NORTHWESTERN UNIVERSITY

Readily Available Learning Experiences in Production Code

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

Field of Computer Science

By

Joshua James Hibschman

EVANSTON, ILLINOIS

September 2017

© Copyright by Joshua James Hibschman 2017

All Rights Reserved

ABSTRACT

Readily Available Learning Experiences in Production Code

Joshua James Hibschman

Online platforms for learning to code such as Coursera, CodeCademy, and CodeSchool attract millions of learners and significantly expand the pool of self-starting developers, yet critical gaps in knowledge and experience remain between inexperienced learners and professionals. With vast amounts of professionally-authored source code made readily available by the client-server architecture of the web, design patterns and implementation decisions found in source code can be used to provide on-demand learning experiences for users seeking to advance their skills in professional web development. Specifically, this thesis focuses on creating *Readily Available Learning Experiences* (RALE) for inexperienced learners who wish to become professional contributors but lack the means necessary to advance beyond their gaps in knowledge. The central claims of RALE are (1) surfacing hidden design patterns, code constructs, and relationships (both direct and indirect) from professional websites, (2) minimizing learning barriers while supporting personalized exploration of unfamiliar website code, (3) scaffolding mixed-initiative sensemaking to help users walk through unfamiliar complexities, and (4) scaling the conversion of examples into

learning resources without additional authorship or maintenance. Specifically, I propose to transform the entire domain of professional websites into opportunities for authentic learning. Professional websites offer rich details missing from training examples, providing real-world content and opportunities to think in the modes of the discipline. They embed programming concepts and implementation techniques that are used by professionals and are continually updated as new solutions arise. However, despite the abundant availability of web client source code, professional website source is complex and difficult for learners to understand. This thesis contributes three technical contributions to support RALE on the open web: (1) an API Harness for surfacing relevant code that modifies the DOM, (2) a Wisat architecture and Sleight-of-Hand technique to enable source instrumentation on production websites, and (3) a Serialized Deanonymization technique to expose hidden asynchronous links between logical JavaScript components. With these techniques for transforming websites into learning experiences, aspiring web developers have immediate opportunities to gain authentic practice in professional web development beyond what authored learning materials currently provide.

Acknowledgements

Thank you to my advisor Haoqi Zhang for his coaching and mentoring during my journey through Ph.D. research and graduate life. Through our many whiteboarding sessions, hacked-together prototypes, scrappy paper-drafts, and "stupid" ideas, I am grateful for Haoqi's patient and consistent emphasis on conducting meaningful and impactful research over chasing publication counts. I am especially grateful that during our time working together, Haoqi prioritized quality of life and mental well-being. It has been especially great to see Haoqi treat all of his students this way, promoting community and quality of life in our hybrid graduate/undergraduate research environment.

Thank you to my committee for their valuable guidance and feedback through the final phase of my Ph.D. Thank you Darren Gergle for teaching me research methods and how to conduct user studies properly, for inspiring my research direction, and for being a good friend of Delta Lab. Thank you Rob Miller for your sound feedback and guidance through the formation of this thesis and for inspiring a great deal of this work in online learning. Thank you Eleanor O'Rourke for your well-timed guidance and feedback, especially in applying aspects of the Learning Sciences in CS.

Thank you to my fellow Northwestern researchers for your support and friendship through my Ph.D. Yongsung Kim, thank you for being the best labmate one could ask for, see Figure A.1. To Mike Greenberg, Emily Harburg, Daniel Rees Lewis, Julie Hui, and Leesha Maliakal thank you for your wonderful friendship and cheering me onto the

finish. Thank you Uri Klarman, David Demeter, and Marlon Twyman for the walks, coffee runs, lunches, and impromptu lab visits, which helped me through the low points in my research and grad life. Thank you Sarah Lim for your outstanding continuation of my research and Kevin Chen, Ryan Madden, Henry Spindell, Kapil Garg for your friendship and inspiring undergraduate and master's work. Thank you John Otto and Shawn O'Bannion for your guidance on finishing a Ph.D.

Thank you to my prior mentors and teachers who have inspired me to think scientifically and have given me the tools to contribute to the field Computer Science. Taylor University professors Tom Nurkkala, Jon Geisler, Stefan Brandle, and Art White thank you for your inspiring undergraduate lectures and projects in the fundamentals of Computer Science. DePaul University professors Corin Pitcher, Massimo DiPierro, Chris Hield, Steve Jost thank you for your mentorship through advanced masters topics in Computer Science. Thank you Will Brennan, Jim Buswell, and Jimmy Bremner for being noteworthy examples of mentors and managers and thank you for believing in my potential as a software engineer — giving me opportunities to grow as a professional contributor. Thank you Caryn Ellison for inspiring such a strong vision of scientific research in my life from a young age.

