shows a UML model of the abstractions that the tool uses, and reveals how they are linked to each other and to client side (JavaScript) and server side traces. The abstractions are further explained below.

- **Full page requests** occur when a whole page is loaded. We use a full page request to group all requests and JavaScript traces that take place before the next full page request occurs, into a chronological list.

- **(Non-Ajax) requests** are contained within a full page request. They are also associated with the server side trace that was recorded for that particular request.

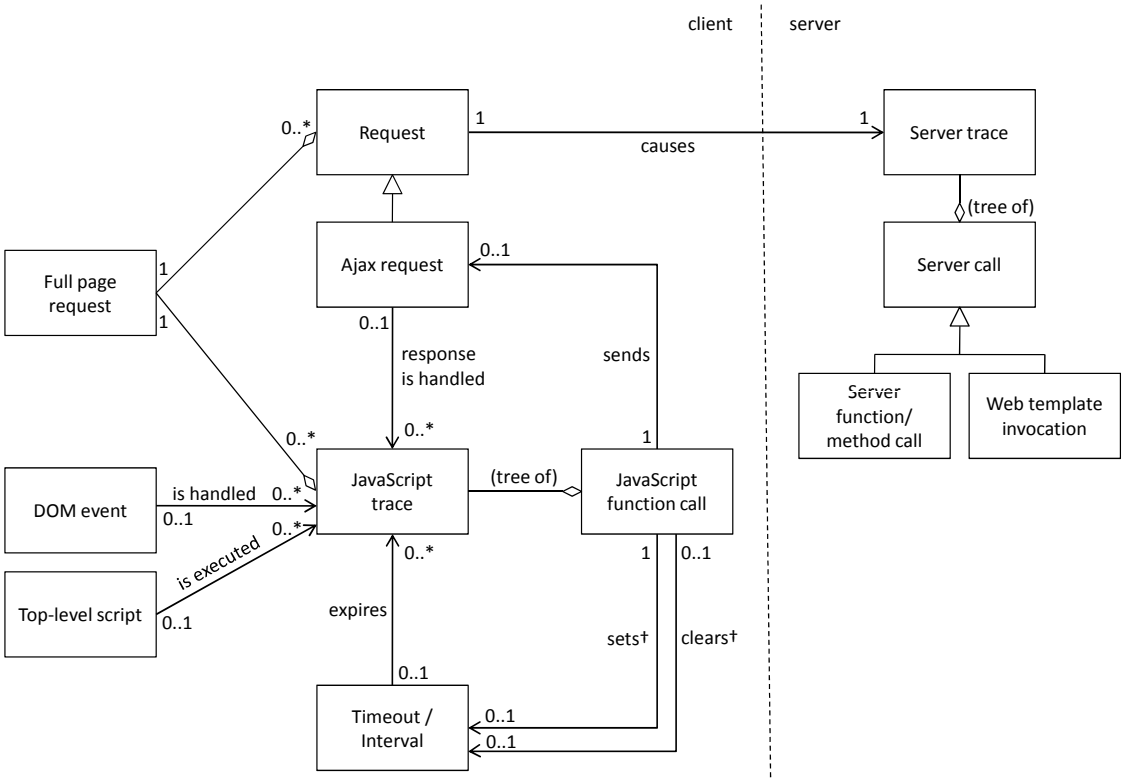


Figure 3.1: UML model of abstractions and traces.

- **Top-level script load invocations** occur when the browser has loaded scripts and executes them. These script loads are linked to the resulting JavaScript trace.
- **DOM events** are events such as "element was clicked" or "page was loaded". They are associated with one or more JavaScript traces that were recorded as a result of event handlers firing for the DOM event in question.
- **Ajax requests**, like other requests, are associated with a single server side trace. They are also linked to the JavaScript call that sent the request and the JavaScript traces that were recorded when handling the response.
- **Timeouts** and **intervals** (in JavaScript) can be set to trigger a timeout handler after a specified time period has elapsed. We link timeouts to the JavaScript traces that were recorded as a result of the timeout handler being invoked, and to the JavaScript calls that started† and stopped† that particular timeout.
- **Web template invocations** are not specific to Ajax, and are used in many web applications. Depending on the back end technologies used in a web application, web templates may be compiled prior to execution. For this reason, they do not always end up in the trace in their original form, and might need to be reconstructed as a part of the call tree.

†In our implementation, these links were only implemented after conducting our empirical study.

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?".

Note that this model is based on our personal observations, and is not meant to be a complete model of all abstractions in the domain of Ajax applications. More abstractions could be added; moreover, the existing abstractions could be linked in different ways, e.g. DOM events could be linked to the JavaScript calls that set up the event handlers for these events. Instead, the model contains the abstractions that we think provide the most value (i.e. the "low hanging fruit") for improving the understanding of Ajax applications. In chapters 6 and 8 we suggest possible additions to the model.

3.2.2 Interactive visualization

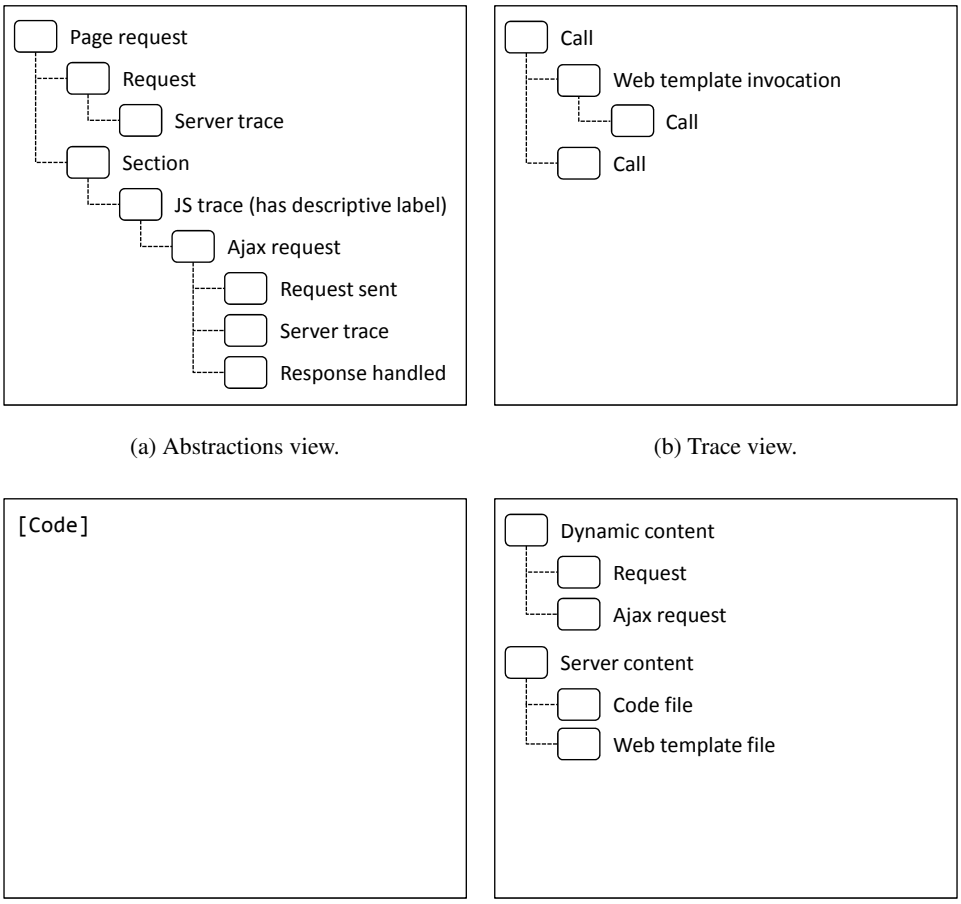
In this section we define how the linked traces and abstractions are visualized. The visualization's design is loosely based on guidelines outlined by Shneiderman [50]: information visualization tools should allow for creating overviews, zooming, filtering, and providing details on demand.

A conceptual outline of the views is shown in Figure 3.2. Three main views are used (a-c), each of which shows a different level of detail. There is also one resources view (d). The views are linked: selecting an element in a particular view updates the other views accordingly.

The abstractions view (a) is a high-level view that shows a tree representation of the aforementioned abstractions (except template invocations). They are organized in a hierarchical fashion, in such a way that the hierarchy roughly matches the way that the abstractions are linked. The view is chronological, i.e. the tree nodes are ordered according to when the abstractions that they represent occurred (with one exception that we discuss shortly).

The top-level nodes in the view represent full page requests. Full page request nodes may be expanded to show non-Ajax requests, which can in turn be expanded to show a single node that represents the server side trace that was generated when the request was handled. Such a node can be selected to view the corresponding trace in the trace view (b). Expansion of full page request nodes may also reveal so-called sections. Sections represent a time slice of the execution of the application that is analyzed, and can be created by the tool user (a feature that we discuss in the next subsection). Section nodes can in turn be expanded to show all JavaScript traces that happened within the section's time window. JavaScript trace nodes can be selected to view the corresponding trace in the trace view. Each of the JavaScript trace nodes is fitted with an informative label to allow the tool user to see the cause of the JavaScript trace (e.g. an Ajax response event, DOM event, top-level script invocation or timeout/interval).

There is one occasion in which a JavaScript trace node can be further expanded: that is when anywhere in the trace, one or more Ajax requests are started. Every child node represents a single Ajax request. An Ajax request node can be expanded to show a collection of child nodes that together represent the life cycle of the Ajax request.



(a) Abstractions view.

(b) Trace view.

(c) Code view.

(d) Resources view.

Figure 3.2: Conceptual outlines of the views that make up the visualization.

The child nodes represent the call that sent the request, the server side trace that was generated as a result of the Ajax request, and JavaScript traces that occurred after the response arrived back in the browser. Note that these last nodes are duplicate nodes that also appear further down in the tree view (possibly contained within a later section). Hence, this is the exception to the chronology of the view. This may confuse users, but we expect that having *a* life cycle view is better than having no such view at all. All child nodes of an Ajax request can be selected to show the corresponding JavaScript calls and traces in the trace view.

The trace view (b) displays one execution trace at a time, as a call tree. Each tree node represents a single call, which can be expanded to show subcalls. When a server trace is displayed, web template invocations may appear in this view as well: they are visualized in the same way as calls. A node can be selected to show and highlight its corresponding source code in the code view (c). The ability to always be able to jump to code is important, as it gives the tool user the opportunity to obtain a fully detailed picture of an Ajax application. Since code does not always reside in files and may be generated on the fly, the tool might need to do some additional bookkeeping to make

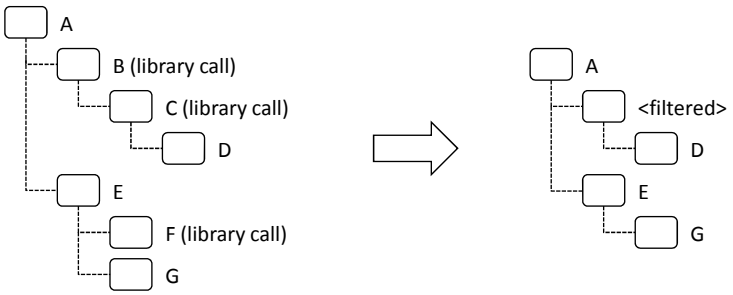


Figure 3.3: Filtering a sample call tree. B and C are replaced with a single dummy node. Note that the call F is not replaced with a dummy node, because there are no non-library calls further down F's branch.

sure all dynamically generated code fragments are captured and accessible within the tool.

The code view (c) is a standard source code view with syntax highlighting. Moreover, tool users can select a block of code (e.g. a function) and select an option named "where is this code called?" to highlight and cycle through invocations of the selected block of code in the abstractions view (a) and trace view (b). This feature allows them to go from low-level code to higher-level entities.

Finally, there is one resources view (d), which shows a tree representation of the resources on the current page. Resources are divided into two categories. The first category contains all requests. Each request (both non-Ajax and Ajax) has a response with content associated with it: this content is dynamically generated on the server[4]. The second category contains all static server web template and code files. Clicking a resource shows its content in the code view. The resource view can also be filtered to only show the resources that were used for a particular full page request, or a single HTTP request, which may greatly reduce the number of files that are shown, and allows a tool user to quickly see which resources are involved.

3.2.3 Barriers to comprehension

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 [15].

The first one is to filter out all of library calls and only keep calls that are specific to the Ajax application that is being analyzed. Both client side libraries and server side libraries are filtered out. In case control flows from the Ajax application into a library, but then back into the application via a callback, a dummy call tree node is inserted in the call tree. This makes the tool user aware of the fact that some calls were filtered out. Figure 3.3 shows the reduction of a small sample call tree.

The second trace reduction mechanism is a time slicing mechanism. By means of a start and stop button users may create time slices of the execution of an Ajax application. Each time slice is then converted into a section, which is shown in the

[4]For FireBug users: these resources correspond to the items shown in FireBug's Net Panel.

abstractions view of the tool. This feature allows the user to find out how a particular interaction with the Ajax application is handled, for example.

A caveat regarding JavaScript tracing is that the language allows a developer to define anonymous functions, a mechanism which is commonly used by web developers. Because many trace visualizations (including this 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 an anonymous function is often assigned to exactly one variable, e.g.: `var f = function(...) { ... }`. Therefore, whenever this is the case, the tool uses the name of the preceding variable to "name" the function. This approach is not always correct in the technical sense: in the example, f could be reassigned another function, later during the execution of the program. However, technical correctness is not the most important quality in this case. Instead, we should try to match the mental model of the user, which this approach is likely to do. Moreover, the approach seems to work well in practice: for example, FireBug currently uses a similar technique (albeit simpler, based on regular expressions) to "name" anonymous functions.

In case an anonymous function definition is *not* preceded by a variable assignment (e.g. the anonymous function it is an argument in a function call), it is simply named "anonymous". Nevertheless, tool users can always simply select the anonymous function call in the trace view and immediately see its definition highlighted in the code view.

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

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

Finally, we already noted the importance of always giving users the option to drill down into code. However, it is actually slightly more complicated than that. In addition to showing code, it is also important to provide the right "code context". For example, consider event handlers that are defined inside HTML code, e.g.: `<a onclick=`[code fragment]`/>`. If this event handler fires, and the tool user inspects the resulting trace, instead of displaying just the [code fragment] the tool should show the code fragment within the context of the HTML output. This way, the tool user can actually understand *where* the code fragment comes from, making it much easier to modify it later on, for example. A similar situation arises when code is "eval"-ed on the fly: e.g.: `eval(`[string fragment]`);`. Rather than showing the [string fragment], the tool should show it within its original context, when possible.

3.2.4 Design constraints

Finally, there are two design constraints that we would like to mention.

The first one concerns real-time analysis: rather than requiring users to start recording, stop recording and open the trace with the tool to inspect it, we want to allow them to start recording and immediately see in the tool what is going on "under the hood" of the Ajax application, while they are using it. Because of the real-time tool feedback on the application under analysis, we expect the tool to be easier to learn.