Spencer Mead and Caleb Durenberger, thank you for providing an opportunity to mentor you through difficult topics in Computer Science and for being the primary inspirations for this thesis.

Tom Lieber and Andy Ko, thank you for your foundational works on Theseus and Learning Barriers, of which this thesis is heavily based upon.

Thank you Jeff Nichols for hosting me as a researcher at Google, advancing my skills as an industry researcher, and making me feel like a valued member of your team.

"Us" especially wants to thank Dr. Rick Marks for your timely counsel, support, and mentorship. We've never been better.

Most of all, thank you to my family. To my parents Jim and Robin Hibschman, thank you for my first computer at age 10, for teaching me to dream, and for your incredible support during this process. Thank you to my parents in-law Ralph and JoAnn Wilson for your friendship, guidance, and consistent encouragement during my Ph.D. Thank you to my beautiful children Lucy, Milo, and Reid for bringing such indescribable joy to me every single day and for giving Daddy so much time on the computer. Thank you to my beautiful wife Kelley for being my advocate, number one supporter, incredible mother to our kids, and consistent believer that I could finish my Ph.D.

Soli Deo gloria.

Table of Contents

List of Figures

these representations to reflect their current understanding by dragging and labeling nodes, editing and commenting on source code in source frames, and adding custom facet filters. By clicking on a node in the call graph, users can open source frame views (middle) which display specific function invocation states in the runtime with their inputs and outputs, parent and child calls, asynchronous declaration context, asynchronous binding, and asynchronous effect if present. Facets (top) allow learners to view functionally-related slices of code in the call graph; predefined facet filters include Mouse, Keyboard, Setup, AJAX, and DOM. Users can apply `or` and `not` operators to engage multiple facets to expose desired views. In this example, the learner added a custom "Hover Effect" facet, comments to the source code, and node labels as they made sense of components in the call tree. 137

5.2 A cluster of related collated function invocations (with their invoke-counts) in a condensed call graph, manually organized here for display. Nodes are colored green for top level calls and yellow for currently-selected; other nodes are colored based on the facets they match: purple for DOM, white for AJAX, and blue for Setup. Edges in the graph are color-coded yellow for call relationships, orange for asynchronous declaration, and purple for asynchronous bind locations. In this toy-example of a lazy-loaded image, a click handler is bound on `#test4`. Upon clicking `#test4`, the handler makes an AJAX

CHAPTER 1

Introduction

With vast amounts of professionally-authored website source code made readily available by the client-server architecture of the web, implementation details and design choices found in source code can be used to provide on-demand learning experiences for users seeking to advance their skills in professional web development. This thesis introduces methods for (1) surfacing hidden design patterns, code constructs, and relationships (both direct and indirect) from professional websites, (2) minimizing learning barriers while supporting personalized exploration of unfamiliar website code, (3) scaffolding mixed-initiative sensemaking to help users walk through unfamiliar complexities, and (4) scaling the conversion of examples into learning resources without additional authorship or maintenance.

Specifically, this thesis focuses on creating learning experiences for inexperienced web developers who wish to become professional contributors but lack the means necessary to advance beyond their gaps in knowledge. Many of these users can setup, read, and write basic JavaScript web applications but lack the conceptual knowledge of design patterns used in professional web solutions. Online platforms for learning to code such as Codecademy, Khan Academy, and CodeSchool attract millions of learners and significantly expand the pool of self-starting developers, yet these platforms primarily teach syntax or provide practice on constrained tutorial examples. Further, these platforms lack the authenticity required to support the progression from writing functional code

to writing professional-quality software. As a result, significant gaps in knowledge and experience remain between inexperienced developers and professional developers.

This thesis proposes to address conceptual knowledge gaps for inexperienced web developers by transforming the entire domain of professional websites into opportunities for authentic learning. Professional websites offer rich details missing from training examples, providing real-world content and opportunities to think in the modes of the discipline. They embed programming concepts and implementation techniques that are used by professionals and are continually updated as new solutions arise. However, despite the abundant availability of front-end code, professional examples are complex and difficult for learners to understand.

Deriving learning material from websites presents design and technical challenges due to the magnitude and complexity of the underlying source code. A simple UI interaction may require only ten lines of JavaScript, but modern web production engineering practices make use of libraries and build processes that can push front-end lines of code into the tens of thousands [**6, 77, 86**]. Bindings between HTML and JavaScript support an interaction, but it is difficult to determine how such bindings are constructed. A simple calendar widget, for example, could be created entirely in JavaScript and appended to the DOM with listeners, or it could be built in HTML and CSS with inline calls to JavaScript hooks. Embedding the widget amidst all its library or utility code in a minification build process blurs the location and scope of code most relevant to enabling the widget's functionality.

JavaScript functions are often executed asynchronously, and visualizations of execution order provide little information about the conceptual structure of web programs. One

could understand the structure by walking through the entire execution path as they might when debugging, but this can contain thousands of steps for professional examples. Surfacing relevant information, such as the most-called functions, is a reasonable approach for identifying important functional components, but it can hide lower-level functions that become necessary bridges for understanding how components work together to produce a feature.

With prior systems **[53, 67, 15, 31, 5]**, it is difficult to (1) differentiate relevance amidst JavaScript sources for a given interaction, (2) control the scope of JavaScript being analyzed, (3) identify the interplay between JavaScript and HTML that causes a visual change, (4) trim away inactive code and library code that get in the way of learning, (5) elicit code constructs and relationships with JavaScript sources, and (6) support a user's sensemaking process through complex JavaScript artifacts. Our work on RALE overcomes these challenges to give users opportunities to learn authentically from professional websites.

1.1. Readily Available Learning Experiences

A RALE is defined as a learning scaffold created for a professional website that engages with a learner to support authentic practice with constructs and techniques used to compose the product. The purpose of RALE is to enable inexperienced learners to gain insights from professional products to bridge knowledge gaps in their journey of becoming a professional contributor. To advance this purpose, we argue that a RALE should:

(1) Surface hidden design patterns, code constructs, and relationships (both direct and indirect) from professional websites.

(2) Minimize learning barriers while supporting personalized exploration of unfamiliar website code.

(3) Scaffold mixed-initiative sensemaking to help users walk through unfamiliar complexities in the surfaced resources.

(4) Scale the conversion of examples into learning resources without additional authorship or maintenance.

Surfacing patterns, constructs, and relationships from professional websites (claim 1) helps inexperienced developers to overcome gaps in knowledge, shortcuts inefficient learning routes such as incomplete tutorials, and provides users with opportunities to learn authentically — in a personally meaningful way using multiple modes of the discipline (i.e. web application programming). Prior systems to RALE are able to surface information about a runtime but require the user to infer relationships and patterns for themselves. Without scaffolds in place, learning barriers often inhibit users from gaining meaningful insights from any information surfaced [**49**].

RALE calls for minimizing the effects of additional learning barriers that can be created by surfacing hidden details from software while still supporting in-depth exploration (claim 2). While there is a large body of work on the theme of extracting concepts from code, few address the risk that surfacing patterns, constructs, and relationships could overwhelm a user and present them with additional learning barriers. The studies presented later in this thesis reveal that learners are easily overwhelmed by overly detailed stack traces, verbose variable states, and complex nested program flows. However, there are times when learners seek these higher levels of detail. Thus, RALE supports personal in-depth exploration in its second claim.

RALE calls for techniques and affordances to scaffold learners in sensemaking, providing them with cues to engage in multiple modes of the web programming discipline such as architecture or implementation (claim 3). Further, by facilitating sensemaking as a mixed-initiative system, learners can work with the system toward their goal [**42**] to analyze, modify, and iterate on conceptual models of programming constructs. While prior tools [**26, 8, 9**] effectively support opportunistic sensemaking by leveraging context and online resources to provide relevant programming scaffolds, they are designed to support expert developers; they do not provide affordances to build on beginners' understandings [**56**] or help beginners reason about the structure of code [**70**].

With the first 3 claims in mind, a RALE must apply these claims while scaling continuously across its application domain without burdens of authorship or maintenance (claim 4). Tutorials and Q&A require authorship, or the manual creation of learning materials, and often leave learning gaps or fail to provide an adequate case library for each learner's goals. The underlying technologies enabling RALE should not require expert authoring for surfacing the underlying information from a runtime. With millions of active learners looking to the web for online learning and the fast pace of innovation in web programming, teachers and content authors meet only a small portion of the increasing demand for learning materials. The primary goal of this thesis is to transform inspiring professionally-produced websites into opportunities for learning without dependencies on authoring to bridge gaps in becoming a professional contributor.

1.2. Contributions: Three Systems Toward RALE

This thesis contributes three systems toward the goal of creating Readily Available Learning Experiences (RALE) for professional websites. Each system was developed sequentially in an effort to get closer to an overall goal of proving RALE on the open web. This section introduces the systems and their contributions toward RALE.

1.2.1. Unravel: Tracing, Organizing, and Identifying Relevant Runtime Code

Unravel is an extension of the Chrome Developer Tools for quickly tracing and visualizing HTML changes, JavaScript method calls, and JavaScript libraries. Unravel aids the reverse engineering of websites by providing comprehensive yet targeted views of JavaScript invocations, HTML changes, and included libraries (see Figure 1.1). Unravel enhances Google Chrome's existing developer toolkit by linking all HTML and JavaScript components to their corresponding inspection panes for quick examination. Unravel works on all websites without interfering with existing functionality. For example, a developer can navigate to a landing page, record a parallax effect, and watch Unravel identify which lines of JavaScript were executed, which DOM elements were modified, and which attributes were modified per each element. Toward RALE, Unravel was the first system to surface code constructs in terms of relevance for a given website feature (founding claim 1) while lowering informational learning barriers (founding claim 2) and scaling to work across its domain without the need for external dependencies (founding claim 4).