The second constraint concerns browser compatibility. We would like for the tool not to change a typical interaction with the Ajax application under analysis: the user should be able to interact with the application as usual, i.e. by using a typical browser. The underlying idea is that when users can use a browser that they are familiar with to use and investigate the Ajax application, this further lowers the learning curve of the tool.

Note that these two constraints are not necessary in the strict sense, i.e. the tool could be implemented such that it works in a non-real time fashion and has its own custom browser, for example, and it could work fine. However, in our implementation (see chapter 4) we have respected the above constraints, a decision that was influenced by the empirical study that we were planning to conduct. Because of the short time that participants were given to work with the tool, we wanted to avoid them getting stuck learning the tool. Hence, making the tool easy to learn was important (see chapter 5).

Chapter 4

Tool implementation

The next step in our research process was to create a concrete implementation of the tool that we designed in the previous chapter. This chapter describes our implementation and it highlights interesting decisions that we made during the implementation process. At the end of the chapter we answer our second research question: *is it feasible to build a tool in which trace analysis techniques are applied to the domain of Ajax applications?*

4.1 Architecture

The tool design intentionally leaves certain gaps in its specification – for the most part, we tried to keep it agnostic of specific platforms and technologies. This is favorable because it shows that the tool design can theoretically be implemented on different platforms and can be adapted to different server side technologies. The first step in creating a concrete tool is to fill in these gaps.

Our tool is called FireDetective[1]. Its architecture is shown in Figure 4.1. FireDetective is able to analyze Ajax applications with a Java + JSP back-end, a decision that was influenced by the target application that we chose for our empirical study (see Section 5.5). The tool consists of a Firefox add-on which records JavaScript traces and information about Ajax abstractions, and a server tracer which can be hooked into a Java EE[2] web server. Both of these components forward the data that they record (via sockets) to the visualizer, the third and final component of FireDetective. The visualizer then processes and visualizes the data in real-time.

A benefit of this architecture is that it allows users to use Firefox to interact with an Ajax application, as they normally would, and then use the FireDetective visualizer to inspect what is going on "under the hood". The architecture also enables components to run across different machines (e.g. Firefox add-on + visualizer on one machine, Java server tracer on another). Moreover, additional future components could easily be added, such as a SQL tracer component which could provide insights into the database back-end that many Ajax and web applications rely on.

[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/`.

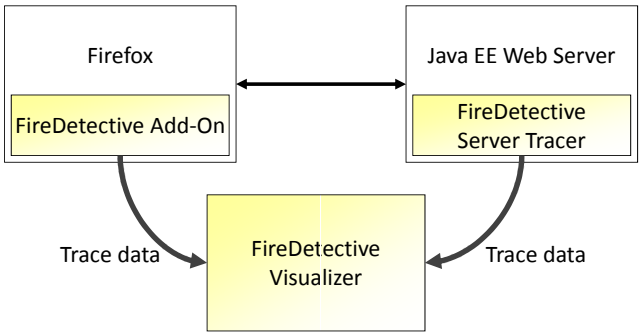


Figure 4.1: Architecture of FireDetective.

The various components of FireDetective are implemented in different languages, using different APIs. First, the FireDetective add-on is implemented in JavaScript, using the add-on API that Firefox provides[3]. We chose Firefox because it satisfies our requirement that the browser needs to be well-known (see Section 3.2.4), and because it provides a relatively mature platform for building browser add-ons. The add-on consists of about 2.2Kloc. The FireDetective server tracer is implemented in C++ (and a tiny bit of Java), a choice which was dictated by the tool interface that we use for tracing the execution of Java code. The server tracer consists of about 1.4Kloc.

Finally, the visualizer is implemented in C#. As a result, the tool only runs on Windows. However, C# allowed us to create the tool much faster since we were already very familiar with the language and its APIs (the .NET framework). Given our goal to create a prototype for our empirical study, rather than a fully polished release candidate, and our limited amount of time, this decision is not hard to justify. The visualizer is the largest component in terms of lines of code; it consists of about 8.1Kloc. In total, this amounts to 11.6Kloc for the whole tool.

4.1.1 Alternative architecture

During the implementation of FireDetective we considered an alternative architecture. The alternative architecture is similar to our current one, except that there is no Firefox add-on. Instead, there is an HTTP proxy component that resides between the browser and server, which intercepts all HTTP traffic and instruments all JavaScript code that it encounters (an approach that is also used in [38], for example). An advantage of this architecture is that the tool user is able to use any browser.

However, dynamically generated code cannot be instrumented, because the code is not present in the HTTP traffic[4]. Moreover, our goal is to build a prototype tool that we can use in our user study, which takes place in a controlled setting. This means we can easily control the browser that our participants will be using. Since our participants

[3] Also see `https://developer.mozilla.org/en/Extensions`.

[4] Theoretically, one can imagine overriding all functions that handle dynamically generated code, functions such as `window.eval`, `window.setTimeout` and the `HTMLelement.onclick` setter, to name a few, and have them instrument each fragment of JavaScript code on the fly, before it is run. However, this would considerably raise the required implementation effort, especially when compared to our current architecture.

are web developers, it is reasonable to assume that they know Firefox. Hence, while support for multiple browsers would be nice, it does not really gain us much in terms of our research. Therefore, we rejected the alternative architecture in favor of our current architecture.

4.2 Firefox add-on

FireDetective's Firefox add-on is responsible for recording JavaScript execution traces and most abstractions: full page requests, (Ajax) requests, DOM events, top-level script loads and timeouts/intervals. For some of these, Firefox offers relatively easy ways to capture them. For others, we need to rely on implementation details and "hacks" to get things to work. Hence, some of the approaches that are discussed here might fail to work in future versions of Firefox.

4.2.1 Recording JavaScript execution traces

We use Firefox' `jsdIDebuggerService` to capture JavaScript execution traces. The interface is able to notify us about a number of interesting events. There is a "script created" callback that is fired for each script that is parsed, and there are "enter script" and "leave script" callbacks that are fired when control flow enters or leaves a script. Firefox passes a `jsdIScript` instance along with every notification; this object can be examined for more information.

Because Firefox notifies us of *all* scripts, even Firefox' internal ones, we first apply a list of filters to see if a script really belongs to the Ajax application that is currently active in the browser. For performance reasons, this list of filters is only applied once for each script, when the script is created. The result of this check is then stored in a hash table, which maps a script's id (the `jsdIScript.tag` property) to the computed information for that script. When our "enter script" or "leave script" callback is called, the tool consults the hash table to quickly decide whether the call should be recorded or not[5].

For our purposes, we distinguish between two kinds of scripts: container scripts and nested scripts. Figure 4.2 illustrates this distinction. A container script refers to a complete fragment of JavaScript code. Examples include top-level scripts (scripts that are included by means of a `<script>` tag within the HTML of the page), event handlers that are defined as string literals (possibly within the HTML of the page) and code fragments that are evaluated by using `window.eval`. Nested scripts refer to JavaScript functions within a container script. In practice, most scripts are nested scripts, and thus refer to JavaScript functions.

For each script, Firefox provides its corresponding function name (when applicable) and a source location, consisting of an URL and a pair of line numbers (first line, last line). The URL refers to an HTTP request that was made earlier. For example, this can be a request for a JavaScript source file, or a request to the main HTML document – in the latter case, the line numbers would refer to a `<script>` element within

[5]Since JavaScript allows us to add members to individual *instances* of classes on the fly, ideally, we would just set a new member variable for every `jsdIScript` instance that needs to be recorded. Unfortunately, since `jsdIScript` is a native class this is not possible, and hence we need the hash table.

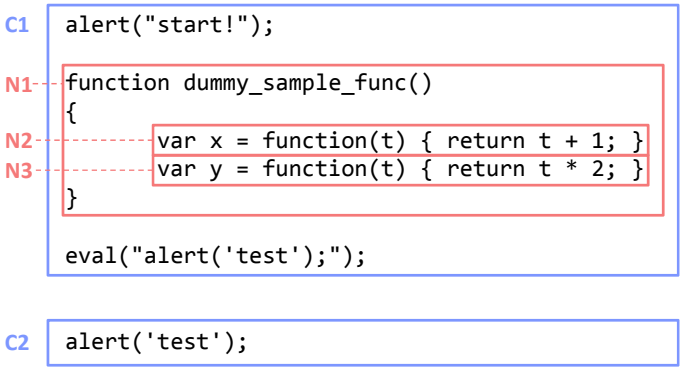


Figure 4.2: Sample code illustrating the difference between container scripts (C1, C2) and nested scripts (N1, N2, N3). C2 is only generated when the call to `eval` takes place. Note that the evaluated code fragment is a container script and *not* a nested script, because it is a string expression.

the document, or a JavaScript function within this element. Note that no source URL is available for dynamically generated code. This is problematic, since it prohibits the tool from showing code for calls to dynamically generated code. We revisit this problem in Section 4.2.9.

From the list of recorded "script enter" and "script leave" calls, we can now reconstruct JavaScript call trees of all JavaScript code that was executed. We also record information about each script, such as its source URL and line number information. This is important, as it enables the tool to jump to the source code of a script.

Finally, library calls (e.g. Dojo or jQuery) are filtered from the trace by using the filtering algorithm described in Section 3.2.3. In our implementation, the filtering is actually performed in C#, inside the visualizer component. The reason for doing so was that we initially wanted to offer two different modes, a "non-filtered" and a "filtered" mode within the visualizer that the tool user could switch between[6]. However, the non-filtered mode was removed from the prototype during one of the pilot sessions of our empirical study (see Section 5.8), because it was not needed during the study, and only added to the learning curve of the tool.

4.2.2 Recording requests

We use Firefox' `nsIObserverService` to record information about outgoing HTTP requests. Via the interface we can set up callbacks that are called for every request that is sent and every response that is received. It is also possible to modify the headers that are sent with the request, and to record (and modify) response data. All of these possibilities are used by the tool, as we explain further below.

When a request occurs, it is first checked against a number of filters. This is necessary because Firefox notifies us of *all* outgoing requests, including requests that Firefox and other add-ons use internally, to check for updates, for example. We only want to record requests that are part of the application under analysis; all other requests are

[6]It is interesting to note that from our personal experience with the non-filtered mode we found that the mode offered a nice insight into the inner workings of some JavaScript libraries, e.g. jQuery.

filtered out and are not recorded. This is achieved by having a blacklist of URLs that the requested URL is matched against. If it is on the black list, it is not recorded. Note that we cannot possibly anticipate which other add-ons a user might have installed, and which HTTP requests they may trigger. Hence, to be sure, all other add-ons should be disabled for FireDetective to work properly[7].

Firefox sets a special flag when a full-page request occurs, which makes it easy for us to detect and record them. Other requests are a bit more complicated, for the following reason: since Firefox has a multi-document/multi-tab user interface, pages may be viewed concurrently. As a result, for each request, the tool needs to figure out to which full-page request it belongs. In fact, this is not only the case for requests, but for all recorded entities, such as JavaScript calls and DOM events. We found that this was not trivial to do, and therefore, we decided for the tool to only support a single instance of Firefox with a single open tab – multiple windows or tabs are not supported.

As a result, assigning recorded entities to the correct full-page request becomes much easier. The process consists of three phases. During the first phase, all recorded entities are assigned to the current full-page request. Then, a new full-page request is initiated (e.g. because the user types an URL into the address bar and hits enter). As Firefox initiates the request, recorded entities are still assigned to the current full-page request. Next, the response comes back. Because the observer service informs us about the response *before* Firefox starts processing it, at this point, there might still be new requests and script calls for the current page. Hence, all recorded entities are still assigned to the current full-page request.

This finally changes when Firefox lets us know that it has started processing the new page. We create a custom implementation of the `nsIWebProgressListener` interface that we pass to Firefox to obtain this information. When Firefox calls the interface's `onLocationChange` method we know it has started processing a new page. This event marks the start of the second phase: non-Ajax requests are assigned to the new full-page request. At the same time, the other full-page request could still be unloading, with various scripts running as a result. Therefore, Ajax requests, DOM events and JavaScript calls are only assigned to the new page when the first function of the new page is parsed. This marks the starts of the third phase, during which all recorded entities are associated with the latest full-page request.

Finally, in the previous section we explained that each script has a source URL which refers to an HTTP request. To be able to show the code later on, we need to capture the HTTP response texts for those requests. This is done by creating a custom implementation of the `nsIStreamListener` interface and asking Firefox to channel every HTTP response through the interface. The tool looks at the content type to determine whether the content of the response should be captured: HTML, CSS and JavaScript documents are captured, but non-code content (e.g. images) is not. The response texts of Ajax requests are always captured.

[7] Section 4.5 gives an overview of the tool's practical limitations, including this one.

4.2.3 Linking browser and server

FireDetective needs to match outgoing requests in Firefox to incoming requests on the server. This is achieved by assigning every outgoing request a unique id. This id is appended to the request in the form of an extra HTTP header named `X-FIRE-DETECTIVE-REQUEST-ID`. Subsequently, the server tracer can detect and record the id, enabling requests that were recorded in Firefox to be matched with incoming requests on the server side.

Note that we cannot rely on just the URL of the request for matching: multiple requests for the same URL may occur, even within a very short time span (e.g. two Ajax requests). Also note that the order in which the requests leave Firefox is not guaranteed to be the same as the the order in which they arrive on the server side. Hence, the extra HTTP header is required in order to successfully connect browser and server.

4.2.4 Recording DOM events

In section 4.2.2 we discussed how we used the `nsIWebProgressListener` interface to let Firefox notify us when it starts processing a new page. This is also the perfect opportunity for setting up the recording mechanism for DOM events. When Firefox notifies us, the page's `window` and `document` objects are guaranteed to exist, but no DOM events have yet been fired.

At this point, we set up event handlers for every possible DOM event[8] on both the `window` and `document` objects. Event listeners are added with their capture flag set, meaning that they will be fired during the initial capture phase of a DOM event, in which it propagates from the top of the DOM hierarchy to the target element that it was fired on. Since the `window` and `document` objects are at the top of the DOM hierarchy, every DOM event that is going to fire on the page, will first trigger one of our event handlers. Note that event handlers need to be added to both the `window` and `document` object because some DOM events only propagate through one of the two.

These handlers do nothing more than recording the name of the DOM event and allowing the normal event propagation to continue. However, subsequent JavaScript top-calls – calls that directly originate from the JavaScript engine, which are at stack depth 0 and which may form the root of a call tree – may now be identified as event handlers and are linked to the latest DOM event that was recorded, offering the tool user an explanation for why the call happened. Note that this approach relies only the single threaded nature of the JavaScript engine: without this property, this approach would not be possible. Also, note that multiple event handlers can be registered for any DOM event, so all subsequent JavaScript top-calls are linked to the event, until another event occurs.

A potential problem with this approach is the number of false positives that it may generate: what if a top-call is triggered by something other than a DOM event? In practice, there are a few common ways in which this can happen: top-level script loads, Ajax event handlers (`XMLHttpRequest.onreadystatestatechanged`) and timeout and interval handlers being called. In the following sections, we explain how

[8] The DOM events specification can be found at `http://www.w3.org/TR/DOM-Level-3-Events/`.

we can detect these situations to prevent misclassification of these situations as DOM events.

Finally, there are more ways to trigger a JavaScript top-call, e.g. using `document.write` to write a `<script>` fragment to the page. These are not detected by the tool. As a result, such a call *will* be misclassified as an event handler for the DOM event that occurred last. However, these situations are not important enough in the target application that we use in our empirical study (see Section 5.5) to warrant inclusion in this tool prototype.

4.2.5 Detecting top-level script loads

Scripts that are included by means of a `<script>` tag within the HTML of the page, are called top-level scripts. As a page loads, Firefox calls each top-level script in turn.

Upon encountering a top-level script, Firefox will parse it, causing it to generate "script created" notifications for each function (nested script) within it, and one final notification for the whole script (container script) itself. This notification is immediately followed by a call to the newly created script. Using the callbacks that we described in Section 4.2.1 it is easy to detect this situation. However, Firefox exhibits the same behavior for *all* container scripts; these are false positives that we need to detect. Fortunately, they can be identified with a few simple tests.

The first type of container script that we consider are DOM event handlers that are defined as strings (either within the HTML or within JavaScript code). When an event handler fires, the `this` pointer is set to the target DOM element for which the event fired. Hence, if `this` points to a DOM element, we know it is an event handler, and not a top-level script. In order to test this, we use a feature of Firefox' debugger interface that enables us to run some JavaScript code in the context of the current stack frame. By simply evaluating the expression `this instanceof Element`, we can decide how to classify the call.

The second type of container script that we consider are pieces of dynamically generated code, evaluated by a call to `window.eval`. These scripts can be discarded in a simple manner. Calls to `eval` are always caused by some other code calling the `eval` function, meaning that the stack will not be empty at that point. In contrast, the stack *will* be empty right before a top-level script load occurs. This allows us to distinguish the two situations.

Other types of container scripts are Ajax event handlers and timeout and interval handlers. In the following two sections we describe how they can be detected.

4.2.6 Linking Ajax requests to code

In an Ajax application, an Ajax request is represented by an instance of the `XMLHttpRequest` class[9,10]. The class has various members that allow the Ajax application to send the request, listen for a response, poll the request's status and read the response text. Our idea is to record invocations of these members, and to link these

[9]The `XMLHttpRequest` specification can be found at `http://www.w3.org/TR/XMLHttpRequest/`.

[10]While JavaScript does not directly support the concept of classes, it supports constructors and prototype objects, from which classes can be built.

invocations to the corresponding Ajax request: by using this information we can then realize the tool's Ajax request life cycle view as discussed in Section 3.2.2.

JavaScript is a highly dynamic language, and allows even classes to be redefined at run-time, by changing their so-called prototype. For our purposes, this is very useful: it allows us to change `XMLHttpRequest`'s prototype and set up code that collects the information that we need.

Our add-on and the Ajax application have separate copies of the `XMLHttpRequest` prototype, because they operate within separate window contexts. Hence, modifying the add-on's local copy of the `XMLHttpRequest` will not work: we need to modify the copy within the Ajax application's window context. Fortunately, accessing the Ajax application's window context and the `XMLHttpRequest` prototype within it from the add-on's context is easy (the other way around is forbidden, since the add-on is running in a more privileged context than the Ajax application). However, while we *were* able to override member functions of the `XMLHttpRequest` prototype, we were unable to install property getter and setter handlers for properties of the prototype across window contexts[11].

To circumvent this issue, we change the `XMLHttpRequest` prototype from within the Ajax application's window context itself. This can be achieved via script injection into the Ajax application. To inject additional code, we use the mechanism that we use to record response content, as described in Section 4.2.2. The interfaces that we use also enable us to rewrite the response text before Firefox processes it. This gives us the perfect opportunity to inject an additional `<script>` tag all the way at the top of the page, which (hence) runs before any other script, and which modifies the `XMLHttpRequest` prototype.

Various members of the prototype are overridden, such as the `send` method, the `onreadystatechanged` event handler setter, and getters of the `status`, `statusText`, `responseText` and `responseXML` variables. The new versions of these members are similar to their original counterparts, except that they also log the action. Furthermore, every `onreadystatechanged` event handler is wrapped in our custom wrapper function which logs the occurrence of the event. Via this approach, we are able to detect occurences of Ajax events and we can prevent misclassifying them as either DOM events or top-level script loads.

Since the injected code runs in the less privileged window context of the Ajax application, communicating recorded actions back to the add-on suddenly becomes a challenge. In order to overcome this problem, we introduce an object called `info` and an empty function called `register`; both are injected into the page[12]. When the injected code wants to communicate with the add-on, it first puts the message that it wants to communicate in the `info` object; then, it calls `register`. This causes the "enter script" callback of our add-on to be called. However, instead of treating the call as a regular function call, the tool recognizes the special name `register`, accesses the `info` object of the Ajax application's `window` object and takes the appropriate action, which is now possible since the add-on still runs in a privileged context.

[11] This has to do with how Firefox wraps objects when they are accessed across window contexts. See `https://developer.mozilla.org/en/XPCNativeWrapper`.

[12] In the tool implementation, these identifiers are prefixed with a string to reduce the likelihood of name clashes with functions and variables from the Ajax application.

A smaller issue is that the injected code also appears in the recorded execution traces. This might confuse tool users, since it is not clear which code is part of the Ajax application and which code was injected by the tool. Therefore, all calls to functions injected by the add-on are stripped from the trace.

Finally, linking each recorded action to the related request is done through a list which represents a mapping from `XMLHttpRequest` instances to ids, which is also injected into the page. When an Ajax request is detected by the observer service, it adds an entry to the list. Subsequently, when action happens, the id can be found by looking for the `XMLHttpRequest` instance in the list.

4.2.7 Detecting timeouts and intervals

The functions `window.setTimeout` and `window.setInterval` take a handler and a time period as their arguments: the JavaScript engine then makes sure that the handler will be executed after the time period has elapsed. This is done once for a timeout, and repeatedly for an interval. The id that the functions return can be used to cancel the timeout or interval by passing the id to `window.clearTimeout` and `window.clearInterval`.

To record timeouts and intervals we use an approach that is similar to the approach used for Ajax requests. Using an injected script, we override the timeout and interval set and clear functions. The overridden versions record the action, and then perform their normal task. The overridden versions of the functions to set a timeout or interval also wrap the handler with a wrapper function, which logs the occurrence of the timeout or interval. Via this approach, we are able to detect timeouts and intervals and we can prevent misclassifying them as DOM events or top-level script loads.

Moreover, the overridden functions and wrapper functions do some additional bookkeeping: a list of timeouts and intervals is maintained, enabling the tool to link timeout and interval handler invocations, and their corresponding set and clear calls.

Note that the Ajax application can specify the handler as a string fragment containing JavaScript code. In this case, we cannot simply call the handler from within the wrapper. As a solution, we let the wrapper call `window.eval` on the code fragment.

4.2.8 Handling anonymous functions

The quality of the execution traces that we record can be slightly improved. For every call in every trace, we want our tool to be able to show and highlight the source code of the corresponding function. However, when multiple anonymous functions are defined on one line, and one of them is called, this becomes impossible. This is because Firefox only provides line number and no column number information per function. From this information alone we cannot decide which of the two functions is actually called.

In order to reconstruct the exact location of the called function, we would ideally access the parse trees that Firefox constructs for each script. However, the debugger interface does not allow us to obtain this information.

Therefore, we build our own JavaScript tokenizer, and a rudimentary JavaScript parser that is able to recognize JavaScript functions. Furthermore, we build a parser for parsing HTML text. The parser is based on the idea of “island grammars”: the

<script> tags represent the islands; the rest of the HTML text is water. Subsequently, the islands are passed to the JavaScript tokenizer and parser.

When Firefox notifies us that it has parsed a container script via a "script created" notification, we parse the script as well with our parser. Recall that Firefox also generates "script created" notifications for nested scripts, i.e. functions. These notifications are generated in a deterministic order[13]. By constructing our parser in such a way that it recognizes functions in exactly the same order as Firefox does, we can assign an exact location (line and column numbers) to each function that Firefox parses.

To further improve the quality of traces, we use a technique to "name" anonymous functions, as described in Section 3.2.3. Since we already have a JavaScript parser, finding the identifier that precedes a function definition is not hard to do. We simply walk back along the leaves of the parse tree and look for an assignment (=) or field initialization (:) token that is preceded by an "lvar", e.g. a variable name, field name (object.field), array element (array[element]) or combinations of these.

In our implementation, the parsers and tokenizers are actually implemented in C#, as a part of the visualizer component. This is done for technical reasons. First, we need the tokenizers and parsers for the source code view of the visualizer anyway, so this saves us from having to implement them in both JavaScript and C#. Second, parsing can be a quite intensive process, and in practice C# code runs faster than JavaScript code[14]. Third, parsing may benefit from using multiple threads, for which support is only rudimentary in Firefox but mature in C#.

4.2.9 Recovering dynamically generated code

As we briefly described in Section 4.2.1, Firefox does not provide a source URL for container and nested scripts that are defined within dynamically generated code. Moreover, the line number information that Firefox provides for an "eval"-ed script is incorrect [5]. This is a problem, because it prohibits the tool from showing source code for calls to dynamically generated code.

To avoid this problem, we can access Firefox' internal copy of a script's source (via the jsdIScript.functionSource property). However, this internal copy has been stripped from its code comments and its layout has been normalized. This goes directly against our aim to always provide a good code context, as outlined in section 3.2.3! If the tool were to present this code to the user, the user would (a) not know where the code fragment came from, and (b) it would most likely not even exist in the code base in this exact form.

Clearly, we would like to do better. In the following two sections, we explore two ways to circumvent the aforementioned problem for "eval"-ed code and event handlers defined as strings.

4.2.10 Recovering code: window.eval

To recover the code fragment in window.eval([code fragment]);, we again rely on script injection into the page. The injected script overrides window.eval, and in-

[13] The functions are traversed in a depth-first way, giving priority to named functions over anonymous functions.

[14] Given the current platforms.

```
<a onclick="javascript: alert(123);">
Click me
</a>
```

```
<a onclick="javascript: alert(123);"
       onclick-pos-info="12;35">
Click me
</a>
```

Figure 4.3: Sample (HTML) code illustrating event handler position information.

troduces an extra variable called `evalExpression`. Upon calling the new version of `eval`, the code fragment – the first argument of the `eval` call – is stored in the variable `evalExpression`. Then, the code fragment is "eval"-ed as usual. However, when the add-on subsequently receives a "script created" notification for a script without a source URL, it can access the variable `evalExpression` in the Ajax application's window context to obtain the code fragment that was just "eval"-ed. The line numbers can be corrected with a simple trick: we need to subtract the line number at which the "eval" call was made from the line numbers that Firefox gives us to find the correct line numbers [5].

For every code fragment with source URL we display the URL. This is important, since it gives the tool user a sense of where the code came from. However, for "eval"-ed code no such URL may exist, since arbitrary code fragments can be "eval"-ed. Yet, in the case of "lazy loading" (see Section 3.2.3) of JavaScript files we can actually provide such an URL. The add-on calculates a hash code of every Ajax request response text[15], as well as a hash code for every "eval"-ed code fragment. If the tool finds two matching hash codes, it can also reconstruct the source URL for the code fragment.

4.2.11 Recovering code: event handlers

Event handlers that are defined as strings may be defined either within HTML, e.g. `<a onclick="`[code fragment]`"/>` or within JavaScript code, e.g. `element.onclick = "`[code fragment]`";`. In Section 4.2.5 we already mentioned that when an event handler is executed, the `this` variable will be set to the target element that the event was dispatched to. This allows us to recover the code fragment by evaluating the expression `this.getAttribute("on`[event name]`")` for each DOM event. For instance, when a click event occurs, we can evaluate `this.getAttribute("onclick")` to obtain the handler code[16].

We would like to take this one step further. When an event handler is defined within the HTML, we would like to show the event handler code fragment within the context

[15] We use the `djb2` string hashing algorithm because of its simplicity. See `http://www.cse.yorku.ca/~oz/hash.html`.

[16] This approach does not work for event handlers that are defined as strings and that are added via the `addEventListener` function, e.g. `element.addEventListener("`[code fragment]`");`. Fortunately, it is more common to specify the handler as a nested function, e.g. `element.addEventListener(function() {`[code fragment]`});`

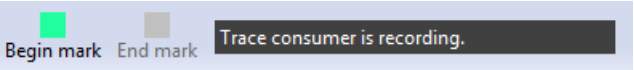


Figure 4.4: FireDetective add-on toolbar, containing the "Begin mark" and "End mark" buttons and a status label.

of the HTML output, rather than just showing the code fragment by itself. To this end, we adapt our script injection component. Instead of only injecting scripts at the top of the page, we modify it to also scan the whole HTML document for event handlers that are defined with the HTML. If it finds an element with such an event handler, it adds an additional attribute named [event name] + `"-pos-info"` to the element[17]. This attribute contains the exact location (i.e. start position and end position) of the event handler code fragment. Figure 4.3 illustrates this process.

When the event handler is executed, we first try to evaluate `this.getAttribute(`[function name] + `"-pos-info")` to obtain position information for the event handler. If this information is available, we can show the code fragment within the context of the HTML that it was defined in. If not, we fall back on the previously described method.

4.2.12 User interface

Tool users use the visualizer to interact with the tool, with one tiny exception: the add-on adds a small toolbar to Firefox, as shown in Figure 4.4. The toolbar shows two buttons, named "Begin mark" and "End mark". These buttons allow users to mark a time slice of the execution of the Ajax application, as discussed in Section 3.2.2. Since tool users will be interacting with the Ajax application when they use the buttons, it is more natural to put them in the Firefox add-on, as opposed to putting them in the visualizer. The toolbar also indicates the status of the connection with the visualizer.

4.2.13 Communicating recorded data

Recorded information is immediately transmitted to enable real-time analysis. The add-on acts as a data provider: it opens a TCP server socket that interesting parties (i.e. the visualizer) can connect to in order to receive all recorded data. Recorded data is encoded according to a custom text based protocol, with fairly straightforward messages such as `REQUEST`, `RESPONSE`, `SCRIPT`, `CALL`, `EVENT`, `CALL-INFO`, `BEGIN-MARK` and `END-MARK` which encode information about HTTP requests, responses (including response texts), scripts (e.g. function names, line numbers), script calls, DOM events, Ajax/timeout/interval specific information and user generated markers, respectively.

[17] Transforming an HTML stream as it arrives in chunks (as is the case in Firefox) is quite interesting. We do not want to delay the response data as this would slow the Ajax application down, i.e. we cannot wait for a whole file to arrive and only then forward it for Firefox to process it. Rather, we need to deal with chunks as they come in. We wrote a custom HTML parser/transformer that is able to handle these constraints.

4.3 Server tracer

The server tracer is responsible for recording Java execution traces and two types of abstractions: incoming HTTP requests and JSP invocations.