The main conceptual contribution of Unravel is the idea of *tracing, identifying, and organizing the most relevant functions and DOM elements manipulated to support reverse engineering and understanding interactions on complex professional websites.* Unravel

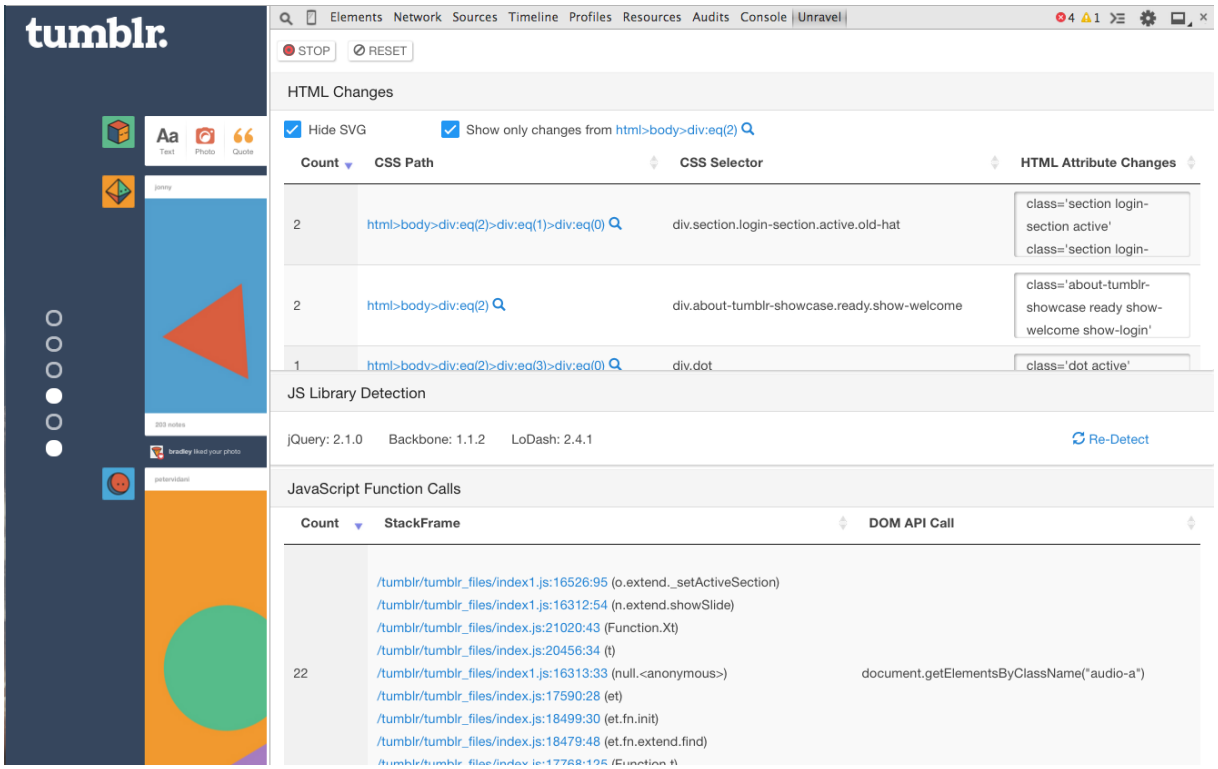

Figure 1.1. The Unravel recording interface consists of the HTML Changes pane (top), the JS Library Detection pane (middle), and the JavaScript Function Calls pane (bottom). Unravel has two controls for recording/stopping and resetting change detection (top). The HTML Changes pane shows the total count, CSS Path, CSS Selector, and HTML Attribute Changes for each element change. The JS Library Detection pane shows libraries detected. The JavaScript Function Calls pane shows the total count, stack frames, and DOM API call for each JavaScript call-stack recorded.

aggregates changes monitored from within a website and provides affordances to reduce, scope, and sort observations. Sources most frequently called become obvious choices for the user to examine. Complex UI features can invoke an enormous number of method calls and HTML changes. Navigating unstructured lists of change events for inspection is counterproductive. Unravel aggregates similar JavaScript call-stacks and HTML changes, increasing counts with each occurrence. While users frequently repeat an interaction to see

its effect, Unravel's change panels are continually sorted by highest counts first, bubbling the most changed element or most called trace to the top.