4.3.1 Recording Java execution traces

To capture execution traces we use the JVMTI[18] interface. The interface is able to send various types of notifications, such as "enter method" and "leave method" notifications. Moreover, method, class and package names, line number information and execution thread information (necessary, since Java is multi-threaded) can be retrieved. Finally, the notifications can be enabled and disabled at run-time.

The JVMTI can be programmed from within C++. Our server tracer consists of a C++ library (a DLL, in our case), which is passed as a Java VM argument when the Java EE server starts. The Java EE server handles requests in a multi-threaded fashion, i.e. multiple requests may be handled at the same time. "enter method" and "leave method" notifications may occur on different threads and may be interleaved. Fortunately, for any given notification is it easy to determine which thread it occurred on, and we are able to reconstruct a separate execution trace for each execution thread.

However, we quickly found out that having "enter method" and "leave method" notifications enabled all the time slowed the Java EE web server down to a crawl, due to the massive amount of Java calls that occur within the Java EE server. When we performed a quick test and left our "enter method" and "leave method" handlers empty, the server was still very much too slow: the overhead of sending two notifications per method call is just too big. Fortunately, we do not need to record *all* method calls: we are merely interested in the ones that are related to the Ajax application under analysis.

Therefore, we would like to find a way to enable tracing when an HTTP request arrives, and disable it when the HTTP response is returned.

4.3.2 Detecting incoming requests

To detect the start and end of HTTP requests we use a tiny bit of Java code. The Java code takes the form of a single class, called `TraceFilter`. The class is a servlet filter that can be hooked into any typical Java web application. The idea behind a servlet filter is that it gets a chance to process every incoming HTTP request for the web application. Now, in our server tracer, we *could* look for calls to the filter class as a sign that we need to start recording. However, this would still require "enter method" and "leave method" notifications to be on all the time, which is not feasible.

We solve the problem in the following way. When the tracer starts, it disables method call notifications, but it enables exception notifications, that is, the JVMTI notifies us when an exception (even an exception that is later caught) occurs. The general idea behind exceptions is that they are exceptional, i.e. they do not occur often. Therefore, having exception notifications enabled all the time incurs only a minimal performance penalty.

[18] Java Virtual Machine Tool Interface. See `http://java.sun.com/javase/6/docs/technotes/guides/jvmti/`.

Next, when the servlet filter class processes a HTTP request, it throws a `StartTraceException` which it immediately catches, right before the HTTP request is propagated to the Ajax application. At this moment, the server tracer is notified about the exception. Furthermore, it is able to detect that it is a `StartTraceException`, and consequently it enables "method enter" and "method leave" notifications. A similar trick with a `StopTraceException` is used to disable the method notifications again when the Ajax application has generated the response[19]. As a result, we obtain a single execution trace for every HTTP request.

The servlet filter also examines the HTTP request's headers and looks for the `X-FIRE-DETECTIVE-REQUEST-ID` header. If found, it will store the id in a local variable. Upon being notified of the `StartTraceException`, the tracer examines the value of the variable and records it along with the execution trace. Hence, when the visualizer later processes the trace, it will come across the id, allowing it to match the id with one of the requests that was recorded in the Firefox add-on.

4.3.3 Detecting JSP invocations

JSP invocations are reconstructed by recognizing certain calls that occur within the JSP engine. In particular, the tracer looks for classes in the `org.apache.jsp` package, on which the `_jspService` method is called. These classes take the name of [jsp-file]`_jsp`, and refer to [jsp-file].jsp (e.g. a class named `index_jsp` corresponds to index.jsp). When the tracer encounters such a situation, the JSP invocation is detected and can be recorded.

While this approach works well for our target application of our empirical study, 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.

4.3.4 Improving performance

Up to this point, our tracer was still quite slow. We made various changes to improve its performance. First, we only once record method information (its id, fully qualified name, source file and line number information), when a method is called for the first time. For subsequent calls to the method, we just record the method's id. Next, a performance penalty is incurred by directly forwarding the data to the visualizer via a socket. Notifications may arrive simultaneously on multiple threads, but writing to a socket from multiple threads is not supported; therefore, the operation needs to be protected with a critical section. However, this may cause the "enter method" and "leave method" handlers to block as they wait for their turn to write to the socket, thereby preventing the Java program from continuing, as the program can only proceed when the handler returns.

[19] In the JVMTI, we did not find a way to enable notifications on a thread-by-thread basis. Instead, notifications are either on for all threads or off for all threads. We maintain a thread-safe counter (implemented by using critical sections), which is incremented when a request is processed, and decremented when processing for a request has finished. When the counter goes from zero to one, notifications are enabled. When it goes from one to zero, they are disabled.

Our solution is to buffer all calls in a memory buffer. Only when the end of a trace is reached (since we are dealing with traces that represent a single HTTP request, they normally do not take longer than a few seconds) control of the memory buffer is transferred to an extra thread within the server tracer. This thread simply sends each whole buffer to the visualizer. Since the thread is the only one that writes to the socket, no critical sections are required.

These improvements made the Java EE web server run quite fast, with most pages of our mid-size target application of our empirical study (see Section 5.5) loading within a few seconds, especially after a page had been loaded before. However, the visualizer still took several minutes to receive and process all execution trace data. This was due to the high number of library calls that were still present in the traces. These calls were then filtered out by the visualizer, by using the filtering algorithm that we specified in Section 3.2.3.

To speed things up, we decided to push the filtering operation back to the server tracer. This dramatically reduced the amount of trace data that needed to be sent to the visualizer. After the modification, the visualizer processes most requests of our target application within a few seconds, which is sufficient for real-time analysis.

4.3.5 Communicating recorded data

Similar to the Firefox add-on, the server tracer acts a data provider which opens a TCP server socket to which the visualizer can connect. Again, a custom text based protocol is used, with messages such as `REQUEST`, `METHOD` and `CALL`.

4.4 Visualizer

The visualizer connects to both the Firefox add-on and server tracer via two TCP client sockets, processes the data that it receives, and stores it in its data model, which is based on the UML model from Section 3.2.1. The views (which are decoupled from the model) are then notified so they can update themselves in real-time. The views themselves are a straightforward implementation of the visualization that is outlined in Section 3.2.2. Figure 4.5 shows the visualizer in action.

4.5 Conclusion

In the introduction chapter of this thesis we outlined our goal to build a tool that (a) works and (b) is reasonably fast to be of practical use to web developers. After designing and implementing the tool, we can say with confidence that we achieved these goals. A practical advantage is that the tool does not require code instrumentation of an Ajax application, and can be used with a modest number of installation steps[20]. Our second research question – is it feasible to build a tool in which trace analysis techniques are applied to the domain of Ajax applications? – can be answered with a clear yes.

Nevertheless, FireDetective has a couple of limitations that may reduce the tool's practical usefulness, which were introduced during the implementation process. First,

[20]The installation steps are included with the source code distribution of FireDetective.

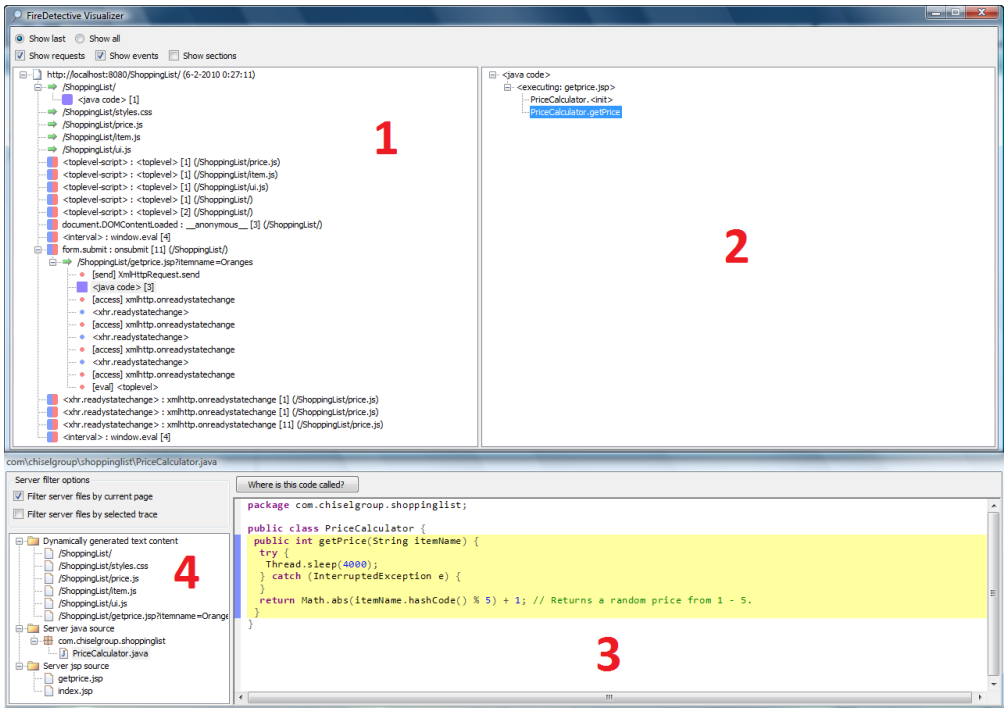


Figure 4.5: The visualizer, showing an analysis of a small sample application. The view numbers refer to the views that are outlined in Section 3.2.2. 1. Abstractions view. An Ajax request is expanded; related traces/calls are shown. 2. Trace view. A number of calls are expanded. 3 Code view. The source code that is related to the call that is selected in the trace view is highlighted. 4. Resource view, showing only the resources that were used on the current page.

we use Firefox' debugger service to record execution traces (see Section 4.2.1). A downside of this interface is that it only allows a single listener per notification type. This means that the service cannot be shared by multiple add-ons, which makes FireDetective incompatible with any other add-on that uses it (such as FireBug). A solution could be to implement FireDetective on top of FireBug. We could then register for the notifications through FireBug, allowing both add-ons to receive the notifications. However, integrating both tools is beyond the scope of this project.

Second, due to the way that we filter HTTP requests, all other add-ons need to be disabled for FireDetective to work correctly. Next, caching needs to be turned off, otherwise, certain HTTP responses and response texts will be missing. Also, the tool only supports a single web-page to be loaded at a time, and does not support multiple browser windows or tabs (see Section 4.2.2). Finally, certain calls may be incorrectly classified as DOM events (see Section 4.2.4).

These limitations are technical and could have been circumvented with more development effort. However, our goal is to create a prototype tool that we can use in our empirical study, which takes place in controlled setting. In such a setting, these limitations do not play a big role, since the tool can be configured beforehand and participants can simply be instructed to work around certain limitations, e.g. asking them not to use multiple tabs. However, we acknowledge that, for a case study in which

developers use the tool on a day-to-day basis, or for a fully polished release candidate, these limitations might need another look.

Chapter 5

Study design

We conducted an exploratory user study to address our two remaining research questions: *which strategies do web developers currently use?* and: *can dynamic analysis improve program understanding for Ajax applications?* This chapter documents the design of the study. It also describes the pilot sessions and main study sessions. The next chapter describes our findings.

5.1 Empirical method

We considered a number of possible empirical methods to use for our empirical study (e.g. [28]).

A case study was the first alternative that we considered. In fact, the approach that we took for getting an initial answer to our first research question can be regarded as a small case study (see Section 3.1). A larger case study could involve observing web developers in a real development setting. However, because of time and resource constraints, this approach was not feasible.

Next, we considered survey research. This would involve making our tool publicly available and sending out surveys to the web developers that use it. However, we were not sure if this approach would give us the necessary level of detail for answering our exploratory research questions. Moreover, both a case study and survey research require more tool polish; thus, more development time is required. Additionally, for our survey research, we would need to "market" the tool and get people to actually use it, which could turn out to be a time consuming effort.

A controlled experiment is an option that does not require more tool development time or marketing. However, an issue with this type of experiment is that it often requires a significant number of participants to yield statistically valid results. It also requires a clearly defined hypothesis, which our exploratory research questions do not provide.

Finally, we decided on an exploratory, pre-experimental user study. 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 to report on facts of real user-behavior, even those observed in under-controlled, limited-sample experiences. We opt for this pre-experimental design because it fits the early stage of this research.

5.2 Design overview

In our study we observed 8 participants working on a number of program understanding tasks. Participants were required to have web development experience. Each participant's session consisted of two distinct parts:

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

In total, each full session took about 2 hours, including a 5-minute break in between the two parts.

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

The handouts that were given out to participants can be found in Appendix A, including: the consent form[1], an introduction sheet, two questionnaires and the two task sets (one for each part of the study).

5.3 Part A: Observing current understanding strategies

Part A is structured as follows:

- Background interview and questionnaire (10 minutes)
- Introduction to tools (10 minutes)
- Working on tasks (35 minutes)
- Short interview (10 minutes)

5.3.1 Background interview and questionnaire

During the background interview we asked participants about their educational background, current occupation (current part-time job or full-time job related to software development), number of years of software development experience and number of years of web development experience. The reason for choosing an interview over

[1]The study was approved by the Human Research Ethics Board of the University of Victoria, where this research took place.

including open-ended questions in the questionnaire was to invite participants to elaborate a bit more about their development experience.

The interview was followed by a background questionnaire, which asked the participant to rate his/her experience with a number of technologies (Java, JSP, JavaScript, Dojo), tools (Eclipse, Firefox, FireBug) and applications (the Java PetStore, target application for the user study) relevant to the study. We used a custom 5-point answer scale to differentiate between experience levels: 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". The reason for using this scale instead of a standard 5-point Likert scale (which typically runs from strongly disagree to strongly agree) was to reduce the subjectiveness of rating one's experience, e.g. a participant answering "strongly agree" to a question such as "I am familiar with Eclipse" might have a different meaning for different participants.

The questionnaire also included a second page that was relevant to part B of the study, which is discussed in section 5.4.

5.3.2 Introduction to tools

Next, participants were given an introduction to the tools that they were going to use. For part A of the study, the tools are Eclipse and FireBug. We chose these tools because we believe they are standard web development tools for working on Java-based web applications.

During the introduction session, the participant was in control of the computer and was given instructions on what to do by the experiment leader. Each session started with a look at Eclipse's project explorer window. The experiment leader showed where the files that were relevant to the tasks could be found, and briefly explained the basic ideas behind Ajax and JSP. Next, the participant was shown three basic FireBug views, namely: the "HTML" view, which shows the current DOM tree in a tree view (and which comes with the "DOM locator" feature that allows a user to click a DOM element on the page to highlight the element within the view), the "script" view, which shows scripts, and the "net" view, which shows the HTTP requests (including Ajax requests) that were sent by the current page. Finally, participants were made clear that they were free to use any feature they liked.

Since participants were likely to have experience with these tools (this was indeed the case, see Section 6.1), the introduction session served mostly to refresh the participants' memory.

5.3.3 Working on the tasks

Before participants started working on the tasks, we indicated that there was no pressure to finish any number of tasks; to reduce pressure on participants, tasks (each task consisted of 2 or 3 subtasks) were handed out one at a time.

Furthermore, participants were informed that they could move on to the next task if they failed to make progress on their current task, and that they could ask questions at any time (questions about the target application itself were not answered, for obvious reasons).