Unravel is supported technically by a novel observation agent that deploys an API harness for observing and recording UI interactions from within a website. The API harness is an approach for monitoring an application's interaction with an API through a removable recording adapter placed between the application and the API. Unravel's observation agent publishes HTML changes and uses the API harness to monitor calls to the document API. While previous work was able to record and replay events, these solutions depended on access to a remote debugging API. Unravel's observation architecture only depends on native JavaScript and HTML, widening its application domain to other UI inspection toolkits.

Unravel was evaluated with 13 web developers on 5 large-scale websites. The results included a 53% decrease in time to discovering the first key source behind a UI feature and a 32% decrease in time to understanding how to fully recreate a feature. In summary, Unravel can be applied to help developers find entry points into complex code quickly with lowered barriers of entry.

1.2.2. Telescope: Low-Barrier Learning Materials from Runtime Code

Telescope is a platform that supports the discovery of website feature implementation by allowing the user to fine-tune a composite view of responsible JavaScript and explore visual links between JavaScript, HTML, and rendered UI components (see Figure 1.2). Telescope helps users generate low-barrier learning materials — less than two hundred lines of code — from tens of thousands of lines of complex website code. For example, a

curious user could discover how an interactive map component achieves its dragging effect in JavaScript and HTML by setting Telescope's JavaScript detail level to minimum (DOM-modifiers only) and time constraints before and after the click-and-drag. By clicking call and query markers in the interface, visual lines connect JavaScript methods to queried DOM elements, and corresponding DOM components are highlighted in the website.

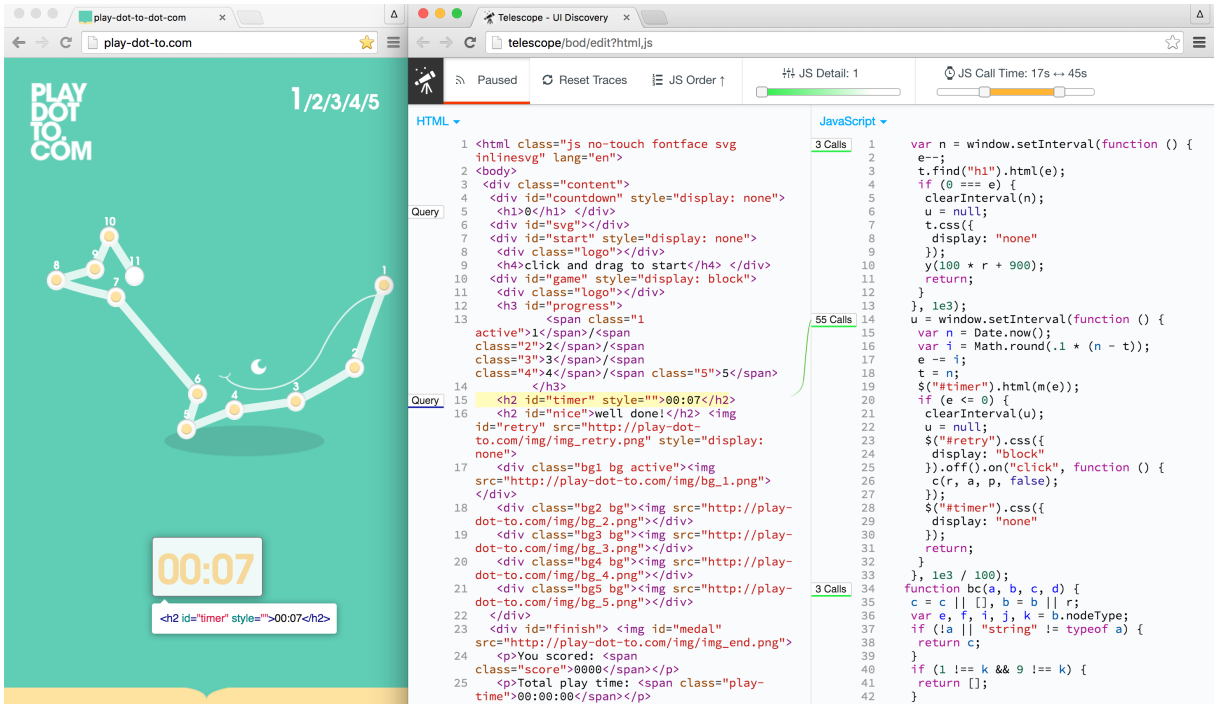

Figure 1.2. The Telescope interface is being used to discover how this HTML5 connect-the-dot game's timer works. The interface is paused to freeze the current view. The detail level is set at minimum, and the JavaScript call time is constrained between the 17th and 45th second of execution. The left Telescope panel (middle) shows a filtered HTML view, where an active element is highlighted and query markers denote that JavaScript queried those lines during the chosen time window. The right Telescope panel shows the website's JavaScript, filtered by time and detail. With the current settings, only the most relevant JavaScript is displayed: active non-library JavaScript which queried the DOM in the constrained time frame. A curved line is drawn to connect the JavaScript line to its DOM query.