Participants were given 35 minutes to work on the tasks, but this constraint was enforced with a margin of a few minutes, e.g. to prevent cutting off participants as they were on the verge of completing a subtask, as our goal was to find out as much as we could about the strategies that participants used. Also, if the experiment leader noticed that a participant was struggling wi th a particular tool feature, the participant would be given a short explanation of the feature. Again, since our goal was to find out as much as we could about the strategies that participants used, we did not want them to get stuck for too long.

As we already briefly mentioned in Section 5.2, we used a "think aloud" approach. To make sure participants kept vocalizing their thoughts, we asked them questions such as "Can you tell me what you are thinking?", "Why did you just do (x)?", etc. when they fell silent.

5.3.4 Short interview

A short interview asking participants about encountered problems concluded part A. The subtasks were revisited one by one, and for each subtask we asked what problems were encountered. Furthermore, we asked about the biggest problems that participants encountered overall.

5.4 Part B: Support through dynamic analysis

After a 5-minute break, participants continued with part B of the study. During this part, they use Eclipse and FireDetective. Our initial idea was to give participants the tools from part A (Eclipse and FireBug) *and* FireDetective. Unfortunately, FireBug and FireDetective are currently incompatible (see Section 4.5), which is why we disabled FireBug during this part of the study.

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

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

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. The following sections provide more details on the pretest and posttest.

Part B is structured as follows:

- Pretest questionnaire (<5 minutes, conducted before part A)
- Introduction to tools (10 minutes)
- Working on tasks (25 minutes)
- Posttest questionnaire (5 minutes)
- Short interview (10 minutes)

5.4.1 Pretest

The pretest was conducted directly after the background questionnaire of part A, i.e. before any tool introductions or tasks (the background questionnaire and pretest were combined into one questionnaire). The reason for doing so was that we did not want to risk influencing participants' expectations by exposing them to part A of the study. In order to test the expectation of participants, we gave them the following abstract description of a tool like FireDetective:

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.

5.4.2 Introduction to tools

Since we already briefed participants on Eclipse, this introduction focused on FireDetective. To keep the length of the introduction under 10 minutes, the experiment leader took control of the computer and demoed the tool. Participants could interrupt and ask questions at any time. All features of FireDetective were shown, to prevent biasing participants towards specific features.

Since the introduction was more in-depth than the introduction of part A, a small sample application[2] was used to demonstrate the tool instead of the Pet Store. This was done to prevent participants from learning about the Pet Store during the FireDetective introduction.

5.4.3 Working on the tasks

Working on the tasks happened in a way similar to part A of the study. However, another task set was used, and participants worked not for 35, but 25 minutes instead. This choice was made to keep the complete duration of the study under two hours. Since our intent is not to compare the effectiveness of FireDetective to FireBug, this difference in timing is not a concern in our study design. We decided to allocate more time to Part A as understanding how developers use existing tools was more important to us at this stage of our research. The target application was the same as in part A.

[2]We used an application called the ShoppingList, a simple Ajax application written by us. It is included in the source code distribution of FireDetective.

5.4.4 Posttest and final questionnaire

Working on the tasks was followed by a final questionnaire. The first section of the questionnaire represented the posttest.

Next, the questionnaire included two questions regarding the usability of the tool and usefulness of dynamic analysis in general. A 5-point Likert scale was used for both. We added the questions in case participants would rate the tool negatively; in such a case, the two questions might help us understand why the tool was negatively rated. Furthermore, we asked participants whether they thought FireDetective should have been integrated with Eclipse.

Finally, the questionnaire listed 6 interesting features of FireDetective. Participants were asked to choose their top 3 features, i.e. to indicate their best, second best and third best features. They were also asked to indicate the features that they did not find useful at all. The reason for asking about features is that they might help explain why the tool was positively rated (if this is the case) and might shed light on which general techniques are useful for improving program understanding of Ajax applications.

Alternatively, we could have asked participants to rate each feature individually. However, the intention of our approach was to force participants to think a bit more about which features they actually like, since they could not simply choose all of them.

5.4.5 Short interview

A short interview concluded part B. Similar to the interview at the end of part A of the study, the subtasks were revisited one by one, and for each subtask we asked what problems were encountered. This was followed up by questions about what participants liked best about the tool, what they least liked, and finally, whether they have suggestions for improving the tool.

5.5 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"[3]. The application consists of 12KLoc, which are written in a variety of languages, such as HTML, CSS and JavaScript on the client side, and Java and JSP on the server side. All of these files were made available in an Eclipse workspace.

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

[3]Quoted from `http://java.sun.com/developer/releases/petstore/`, retrieved on December 14th, 2009.

5.6 Task design

The study required the design of two task sets, one for each part of the study. We constructed the tasks ourselves, by drawing from our own experience with the Pet Store. Each task set consisted of 4 tasks, divided into 2 or 3 subtasks each, adding up to a total of 10 subtasks per task set. The task descriptions can be found in Appendix A, along with the other handouts that were given to participants during the study.

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 our task sets using Pacione's taxonomy of 9 principal activities [44]. They are listed below:

- 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.
- A7. Investigating patterns in the system's execution (not covered).
- A8. Assessing the quality of the system's design (not covered).
- A9. Understanding the domain of the system (not covered).

We strove for coverage of the first six principal activities, A1 through A6. We did not cover the last three principal activities to limit the number of tasks and reduce the risk of our participants becoming fatigued during the study. We acknowledge that a follow-up study should most likely cover all of Pacione's principal activities.

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.

5.7 Recruiting participants

An important requirement for the participants of our study is that they are representative of the target population that we are investigating, i.e. web developers. Since this research was conducted at a university, students were obvious potential candidates. One potential disadvantage of students is that they might lack the experience that seasoned developers have. Hence, they might not accurately represent our target population.

On the other hand, some students might have software (or web) development jobs on the side. Also, students are frequently used in other empirical studies, mainly because they are easily accessible to researchers and are more willing to spare some of their time than professional software developers. Given these reasons and the fact that we have limited resources, we opted for student participants as well.

We were looking for 8 participants to take part in the study, a number that was largely dictated by the exploratory nature of the study. We expect that a lower number

of participants would have impacted the generalizability of the study. While "more participants" equals "more data" in an absolute sense, for this exploratory study, the additional value that every additional participants brings to the table is likely to diminish as the number of participants grows. Meanwhile, the required amount of data analysis grows linearly with the number of participants (the data analysis process is quite intensive, as it involves going through the screen and audio recordings for every participant, making transcripts of actions, audio and participant thinking steps, and analyzing these transcripts). Given our limited resources, adding more participants to the study would not have been feasible.

From a recruiting perspective, the number is quite modest. However, even in a university setting it can be hard to find participants for an empirical study, especially when there are additional constraints involved (in our case: having web development experience). Therefore, we advertised with free food and drinks and used the following three recruitment strategies:

- Sending an email to the computer science and software engineering graduate student mailing lists.
- Giving two short pitches, one at the start of a computer science course lecture and another one at the start of a software engineering course lecture (both undergraduate courses).
- Word of mouth.

5.8 Pilot sessions

Three pilot sessions were conducted to fine tune the study. Two of the three pilot participants were coworkers of the author; the third pilot participant was recruited via the route that we outlined in the previous section.

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

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

Another issue that came up during the second pilot session was the usability of the tool. Participants used a dual screen computer during the study, and FireDetective originally took up all of this available screen estate. However, participants also needed to the browser in addition to the tool, leading to flipping back and forth between the tool and browser windows. The second pilot participant found this very disorienting. Consequently, we changed the tool in such a way that it would take up only one screen. The other screen could then be used to display the browser window. Also, the tool

user interface was simplified by removing certain viewing options (such as the "non-filtered" mode discussed in Section 4.2.1) to lower the learning curve of the tool.

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.

5.9 Main study sessions

The main study consisted of 9 sessions with 9 different participants. One participant turned out not to be representative of our target population; as a result, we excluded this participant's' data (refer to Section 6.1.1 for more details).

The sessions took place in November 2009, in the usability lab of the Computer Science faculty, at the University of Victoria. This lab allowed us to create a quiet work setting for participants and it offered easy ways to capture audio and computer screens.

During the study, participants used an Intel Quad Core 2.4Ghz 3.25GB RAM machine with two 20 inch monitors, both set to a display resolution of 1600x1200 pixels. The left monitor displayed Eclipse/FireDetective and the right monitor displayed Firefox/FireBug, although participants were free to rearrange these windows according to their liking. The operating system that we used was Windows XP Professional. A user account with default settings was created for use during the study. The Eclipse workspace and Firefox history were manually cleared between sessions. The versions of the tools that we made available to our participants were: Eclipse 3.5 (Galileo) Java EE IDE for Web developers edition, Firefox 3.5.4/3.5.5 and FireBug 1.4.3/1.4.5.

One minor adjustment was made after the first participant's session. During part A of this session, we discovered that the client side source code of the Java BluePrints library was not accessible from Eclipse. Since all of our three pilot participants had relied on FireBug and FireDetective for viewing the client side BluePrints files, this issue had not shown up during the pilot sessions, but led to minor confusion for the first participant. Fortunately, this occurred only at the very end of part A of that session and it did not have an impact on the results. The issue was resolved by including the client side BluePrints and Dojo files in Eclipse's project explorer window.

Finally, Firefox was upgraded from version 3.5.4 to 3.5.5 and FireBug was upgraded from version 1.4.3 to 1.4.5 during the study. Neither upgrade altered the study in any way.

Chapter 6

Study findings

After designing and carrying out the user study, we now continue with the most interesting part of the study: the results. In this chapter we describe and discuss our findings.

6.1 Participant profile

We start with a look at the characteristics of our participants. This section describes their development experience, familiarity with specific technologies and tools and number of tasks worked on.

6.1.1 Exclusion of one participant

All participants in the user study were required to have web development experience. Since the term "web development experience" can be interpreted quite broadly, we specifically asked for basic Java and JavaScript experience. We assumed that when people had experience with these two languages, especially the latter, they would also have experience with web development.

This turned out to be the case for 8 out of 9 participants. One participant indicated to have 0 years of web development experience and this was reflected in the results: the participant was only able to complete the most basic tasks. Because the participant was clearly not representative of the target population, i.e. web developers, we excluded this participant's data. As such, the total number of participants is 8.

6.1.2 Occupations and development experience

Figure 6.1 shows our participants' occupations. Our 8 participants represent our target population quite well. 5 had a professional web development job: 1 full-time and 4 part-time. 2 others had a professional software development job: 1 full-time and 1 part-time. Both of these participants indicated that they worked on web development projects for at least a part of their jobs. Except for the 2 full-time developers, the 6 other participants were either computer science or software engineering students: 4 undergraduate and 2 PhD students.

Figure 6.2 shows participants' development experience. Participants' median number of years of web development experience was 2 years; it can be argued that this is a

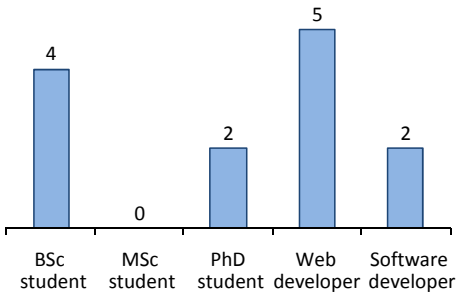


Figure 6.1: Histogram of participants' occupations. Most students also had a part-time software/web development related job (which is the reason that the histogram bars sum up to more than 8).

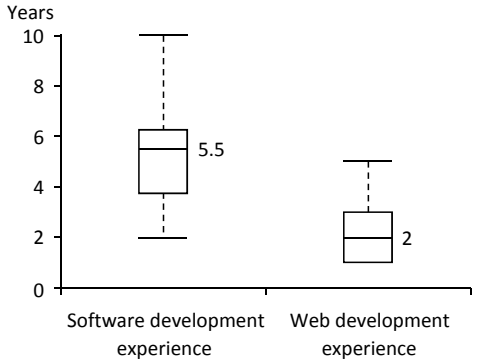


Figure 6.2: Box plots of participants' experience with software and web development.

low number. However, technologies like Ajax have not been around for that long: at the time of writing, the term Ajax has been coined less than 5 years ago [30]. Moreover, from the figure we can see that the median number of years of software development experience was 5.5 years, which shows that participants *did* have general software development skills.

6.1.3 Technology and tool experience

Participants' rated their experience with particular technologies that were relevant to the study. They did so on a custom 5-point scale; the results are shown in Figure 6.3. We can clearly see that participants have a good understanding of Java, JavaScript, Eclipse, Firefox 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 6.4, 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, as well as the questionnaire results (for the Pet Store).

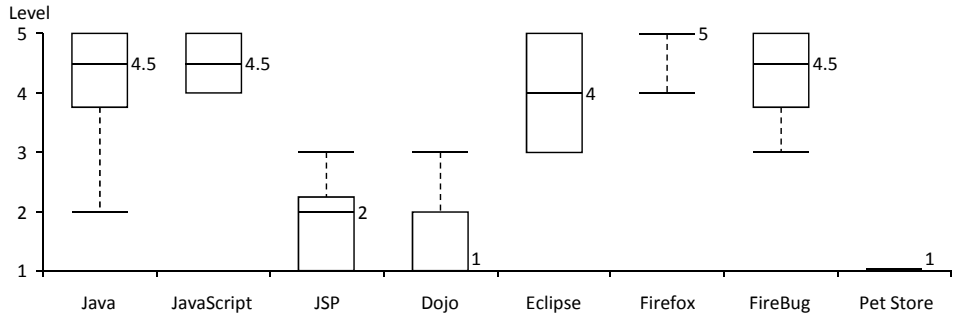


Figure 6.3: Box plots of participants' experience with relevant technologies and tools. The values on the 5-point scale (vertical axis) correspond to 1 = "Never used it", 2 = "Used it for a couple of hours or less", 3 = "Used it for one or two projects", 4 = "I use it regularly" and 5 = "I've been using it regularly for over two years now".

6.1.4 Task completion

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

There was one instance in which a participant did not finish enough tasks to result in coverage of Pacione's first six principal activities (see Section 5.6). In particular, principal activity A2 was not covered during part B of that participant's session. Since this only happened once during the whole study, it is highly unlikely that this impacted the results.

6.2 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?

6.2.1 Methodology

We mainly obtained our insights by observing participants as they worked on tasks. From each participants' audio and screen recordings we created one transcript per participant, which contained all participants' actions, speech and thinking steps. Subsequently, we looked through the transcripts of all participants for interesting sections, recurring patterns, etc.

Due to the exploratory nature of the study, we strove to avoid looking at the transcripts with a predefined set of assumptions and then verifying if we could find evidence for them in the data. Instead, we started from the data: from there we looked for sections of interest, and attempted to link this information to existing theories. The in-

terview questions were mainly used to look for confirming evidence to what we found in the data.

6.2.2 Observations

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 [45], i.e. finding definitions and/or occurrences of functions, methods and classes.

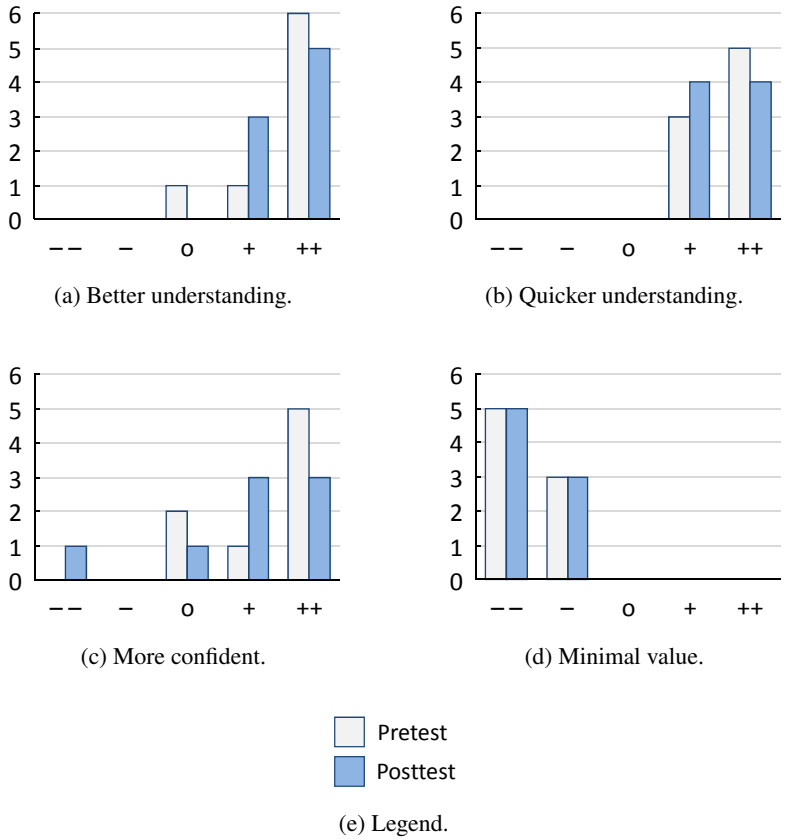
In order to explore these control flow relationships, all participants made heavy use of text search. While Eclipse provides functionality for exploring control flow, e.g., the "Open Declaration" and "Call Hierarchy" functions, these functions were only occasionally used by participants (far less than text search). A possible reason for this might be that these functions (currently) do not always work as expected for web applications: for instance, opening the "Call hierarchy" of a Java method does not show calls made from a JSP file, and "Open Declaration" does not always work well with JavaScript's anonymous functions.

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

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

Finally, we observed frequent use of *ad hoc* strategies such as determining whether to explore a file based on whether a filename "sounds right" and scrolling through code looking for clues.

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



(a) Better understanding.

(b) Quicker understanding.

(c) More confident.

(d) Minimal value.

(e) Legend.

Figure 6.4: Histograms 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.

6.3 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?

6.3.1 Methodology

If dynamic analysis is indeed able to improve program understanding for Ajax applications, we would also like to learn more about *how* this works, and what we can do to further improve understanding. We obtained insights into these questions via four different routes: the pretest–posttest, the questionnaire about feature usefulness, observing participants using the tool and the final interview.

Analysis of the questionnaire answers is straightforward, as it simply involves copying answers and generating tables and graphs. Next, as in part A, we created transcripts of our participants working on tasks, including all their actions, speech and thinking steps. The analysis of the transcripts was a bit different compared to part A: we mainly looked at how FireDetective supported (or failed to support) the partici-

pants' understanding processes. During the interview the emphasis was put on what participants liked and disliked about FireDetective and how they thought the tool could be improved. By asking these questions we hoped to obtain insights into how to further improve program understanding.

6.3.2 Pretest–posttest

The results of the pretest and posttest are shown in Figure 6.4. From a first look at the results we can see that the pretest and posttest results are fairly similar. The posttest results are the most valuable, as they reflect the participants' actual experiences with the tool. Nevertheless, it is interesting to know that participants did not completely switch their opinions before and after using the tool.

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

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

6.3.3 Features

We asked participants' opinion on 6 features of FireDetective that we wanted to investigate in more detail: the abstractions view (F1), the resources view which is filtered and only shows the resources 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 6.5. We can see that there is no clear winning feature. However, we *can* observe some trends, which may give us some insight into how FireDetective helped improve program understanding.

The abstractions view (F1) and time slicing (of the abstractions 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 resources 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 resources view provides an initial overview of these relevant files, such that participants have a better starting point for investigation.

Finally, we did not specifically ask participants about the "where is this code called" feature; the feature had been implemented just a week before conducting the study, was still a little buggy, and we initially did not think of it as a major feature. However, one participant was particularly enthusiastic about it and declared it the best liked feature of the tool during the interview. Interesting is that the feature provides a nice way to go back from code to traces and abstractions, which is important if users are browsing code (by following the static relationships in the code that they are familiar with) and then want to find the context(s) in which the code they have come across is used. As such, the feature might play an important role in the understanding process, and future investigation might yield interesting results about how top-down and bottom-up comprehension strategies are combined by developers when using a trace analysis tool in general (i.e. not necessarily specific to Ajax applications).

Discussion. Participants mainly use a combination of top-down and bottom-up strategies, which is different from part A, during which they mainly relied on only bottom-up strategies. The abstractions view and linking of browser and server play a role in enabling the top-down approach.

However, it is difficult to determine with certainty which elements of FireDetective are the main contributors to its usefulness. Some features, such as "showing the code related to a call in the code view" and "naming of anonymous functions" (automatic), are untestable via a questionnaire: these features are used all the time, but because of that it can be hard for participants to determine whether these features were actually useful. More in depth research is needed, to more precisely identify which techniques contribute to improving understanding, and to better understand how top-down and bottom-up strategies are combined.

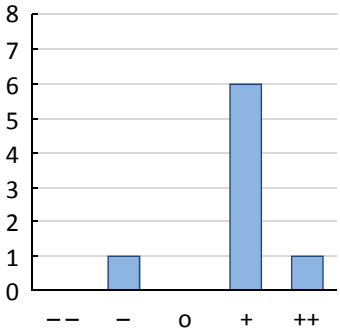
6.3.4 Additional questions

In addition to the posttest, we asked participants three questions (as described in Section 5.4.4). The first one concerned whether the tool was easy-to-use; the second concerned whether dynamic analysis in general was useful. Figure 6.6 shows participants' answers to these questions. The questions were introduced to enable us to explain a potentially negative rating of FireDetective, but since this was not the case (and the answers to the questions are not particularly surprising), they are not very relevant. We have included them for the reader's reference.

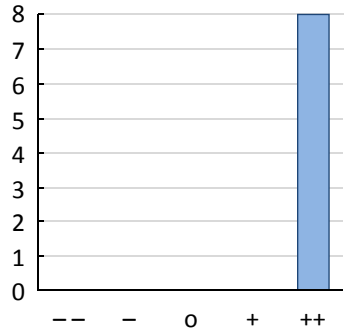
With the third additional question we attempted to poll whether participants thought that FireDetective should have been integrated with Eclipse. FireDetective is not currently integrated with an IDE; however, since an IDE is often a developer's primary

	1	2	3	4	5	6	7	8
F1: Abstraction overview			1	1	2		1	3
F2: Filtered resources 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 6.5: 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.



(a) FireDetective is easy-to-use.



(b) There's added value in using dynamic analysis for analyzing web applications.

Figure 6.6: Histograms of participants' answers to two additional questions. Horizontal axes: 5-point Likert scale, ranging from strongly disagree ("−−") to strongly agree ("++"). Vertical axes: number of participants.

tool for inspecting code, we were wondering about the relation between the two. As it turned out, our phrasing was a little unfortunate: we asked about Eclipse, but should have asked about IDEs in general. We noticed this when the subject came up during the interview. Where one participant would indicate to like Eclipse integration because he/she was very familiar with Eclipse (while such a participant might not have been in favor of IDE integration in general), another participant would indicate to not want Eclipse integration because he/she just did not like the feel of Eclipse (while such a participant might have been in favor of integration with other IDEs).

As such, we decided to discard the answers to this side question. We *do* note that a few participants reported that they would have liked to see a stronger link with the browser, similar to FireBug. While it is too early to draw any conclusions, this could be an interesting topic for further investigation.

6.3.5 Observations

Participants encountered a number of issues when working with FireDetective. In this section, we describe the issues that are related to ideas and techniques behind FireDetective. These may represent interesting leads for future research. Usability issues that are specific to our particular implementation of FireDetective are described in the next section.

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. They can be installed anywhere in the request handler chain. Installation and configuration of filters happens via an XML file. Because the tool records calls to all methods, it also shows calls made to filter classes. However, it cannot show *why* these calls occur, since the internal server logic that calls the filters is hidden from view, and even if the tool were to show these internal calls, it would produce a distorted picture, since the real cause of the filter being called is a binding specified in an XML file. During the study, several participants encountered this problem. They were wondering why the EntryFilter class of the PetStore was invoked, but the tool was unable to give them this information.

Another problem occurred during a task in which participants had to examine a bug, caused by a click handler that contained a syntax error. Participants, still unaware of the cause of the bug, would trigger the click event and search for it in the abstractions 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 abstractions view was filtered to show only the last full-page request. However, participants did not always notice this, causing them to think that they were dealing with an Ajax request, while it was actually a full-page request. This confused them because they were looking for an Ajax request that did not exist.

When asked about potential tool improvements, participants often indicated integration with FireBug, providing evidence for the fact that FireBug and FireDetective are complementary. Participants also asked for mechanisms to reduce the amount of visible information: they were sometimes overwhelmed by the information shown. Since we used only basic trace visualization and reduction techniques, this was to be expected.

In particular, participants asked mostly for features to filter the abstractions view, such as: keyword filtering (i.e. searching for “click” to only show click handlers), a way to separate out requests and events in multiple windows (more FireBug-like), and filtering different types of noise (e.g. timeouts, which occurred frequently since Dojo uses a polling-based method for handling Ajax requests, that is based on frequent

timeouts). The trace view is less useful for larger traces, because the view does not compact repetitions in traces (caused by loops, for example) at all. Thus, in such cases the view may show long lists of calls which are not convenient to work with. Participants would also have liked particular static analysis techniques, such as full text search in the code view, possibly because they are attached to their old way of working, but probably because static and dynamic analysis are complementary techniques.

Discussion. From the observations that we made we distilled three ways in which program understanding for Ajax applications can be further improved:

- **Other types of Ajax/web-related 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. Going back to the EntryFilter example, it would have been useful if the trace view had shown an "XML binding invocation" in the call tree, showing the invoked EntryFilter class as a child node and showing the relevant lines of XML code when clicked.

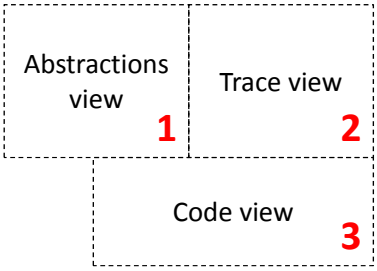
  Other abstractions that we found evidence for being useful are the JavaScript script parsing process and the errors that occur during it and DOM mutations. Existing abstractions could also be linked in more ways. For instance, when a full-page request is triggered by a submitted HTML form on the previous page, this link could be captured and shown.

- **Different kinds of visualizations.** FireDetective's visualizations are straightforward representations of the recorded abstractions and traces. Only simple trace reduction techniques were used, which – expectedly – caused participants to be overloaded with information on various occasions. We should investigate how to visualize the connected network of abstractions, traces and code in better ways.

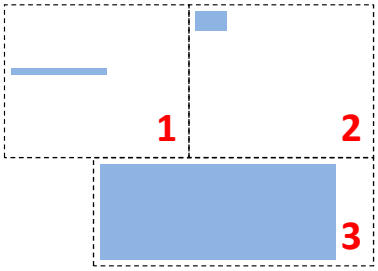
- **Integration with existing tools.** From the study it became obvious that FireDetective and FireBug are complementary tools. It could be interesting to investigate how these tools exactly complement each other and how they can be integrated more tightly.

6.3.6 Usability issues

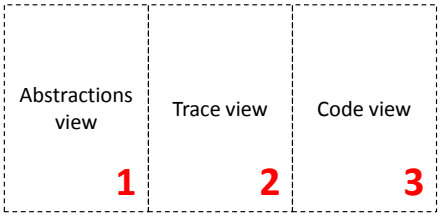
Participants generally found FireDetective easy-to-use (see Section 6.3.4) and we did not find evidence that the user interface significantly hindered use of the tool. However, the study did uncover a number of usability issues with FireDetective that we would like to mention. They deserve to be looked into if development on FireDetective were to continue, especially since they are not particularly expensive to fix. Moreover, they might be relevant to (possible future) tools that are in some way similar to FireDetective, because these tools might share some of the underlying user interface ideas, and therefore also share some of the associated problems.



(a) FireDetective view layout.



(b) Screen areas that usually change when selecting an element from the abstractions view.



(c) Suggested view layout.

Figure 6.7: Illustrating the view layout usability issue.

The biggest usability issue that we encountered had to do with the layout of the views. As we described in Section 3.2.2, the tool contains three main views with increasing levels of detail. This allows users to effectively "zoom in" on parts of the Ajax application, i.e. they can start at the abstractions view (1), then dive into the trace view (2), and finally look at the code view (3). Section 4.4 shows a screenshot of FireDetective and its views: a schematic version of the view layout is shown in Figure 6.7a.

Now, suppose we were to select a trace node in the abstractions view (1). This causes the call tree of the trace to be shown in the trace view (2), and the code of the first call to be shown in the code view (3). We expected users to move their focus to the trace view (2) to examine the calls, clicking different calls to see the related source code in (3). However, this was not what happened! Figure 6.7b shows the screen areas that change, on average, when a node is selected from the abstractions view. As can be seen from the figure, only a small part of the trace view usually changes: this is because traces are initially almost fully collapsed; only the top-most call is expanded. On the other hand, the code window usually changes completely to show the function related to the selected call. Therefore, almost all participants intuitively shifted their focus directly from the abstractions view (1) to the code view (3), completely missing the trace view, which left them wondering how to get more information. In fact, even if the code view did not change completely, showing only a line or two of code, participants would already be "trained" to shift their focus to that view.

Eventually, participants picked up on the view layout; the issue only occurs when learning the tool. Nevertheless, especially in a user study with limited time, it would

be better to avoid it. Positioning the views in a different way, for example as shown in 6.7c (which takes advantage of the fact that modern displays are wide-screen monitors), might help avoid the problem.

Another usability issue was related to scrolling the abstractions view. Some participants preferred to trigger some action in the Ajax application, while monitoring the abstractions view and looking for new events. Interestingly, these participants were the ones that did not use the time slicing feature, instead, they used the real-time updated views of the tool instead to get a similar result. Unfortunately for them, the abstractions view could not be made to scroll automatically, so as soon as the view contained more elements than it could show (which usually happened quite soon), they would need to manually scroll down to see new events when they occurred. A check-box to force the view to always scroll down to show its latest content could be of help here, although we should not forget about alternative visualizations/filtering methods that prevent to view from quickly overflowing in the first place.