Telescope surfaced hidden links beyond Unravel with its two-way visuals between HTML and JavaScript (extending claim 1). Telescope demonstrated the necessity of lowering additional information barriers while allowing for personalized exploration (extending claim 2). Further, Telescope provided a foundational architecture for scaling the conversion of examples into learning resources without additional authorship or maintenance (broadening claim 4).

The conceptual contribution of Telescope is the idea of *helping users understand complex website code by generating low-barrier learning materials.* Telescope introduces three design principles to support this idea:

(1) *Single Composite View*: As a user interacts with a website, Telescope brings together relevant JavaScript for an interaction into a single, composite JavaScript view to resolve the challenges in finding all code relevant to a behavior in unfamiliar code **[35]**. Users can easily hide sources they deem irrelevant or alter the display order of script sources relative to their dependency load order.

(2) *Detail and Time Controls*: The user can scope relevant Javascript by call time and control the amount of detail they wish to see, ranging from showing non-library DOM-modifying code only to showing all JavaScript present in the website. These controls address a critical need discovered through our human-centered design process, when we found users struggling to understand the code for an interaction when there is either too little or too much JavaScript to analyze.

(3) *Visual Links*: Visual links connect active JavaScript to lines of HTML and website DOM components to expose end-to-end functionality.

The technical contributions supporting Telescope enable its ability to examine website UI interactions across the open web in real time. They include: (a) the *Wisat architecture*, which supports source code tracing and instrumentation as well as shared Telescope sessions on public websites and (b) the *Sleight-of-Hand method* (SoH), which swaps a website's client-side implementation during runtime with its instrumented counterpart. The SoH method transitions websites from a non-traceable state to a fully instrumented state, supporting live interaction traces as a user interacts with their website. The Wisat architecture then transmits runtime traces used to decide which JavaScript is displayed in Telescope's composite view and provides the linking data necessary for drawing connections between JavaScript, HTML, and website components. The third system, Isopleth, extends this architecture to enable mixed initiative sensemaking across the open web.

In a case study across seven popular websites, Telescope helped identify less than 150 lines of front-end code out of tens of thousands that accurately described the desired interaction in six of the sites. In an exploratory user study, users discovered difficult unfamiliar programming concepts by leveraging Telescope's ability to reduce code while varying its detail display to provide deeper understanding when needed. In summary, Telescope can be applied to produce low-barrier learning materials for users to discover an accurate glimpse of how a website feature works, with support for them to explore much further.

1.2.3. Isopleth: Mixed-Initiative Sensemaking in Unfamiliar Code

Isopleth is a web-based platform that enables a mixed-initiative sensemaking process by combining system and user-generated facets and source code alterations to support

learners as they make sense of complex JavaScript features in professional websites (see Figure 1.3). Isopleth automatically identifies programming *facets*, or code constructs that are defined by their inputs and outputs, then exposes functional relationships and hidden asynchronous relationships in its call graph. In contrast to existing systems, our goal is to help users identify meaningful entry points into complex code and then interactively explore, label, and filter facets to produce their own understanding of its functionality. Isopleth supports mixed-initiative interaction by integrating newly created facets and labels into recalculated views. For example, a user could explore how autocomplete works by (1) selecting the "Keyboard" facet, (2) creating a new facet filter for the text of their autocomplete query, and (3) following asynchronous links between keyboard-related invocations and query-related invocations. Toward RALE, Isopleth surfaces hidden relationships and facets in the code (extending claim 1), minimizes additional coordination, design, and use learning barriers (extending claim 2), and is the first system to scaffold mixed-initiative sensemaking (founding claim 3).

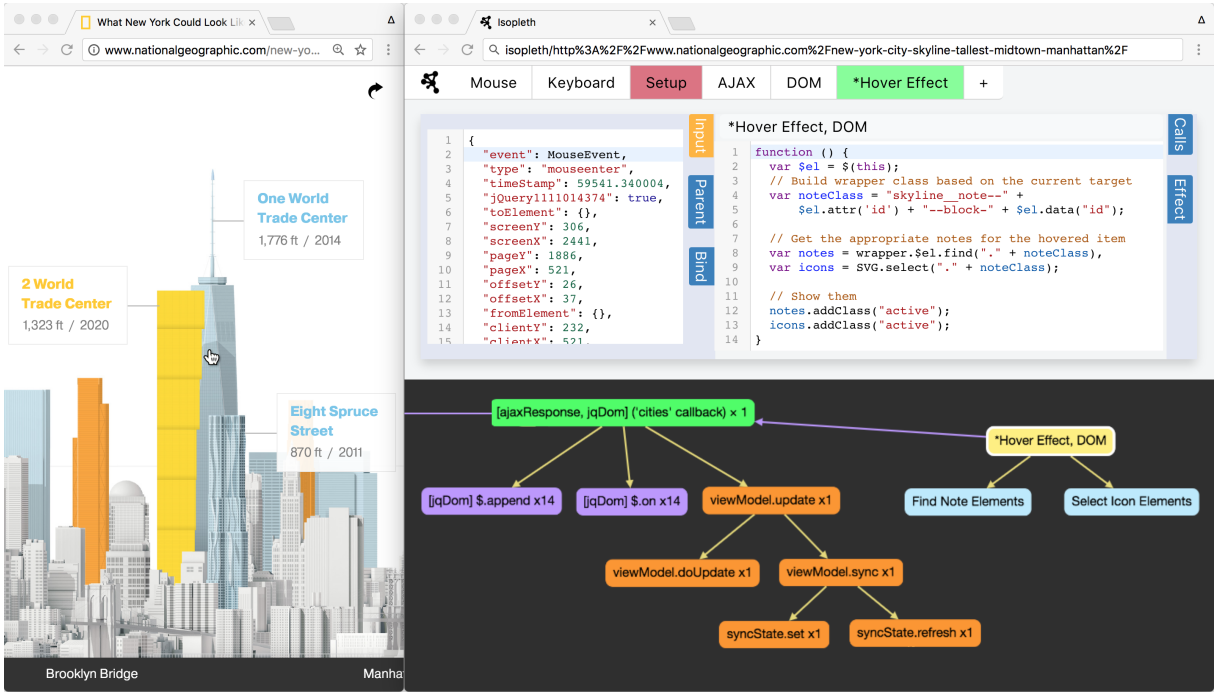

Figure 1.3. A learner is using Isopleth to understand JavaScript code constructs related to moving and scrolling their mouse on National Geographic's New New York Skyline article. Isopleth opened in a new window after the user activated it on the website; it continuously updates with JavaScript activity. Facet filters (top) are used to filter display based on facet, or code constructs defined by their inputs and outputs. Source frame views (middle) display specific function invocation states in the runtime with their inputs and outputs, parent and child calls, asynchronous declaration context, asynchronous binding, and asynchronous effect if present. The condensed call graph (bottom) displays a collated, filtered, labeled, and color-coded JavaScript runtime call graph (See Figure 5.2). Users can apply `or` and `not` operators on the filters by left and right clicking, respectively. To support mixed-initiative sensemaking, users can add custom filters (See Figure 5.3), modify source frame and graph node labels, and add commentary in source code — the system reacts by integrating learner input into its views. Pictured here, a user has added a custom "Hover Effect", altered source code, and updated node labels to make sense of smaller call trees.

The conceptual contribution of Isopleth is the idea of *scaffolding sensemaking of complex professional code by surfacing hidden relationships between code constructs and providing a mixed-initiative process to interactively explore, label, and identify system components and how they relate.* Distinct from function call filters, code detail levels, or web feature-location, Isopleth leverages automated techniques to surface facets, code construct defined by inputs and outputs, and expose hidden asynchronous relationships among function invocations. Users can engage in sensemaking by editing code, rearranging invocations, and composing their own invocation labels. As users explore connections between facets and code constructs, they can define new facet filters for the system to respond with newly surfaced facets and hidden relationships. By modifying and contributing filters to the system, users participate in a mixed-initiative flow, where the user and the system work together to support the user's goal.

The technical contribution of Isopleth is *a Serialized Deanonymization (SD) technique that places unique identifiers in all functions in a web application's JavaScript source to trace how functions are bound, passed, returned, and invoked asynchronously.* This technique provides the ability to take users beyond a UI feature's binding to show them how the feature's bindings was created. Related toolkits [**53, 34**] are limited to linking a function invocation to its declaration context, and therefore cannot expose where the function was bound, passed, or set as a callback. For example, imagine a web application that creates an anonymous function at line 13 of a source file and binds it as a click handler at line 93. When a user clicks, existing tools would point to line 13 and neglect line 93, thereby making it difficult for the learner to see how the function was used. SD provides missing links such as these by adding them to the call graph. This allows us

to see a complete picture of code activity between declaration and invocation, and thus surfaces crucial information for understanding how web features are implemented.

In a case study across 12 popular websites with rich user experiences, Isopleth supported sensemaking and the discovery of 20 different design patterns. It also surfaced common and distinct implementation approaches used across the 12 websites. In a user study with 14 participants, Isopleth was evaluated to analyze its ability in scaffolding sensemaking and informing conceptual models. Users with basic working knowledge of website programming were able to discover and explain design patterns they found in unfamiliar complex professional website code. Most users described either a newly formed or richly extended mental models of a website feature's architecture. In summary, Isopleth was the first system to enable mixed initiative sensemaking for users seeking a deeper understanding of web feature implementation techniques.