Other, smaller issues included: not keeping expanded resource view tree nodes expanded between full-page requests, and lack of line numbers and identifier highlighting (e.g. highlighting all occurences of a variable when the cursor is positioned within the variable name) in the code view.

6.4 Threats to validity

This section discusses several threats to the validity of the study. They are divided into two categories: internal validity and external validity.

6.4.1 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. We mitigated this concern by indicating to participants that only honest answers were valuable.

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

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

6.4.2 External validity

A concern regarding the generalizability of the results is that most participants were students. However, as shown in Section 6.1 a lot of these participants had a relevant part-time job. Participants were not familiar with two of the technologies used in the

study, JSP and Dojo. We admit that the learning curve involved has likely impacted the results. Yet, we also think that this impact is limited because both JSP and Dojo are technologies that are very similar to rivaling technologies. Because all participants had web development experience, even if they are not familiar with JSP and Dojo in particular, they are likely to have experience with similar technologies. Moreover, participants were given a brief introduction to JSP, and were allowed to ask questions about the technologies involved at any time.

The Java Pet Store, our target application, is a showcase application. This might cause one to question whether this application is representative of a real-world Ajax application. For example, the application might be more artificial than a real web application. However, the application represents the state-of-the-practice and manual inspection of the applications 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. [38].

Finally, the tasks might not have been representative of real-world tasks. Because of the limited time frame tasks are likely to be shorter than real-world tasks, and they might not have covered all program understanding aspects. We tried to mitigate this threat by using the Pacione's framework of principal comprehension activities [44] to make sure that the tasks are realistic and cover a significant portion of the program comprehension spectrum. Furthermore, we used open-ended questions instead of multiple choice questions to better mimic real-world tasks.

Chapter 7

Related work

This chapter gives an overview of previous efforts that have focused on program understanding in the web domain, by using dynamic analysis.

7.1 Web applications

Early web application reverse engineering efforts were mainly focused on architecture reconstruction, e.g. [24, 34, 48, 57]. Static analysis alone does not suffice because of the dynamic nature of web applications [2, 57], so in most cases static analysis is complemented by dynamic analysis.

Slightly more recent are efforts by Antoniol *et al.* [2], who present *Wanda*, a tool that records page accesses, HTTP request and session variables, database, file and web service I/O, and accesses to external components. The tool then combines this information into various types of diagrams. The tool is combined with *Ware* [24], a static analysis tool, in [25].

However, the approaches that are listed in this section do not take into account most of the client side aspects that are common in Ajax applications. This limits their usefulness when applied to Ajax applications.

7.2 Web services

De Pauw *et al.* [21, 22] present the *Web Services Navigator*, a tool for offering insight into message and transaction flows in systems of multiple web services, such as in service-oriented architecture applications. The tool combines multiple web service events logs (which we could refer to as traces) to reconstruct meaningful abstractions in the web service domain, such as messages and invocations. This idea has some similarities with the idea behind FireDetective, although the domain is different. Moreover, the tool only shows what happens *between* entities. It does not show what happens *within* entities.

7.3 Ajax applications

Some recent efforts have focused on program understanding of the client side of Ajax applications. Li and Wohlstadter present a tool named *Script InSight* [38], which uses

dynamic analysis to record DOM modifications (i.e. changes to the current page) and relate them to the JavaScript functions that caused them. This allows a web developer to map an element on the page to the points in the code where the element was modified. The tool can also abstract several mutations into a DOM mutation graph. While the authors did not empirically verify the usefulness of their approach, being able to relate DOM modifications to code seems useful – indeed, during our empirical study we encountered a situation where we expect that the technique would have been helpful.

Oney and Myers present a tool named *FireCrystal* [42], which enables a user to view a timeline of DOM events and DOM modifications. The tool shows code coverage for each event and allows DOM events and DOM modifications to be rewound and played back.

Our approach differs from these two approaches in a number of ways. First, our approach visualizes execution traces. Second, it combines client side information with server side information to show a *complete* picture of an Ajax application (as opposed to showing only information about the client side 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 that is of interest, named DynaTrace Ajax[1]. 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. To the best of our knowledge, there exists no published research on DynaTrace Ajax.

As far as we know, our research is the first research effort to investigate the understanding of Ajax applications through an empirical user study.

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

Chapter 8

Conclusions and future work

8.1 Conclusions

In the introduction of this thesis, we created a research plan for answering our three research questions. The questions are listed below, along with the insights that we have obtained over the course of this research.

- **RQ1: Which strategies do web developers currently use when they try to understand Ajax applications?**

We used our own web development experience to provide an initial answer to this question, by using introspection. We found that a lot of manual effort was required for following control flow, and that picking a good starting point for exploration sometimes proved to be difficult.

The first part of our exploratory study showed 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; tool support should be improved.

- **RQ2: Is it feasible to build a tool in which trace analysis techniques are applied to the domain of Ajax applications?**

Using our initial findings to our first research question as a basis, we created a tool design and implemented it. The tool records execution traces on both the browser and server, captures information about Ajax/web abstractions and visualizes the information in a linked way.

A number of technical challenges had to be overcome during the implementation process, such as finding ways to capture information about all abstractions that we were interested in, and linking traces between browser and server. However, after doing so we found that an implementation definitely belonged to the realm of possibility. The result is a tool called FireDetective, and by creating it we demonstrated the answer to our second research question to be a clear yes.

- **RQ3: Can we use trace analysis to improve program understanding for Ajax applications?**

While demonstrating the feasibility of a trace analysis tool for Ajax application was a nice result, it was not the main goal that we had in mind when creating FireDetective. The main idea was to use FireDetective it in the second part of our empirical user study to gain insights into how FireDetective could potentially aid program understanding for Ajax applications, our third research question.

Participants indicated that FireDetective – which uses dynamic analysis – allows them to understand Ajax applications more effectively, more efficiently and with more confidence. A possible explanation could be that the tool offers the option to switch to a more top-down way of understanding, in which the recorded abstractions and the links between them play a role. However, more research is needed to verify this hypothesis, and also to better understand how top-down and bottom-up strategies are combined and how individual elements contribute to the understanding process. From the observations and interviews during the user study we identify three different ways to further support the understanding process: incorporating information about additional Ajax/web domain specific abstractions, exploration of other kinds of visualizations and integration with existing tools.

8.2 Contributions

We have made the following contributions:

- We have created a tool design, in which trace analysis is applied to the domain of Ajax applications. The tool design demonstrates how to employ abstractions from the Ajax/web domain to link execution traces, with the intention of improving program understanding.
- We created FireDetective, a concrete implementation of our tool design. FireDetective is open source and can be downloaded from `http://swerl.tudelft.nl/bin/view/Main/FireDetective`.
- We have carried out a preliminary user study that showed us (a) how developers currently go about understanding Ajax applications and (b) that dynamic analysis techniques can improve their understanding.

8.3 Future work

Understanding the understanding process. Now that we have found preliminary evidence for FireDetective's usefulness, an interesting avenue for future work is to investigate more precisely how FireDetective exactly influences the program understanding process, i.e. *why* it is useful. We have found a first answer to this question, but more research is needed.

Preferably, this would take the form of another empirical study, such as a different exploratory study or a controlled experiment. Such a study might also be able to reconfirm (or refute) our preliminary evidence about the the tool's usefulness, and might

yield additional potential ways to improve program understanding for Ajax applications. A longitudinal study might be of interest for similar reasons.

Improving program understanding. Next, we can also investigate how other techniques can further improve program understanding of Ajax applications. We should explore other types of visualizations and user interfaces: for instance, FireDetective uses (very) straightforward visualization techniques, and we expect they can be much improved. Moreover, we think it would be of interest to investigate integration of tools like FireDetective with existing tools, such as a debugging tool like FireBug.

We already mentioned the introduction of more types of abstractions in FireDetective, e.g. various types of XML bindings and JavaScript parsing errors, as a possible improvement. These are the abstractions for which we found direct evidence in our user study. However, they might really just be the tip of the iceberg.

As a first example, we could detect and record Ajax push (Comet) [9] "requests" in JavaScript code, and present them as such. Next, with the advent of HTML5, and its canvas element, video support and local data storage, to name just a few, many new abstractions arise that can be captured and tracked to offer insights into an Ajax application. Server side frameworks may also contain abstractions that are useful to track. We already mentioned various types of XML bindings, but why stop there? For instance, we could record information about higher-level abstractions such as models, views and controllers within a *model-view-controller*-framework-based Ajax application, and show how they are involved. Finally, abstractions may be linked in more ways. For example, an idea could be to relate DOM elements to the exact server call that generated the element.

After introducing new techniques, we must carefully evaluate empirically how individual techniques influence the understanding process.

Facilitating the learning of frameworks. In environments in which many heterogenous components play a role, such as Ajax applications, it may take a while to get a good understanding of all languages and frameworks that are involved. Especially the ways in which all of these components interact may take time to grasp. In our view, a static perspective (i.e. looking at code) does not really help to get a feel for these interactions.

However, dynamic analysis tools, like FireDetective, are able to present a completely orthogonal view of a complex system such as an Ajax application. In particular, when pieces of dynamic information are linked, as is the case in FireDetective, we expect that developers are able to get a much better sense of interactions between components and frameworks. As an example, suppose we have a new version of FireDetective that records information about XML bindings. Then, by looking at *only a single trace* of the Pet Store application, a developer could learn: that `dojo.io.bind` sends an Ajax request; that the file `web.xml` is responsible for passing the request through an access control filter; that `<%taglib ... %>` tags can be used to include custom tag libraries; that these tag libraries are linked to actual Java classes via a `.tld` file, and which methods are called as an result of using the tag in a JSP page. In our opinion, that it is quite a bit of useful information condensed in a single place.

Web developers who are already familiar with concepts behind web languages and frameworks, but who are migrating between technologies, could benefit as well. For example, a PHP + CakePHP + jQuery developer who now needs to work with

a Java + JSP + EJB[1] + Dojo web application, may use a dynamic analysis tool like FireDetective to get up to speed with the new technologies.

We are not currently aware of how large the need for facilitating the process of learning web technologies is. Depending on this need, it could be interesting to investigate the role that dynamic analysis could play in such a learning process.

[1]Enterprise Java Beans. See `http://java.sun.com/products/ejb/`.

Bibliography

[1] James H. Andrews. Testing using log file analysis: tools, methods, and issues. In *ASE '98: Proceedings of the 13th IEEE International Conference on Automated Software Engineering*, pages 157–166, Washington, DC, USA, 1998. IEEE Computer Society.

[2] Giuliano Antoniol, Massimiliano Di Penta, and Michele Zazzara. Understanding web applications through dynamic analysis. In *Int'l Workshop on Program Comprehension*, pages 120–129. IEEE, 2004.

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

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

[5] John J. Barton. Finding errors in dynamically created Javascript source, 2007. `http://www.almaden.ibm.com/u/bartonjj/fireclipse/test/DynLoadTest/WebContent/DynamicJavascriptErrors.htm`, retrieved on March 11th, 2010.

[6] Laszlo A. Belady and Meir. M. Lehman. A model of large program development. *IBM Systems Journal*, 15(3):225–252, 1976.

[7] Chris Bennett, Del Myers, Margaret-Anne Storey, Daniel M. German, David Ouellet, Martin Salois, and Philippe Charland. 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.

[8] Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas Webster. The concept assignment problem in program understanding. In *ICSE '93: Proceedings of the 15th International Conference on Software Engineering*, pages 482–498, Los Alamitos, CA, USA, 1993. IEEE Computer Society.

[9] Engin Bozdag, Ali Mesbah, and Arie van Deursen. A comparison of push and pull techniques for AJAX. In *WSE '07: Proceedings of the 9th IEEE International Workshop on Web Site Evolution*, pages 15–22, Washington, DC, USA, 2007. IEEE Computer Society.

[10] Silvia Breu, Thomas Zimmermann, and Christian Lindig. Mining eclipse for cross-cutting concerns. In *MSR '06: Proceedings of the 2006 International Workshop on Mining Software Repositories*, pages 94–97, New York, NY, USA, 2006. ACM.

[11] Ruven Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18(6):543–554, 1983.

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

[13] Andrew Chan, Reid Holmes, Gail C. Murphy, and Annie T. T. Ying. Scaling an object-oriented system execution visualizer through sampling. In *IWPC '03: Proceedings of the 11th IEEE International Workshop on Program Comprehension*, pages 237–244, Washington, DC, USA, 2003. IEEE Computer Society.

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

[15] Bas Cornelissen, Leon Moonen, and Andy Zaidman. An assessment methodology for trace reduction techniques. In *Int'l Conf. on Software Maintenance (ICSM)*, pages 107–116. IEEE, 2008.

[16] Bas Cornelissen, Arie van Deursen, Leon Moonen, and Andy Zaidman. Visualizing testsuites to aid in software understanding. In *CSMR '07: Proceedings of the 11th European Conference on Software Maintenance and Reengineering*, pages 213–222, Washington, DC, USA, 2007. IEEE Computer Society.

[17] Bas Cornelissen, Andy Zaidman, Danny Holten, Leon Moonen, Arie van Deursen, and Jarke J. van Wijk. Execution trace analysis through massive sequence and circular bundle views. *Journal of Systems and Software*, 81(12):2252–2268, 2008.

[18] Bas Cornelissen, Andy Zaidman, and Arie van Deursen. A controlled experiment for program comprehension through trace visualization. *IEEE Transactions on Software Engineering*, 0(0):0–0, 2010.

[19] Bas Cornelissen, Andy Zaidman, Arie van Deursen, Leon Moonen, and Rainer Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, 35(5):684–702, 2009.

[20] Brian De Alwis and Gail C. Murphy. Using visual momentum to explain disorientation in the eclipse IDE. In *VLHCC '06: Proceedings of the Visual Languages and Human-Centric Computing*, pages 51–54, Washington, DC, USA, 2006. IEEE Computer Society.

[21] Wim De Pauw, Sophia Krasikov, and John Morar. Execution patterns for visualizing web services. In *Proc. Symposium on Software Visualization (SOFTVIS)*, pages 37–45. ACM, 2006.

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

[23] Wim De Pauw, David Lorenz, John Vlissides, and Mark Wegman. Execution patterns in object-oriented visualization. In *COOTS '98: Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems*, pages 219–234, Berkeley, CA, USA, 1998. USENIX Association.

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

[25] Giuseppe A. Di Lucca and Massimiliano Di Penta. Integrating static and dynamic analysis to improve the comprehension of existing web applications. In *Proc. of the International Symposium on Web Site Evolution (WSE)*, pages 87–94. IEEE, 2005.

[26] Stéphane Ducasse, Michele Lanza, and Roland Bertuli. High-level polymetric views of condensed run-time information. In *CSMR '04: Proceedings of the Eighth Conference on Software Maintenance and Reengineering*, pages 309–318, Washington, DC, USA, 2004. IEEE Computer Society.

[27] Philippe Dugerdil. Using trace sampling techniques to identify dynamic clusters of classes. In *CASCON '07: Proceedings of the 2004 Conference of the Centre for Advanced Studies on Collaborative Research*, pages 306–314, New York, NY, USA, 2007. ACM.

[28] Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. Selecting empirical methods for software engineering research. In *Guide to Advanced Empirical Software Engineering*. Springer, 2008.

[29] Éric Tanter, Jacques Noyé, Denis Caromel, and Pierre Cointe. Partial behavioral reflection: spatial and temporal selection of reification. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 27–46, New York, NY, USA, 2003. ACM.

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

[31] Abdelwahab Hamou-Lhadj and Timothy C. Lethbridge. Techniques for reducing the complexity of object-oriented execution traces. In *VISSOFT '03: Proceedings of the 2nd IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 35–40, 2003.

[32] Abdelwahab Hamou-Lhadj and Timothy C. Lethbridge. A survey of trace exploration tools and techniques. In *CASCON '04: Proceedings of the 2004 Conference of the Centre for Advanced Studies on Collaborative Research*, pages 42–55. IBM, 2004.

[33] Abdelwahab Hamou-Lhadj and Timothy C. Lethbridge. Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. In *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension*, pages 181–190, Washington, DC, USA, 2006. IEEE Computer Society.

[34] Ahmed E. Hassan and Richard C. Holt. Architecture recovery of web applications. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 349–359. ACM, 2002.

[35] Gregor Kiczales. Aspect-oriented programming. *ACM Computing Surveys (CSUR)*, page 154, 1996.

[36] Adrian Kuhn and Orla Greevy. Exploiting the analogy between traces and signal processing. In *ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pages 320–329, Washington, DC, USA, 2006. IEEE Computer Society.

[37] Stanley Letovsky. Cognitive processes in program comprehension. In *Papers presented at the 1st Workshop on Empirical Studies of Programmers*, pages 58–79, Norwood, NJ, USA, 1986. Ablex Publishing Corp.

[38] Peng Li and Eric Wohlstadter. Script InSight: Using models to explore JavaScript code from the browser view. In *Int'l Conf. Web Engineering (ICWE)*, pages 260–274. Springer, 2009.

[39] David C. Littman, Jeannine Pinto, Stanley Letovsky, and Elliot Soloway. Mental models and software maintenance. In *Papers presented at the 1st Workshop on Empirical Studies of Programmers*, pages 80–98, Norwood, NJ, USA, 1986. Ablex Publishing Corp.

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

[41] Del Myers, Margaret-Anne Storey, and Martin Salois. Utilizing debug information to compact loops in large program traces. In *CSMR '10: Proceedings of the 14th European Conference on Software Maintenance and Reengineering*, pages 41–50, 2010.

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

[43] Michael J. Pacione, Marc Roper, and Murray Wood. A comparative evaluation of dynamic visualisation tools. In *WCRE '03: Proceedings of the 10th Working Conference on Reverse Engineering*, pages 80–89, Washington, DC, USA, 2003. IEEE Computer Society.

[44] Micheal Pacione, Mark Roper, and Murray Wood. A novel software visualisation model to support software comprehension. In *Working Conf. Rev. Engineering*, pages 70–79. IEEE, 2004.

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

[46] Steven P. Reiss. Visualizing Java in action. In *SoftVis '03: Proceedings of the 2003 ACM Symposium on Software Visualization*, pages 57–ff, New York, NY, USA, 2003. ACM.

[47] Steven P. Reiss and Manos Renieris. Encoding program executions. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 221–230, Washington, DC, USA, 2001. IEEE Computer Society.

[48] Filippo Ricca and Paolo Tonella. Analysis and testing of web applications. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE)*, pages 25–34. IEEE, 2001.

[49] Tamar Richner and Stéphane Ducasse. Using dynamic information for the iterative recovery of collaborations and roles. In *ICSM '02: Proceedings of the International Conference on Software Maintenance*, pages 34–43, Washington, DC, USA, 2002. IEEE Computer Society.

[50] Ben Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *Proc. Symposium on Visual Languages (VL)*, pages 336–343. IEEE, 1996.

[51] Dag I. K. Sjøberg, Tore Dybå, and Magne Jørgensen. The future of empirical methods in software engineering research. In *FOSE '07: Future of Software Engineering*, pages 358–378, Washington, DC, USA, 2007. IEEE.

[52] Dag I.K. Sjøberg, Jo E. Hannay, Ove Hansen, Vigdis By Kampenes, Amela Karahasanovic, Nils-Kristian Liborg, and Anette C. Rekdal. A survey of controlled experiments in software engineering. *IEEE Transactions on Software Engineering*, 31(9):733–753, 2005.

[53] Elliot Soloway, Robin Lampert, Stanley Letovsky, David C. Littman, and Jeannine Pinto. Designing documentation to compensate for delocalized plans. *Communications of the ACM*, 31(11):1259–1267, 1988.

[54] Joel Spolsky. The law of leaky abstractions, 2002. `http://www.joelonsoftware.com/articles/LeakyAbstractions.html`, retrieved on January 14th, 2009.

[55] Margaret-Anne Storey, Frank Fracchia, and Hausi Müller. Cognitive design elements to support the construction of a mental model during software visualization. In *IWPC '97: Proceedings of the 5th International Workshop on Program Comprehension*, pages 17–28, Washington, DC, USA, 1997. IEEE Computer Society.

[56] Tarja Systä, Kai Koskimies, and Hausi Müller. Shimba – an environment for reverse engineering Java software systems. *Software – Practice & Experience*, 31(4):371–394, 2001.

[57] Paolo Tonella and Filippo Ricca. Dynamic model extraction and statistical analysis of web applications. In *Proc. Int'l Workshop on Web Site Evolution (WSE)*, pages 43–52. IEEE, 2002.

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

Appendix A

Study handouts

Starting with the next page, this appendix contains all handouts that were used for our emprical study. Handouts were originally printed in A4 format; in this appendix, they are rescaled to fit within this document's layout. Handouts are included in the following order: consent form (3 pages), introduction form (1 page), first questionnaire (2 pages), final questionnaire (2 pages), task set of part A (4 pages), task set of part B (4 pages). Note that tasks were handed out to participants one at a time, so most participants did not see all task sheets.

Tool support for understanding web applications

Consent Form

You are being invited to participate in a study entitled "Tool support for understanding web applications". This study is being conducted to evaluate a new tool for understanding web applications (called "FireDetective") and its underlying theories.

The study is being conducted by Nick Matthijssen, a master's student at the *University of Victoria* and *Technische Universiteit Delft* (in The Netherlands). He is supervised by Dr. Margaret-Anne Storey and Dr. Ian Bull, from the Department of Computer Science at the *University of Victoria*, and Dr. Andy Zaidman and Dr. Arie van Deursen, from the Department of Computer Science at the *Technische Universiteit Delft*. To eliminate inducement and coercion, Dr. Storey and Dr. Bull will not be involved in the recruitment process, nor will they participate in the study sessions or interviews. If you have further questions, please contact Dr. Margaret-Anne Storey by email at mstorey@uvic.ca or by phone at (250) 472 5713. For more information about the research groups, please refer to www.thechiselgroup.com and swerl.tudelft.nl.

Purpose and objectives

Understanding existing software so it can be updated and enhanced (software evolution) can be a challenging task that requires sophisticated tool support. The purpose of the research project is to gain more knowledge about how people go about understanding web applications and how tools can support them in that process. This knowledge can be used to further improve tool support.

What is involved?

Participation in this study requires that you work individually on a number of software evolution tasks (e.g., locating a feature, determining the impact of a proposed change, or finding software defect). During these tasks, you will be asked to "think aloud": i.e. vocalize your thoughts. Both before and after these tasks you will be asked to fill in a questionnaire. Also, at the beginning, halfway during the experiment, and after the experiment, you will be interviewed. Each interview will take 5 to 10 minutes.

Figure A.1: Consent form (page 1 of 3).

Voluntary participation

Your participation is completely voluntary. One 2-hour session will be requested of you. Completion of tasks is not required. You may withdraw at any point during this session. If you choose to withdraw, at your request, all data from the session will be destroyed.

Recorded data

If you consent, audio will be recorded and a screen recording will be made during the experiment. Please indicate your consent:

☐ I consent to having my voice recorded.
☐ I do not consent to having my voice recorded.

☐ I consent to having the computer screen(s) recorded.
☐ I do not consent to having the computer screen(s) recorded.

You will be asked to fill in two questionnaires and to participate in three short interviews. The information that you provide as well as the data that is gathered during the experiment will be presented anonymously.

It is anticipated that this information and this data will be used to contribute to a master's thesis and possibly one or more research papers.

Anonymity

In terms of protecting your anonymity, you will be assigned a unique ID so that your identity will only be accessible to the principal researcher. The unique identifiers will be used to aggregate your data but will in no way be used to identify you personally. All information disclosed will be analyzed by trained researchers and all data will be kept secure and protected at all times in password protected files on a secure server. Study data will be kept for three years. At the end of this time computer data files will be deleted, and this consent form will be shredded.

Compensation

Compensation offered for participating in this study will take the form of snacks and drinks provided during the experiment. Compensation will always be provided, even if you withdraw.

You may also benefit from using the tool and you may also gain insight into how to effectively perform software evolution tasks. There are no known or anticipated risks to you by participating in this research.

Figure A.2: Consent form (page 2 of 3).

Benefits

All participants will be able to examine the dissemination of the study results via scholarly publications. In addition to being able to contact the researchers, you may verify the ethical approval of this study, or raise any concerns you might have, by contacting the Human Research Ethics Office, at the University of Victoria (ethics@uvic.ca).

_____	_____	_____
Participant name	Participant signature	Date

Figure A.3: Consent form (page 3 of 3).

Tool support for understanding web applications

Introduction + outline

Thanks for participating in this experiment!

The experiment will take about 2 hours in total, and is structured as follows:

- Introduction + background interview + questionnaire — *15 minutes*

- Session 1: Standard tools
  - Introduction to JSP, JavaScript and FireBug — *10 minutes*
  - Program understanding tasks to be completed with Eclipse, Firefox and FireBug (standard web development tools) — *35 minutes*
  - Short interview — *10 minutes*

- Break — *5 minutes*

- Session 2: FireDetective
  - Introduction to FireDetective — *10 minutes*
  - Program understanding tasks to be completed with Eclipse, Firefox and FireDetective — *25 minutes*

- Final questionnaire + short interview — *10 minutes*

Total time: ±120 minutes.

The tasks

The experiment is broken up into two sessions. Each session consists of an introduction followed by a number of program understanding tasks. During these tasks, you will be studying a web application called the "Java Petstore": an example application written by the folks at Sun. The tasks vary from locating relevant portions of code, finding the cause of a bug, learning about the architecture of the Petstore, etc.

Note that the number of tasks you complete is not important – just do what you can. Feel free to ask questions at any time!

Figure A.4: Introduction form (page 1 of 1).

Tool support for understanding web applications

Background questionnaire

A. Software development experience

Listed below are a number of programming languages, frameworks, tools and applications. Please indicate your experience with them below, where:
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	2	3	4	5
Java	☐	☐	☐	☐	☐
Java Server Pages (JSP)	☐	☐	☐	☐	☐
JavaScript	☐	☐	☐	☐	☐
Dojo JavaScript framework	☐	☐	☐	☐	☐
Eclipse	☐	☐	☐	☐	☐
Firefox	☐	☐	☐	☐	☐
FireBug	☐	☐	☐	☐	☐
Java PetStore	☐	☐	☐	☐	☐

(please flip page)

Figure A.5: First questionnaire (page 1 of 2).

B. Understanding web applications

Please read the following description.

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	2	3	4	5
Such a tool could allow me to better understand web apps.	☐	☐	☐	☐	☐
Such a tool could allow me to be more confident that I really understand the web application that I'm investigating.	☐	☐	☐	☐	☐
The value added by such a tool will be minimal.	☐	☐	☐	☐	☐
Such a tool could save me time.	☐	☐	☐	☐	☐

Figure A.6: First questionnaire (page 2 of 2).

Tool support for understanding web applications

Final questionnaire

Thanks for completing the tasks! Please take a moment to fill out the following questionnaire. The questionnaire will be followed up by a short interview.

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	2	3	4	5
I found FireDetective easy to use.	☐	☐	☐	☐	☐
FireDetective should have been integrated with Eclipse.	☐	☐	☐	☐	☐

B. Tool adequacy

	1	2	3	4	5
There's added value in using dynamic (i.e. runtime) information for analyzing web applications.	☐	☐	☐	☐	☐
The value added by a tool like FireDetective is minimal.	☐	☐	☐	☐	☐
A tool like FireDetective saves me time.	☐	☐	☐	☐	☐
A tool like FireDetective allows me to better understand web apps.	☐	☐	☐	☐	☐
A tool like FireDetective makes me more confident that I really understand the web application that I'm investigating.	☐	☐	☐	☐	☐

(please flip page)

Figure A.7: Final questionnaire (page 1 of 2).

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.

	Having an overview of events and the JavaScript functions that handle them.
	Being able to directly jump from (ajax) requests to the corresponding server side code.
	Real-time trace analysis, i.e.: (almost) no delay between capturing traces and analyzing them.
	Marking sections of a trace using the Firefox add-on, by using "Begin mark" and "End mark".
	Being able to easily track xhr (ajax) requests.
	Filtering packages and java files based on the current page or trace.

Next, please mark features that you didn't find useful with an 'X'.
If you found all of the features useful, please check this box: ☐

Figure A.8: Final questionnaire (page 2 of 2).

Tool support for understanding web applications

Task 1

Please note that your answers will be kept confidential. They will be used for research purposes: we have no interest in evaluating your answers personally.

The number of tasks that you complete is not important – just do what you can. Feel free to ask questions at any time!

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)

1

Figure A.9: Task set A (page 1 of 4).

Tool support for understanding web applications

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?

Figure A.10: Task set A (page 2 of 4).

Tool support for understanding web applications

Task 3

Seller page

Navigate to the seller page.

a) Clicking on the 'next' button does not trigger the form's 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?

3

Figure A.11: Task set A (page 3 of 4).

Tool support for understanding web applications

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 doesn't appear if you quickly hover over a description?

Figure A.12: Task set A (page 4 of 4).

Tool support for understanding web applications

Task 1

Please note that your answers will be kept confidential. They will be used for research purposes: we have no interest in evaluating your answers personally.

The number of tasks that you complete is not important – just do what you can. Feel free to ask questions at any time!

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?

1

Figure A.13: Task set B (page 1 of 4).

Tool support for understanding web applications

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” -> handled by handleEvent -> etc. -> 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)?

Figure A.14: Task set B (page 2 of 4).

Tool support for understanding web applications

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 “Nick’s goldfish” as inappropriate. This is supposed to delete the pet. Why does this not work? (i.e.: where’s the bug?)

3

Figure A.15: Task set B (page 3 of 4).

Tool support for understanding web applications

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?

4

Figure A.16: Task set B (page 4 of 4).