1.2.4. Thesis Statement

Readily Available Learning Experiences on the open web surface hidden design patterns, code constructs, and relationships in professional websites to provide opportunities for inexperienced web developers to overcome learning barriers in unfamiliar professional website code, engage in structured sensemaking with complex artifacts, and bridge essential gaps in knowledge of developing professionally designed systems.

1.3. Thesis Overview

- Chapter 2 establishes related work and presents how the contributions of this thesis both address and advance current research.

- Chapter 3 presents Unravel, the first system toward RALE, which aids the reverse engineering of websites by providing comprehensive yet targeted views of JavaScript invocations, HTML changes, and included libraries.
- Chapter 4 presents Telescope, the second system toward RALE, which supports the discovery of website feature implementation by allowing users to fine-tune a composite view of responsible JavaScript and explore visual links between JavaScript, HTML, and rendered UI components.
- Chapter 5 presents Isopleth, the third system toward RALE, which enables a mixed-initiative sensemaking process by combining system and user-generated content to support learners as they make sense of complex JavaScript features in professional websites.
- Chapter 6 discusses use cases for each system, reviews the design claims and evidence for RALE, and looks into how applications of RALE could be created for other domains.
- Finally, Chapter 7 reviews the contributions of this thesis and proposes future research for furthering the RALE vision.

CHAPTER 2

Related Work

This thesis extends and contributes to four main bodies of work around the central claims of RALE: (1) Surface hidden design patterns, code constructs, and relationships (both direct and indirect) from professional websites. (2) Minimize learning barriers while supporting personalized exploration of unfamiliar website code. (3) Scaffold mixed-initiative sensemaking to help users walk through unfamiliar complexities in the surfaced resources. (4) Scale the conversion of examples into learning resources without additional authorship or maintenance. We detail below the related work around each of these claims and what is necessary beyond prior work, so as to situate the contributions of this thesis.

2.1. Surfacing Information from Code

RALE addresses a class of users who are both frustrated by their knowledge gaps in web development and limited by their ability to analyze complex professional code. Surfacing patterns, constructs, and relationships from professional websites helps users overcome gaps in knowledge, shortcuts inefficient forms of web foraging such as tutorials and Q&A, and provides them with opportunities to learn authentically — in a personally meaningful way using multiple modes of the discipline (i.e. web application programming). This section details prior works in surfacing information from source code.

2.1.1. Theseus, FireCrystal, and Scry

Systems including Theseus [**53**], FireCrystal [**67**], and Scry [**15**] provide techniques and affordances to surface JavaScript, HTML, and CSS feature source code. This thesis both addresses limitations and extends contributions from these works.

Theseus instruments and visualizes runtime information about JavaScript execution in a live editing environment [**53**]. Theseus was designed to address a programmer's misconceptions by drawing attention to similarities and differences between the programmer's idea of what code does and what it actually does. Theseus contributes an underlying framework called Fondue to instrument JavaScript sources for tracing, of which the Telescope (see Chapter 4) and Isopleth (see Chapter 5) directly extend. While Theseus helps developers address misconceptions and visualize execution, it lacks affordances to help users address additional learning barriers such as differentiating relevance in JavaScript, drawing links to interaction with the DOM, or exposing hidden relational links between facets in JavaScript. This thesis contributes these additional affordances while extending Theseus' Fondue instrumentation techniques to support RALE on the open web.

FireCrystal allows users to record an interaction with a website and to replay the interaction with highlighting over sources that are active at each point in time [**67**]. This allows users to find specific JavaScript that ran during different frames of a UI recording. However, rich UI features can involve thousands of lines of JavaScript across multiple call frames, and FireCrystal's interface requires users to replay through interactions to discover relevant sources via linear search, which becomes tedious and time-consuming at professional scale (i.e. thousands of lines of code). Like Theseus, FireCrystal's design goal of visualizing execution through replay is limited to highlighting active JavaScript while

additional barriers in unfamiliar code inhibit ongoing discovery. However, FireCrystal's contribution is a foundational component of the RALE vision.

Scry exposes program state and provides a timeline visualization for users to explore how state changes in response to JavaScript calls [**15**]. This allows users to find specific JavaScript, HTML, and CSS involved in changing the DOM at a specific point or frame in time. However, Scry adopts sequential workflows that require back-and-forth navigation from the interface to individual JavaScript files and limits its observation scope to JavaScript that interacted with the DOM. Further, Scry only provides one-way links from a DOM change to the JavaScript that operated on it. Supporting opportunistic discovery in RALE, this thesis contributes design characteristics such as two-way DOM-JavaScript inspection (see Chapter 4) and extensible facet filters (see Chapter 5) to view JavaScript in more ways than its relative proximity to querying the DOM.

2.1.2. Visual Tools and Web Inspectors

Prior tools in source code visualization were mainly designed to help experienced developers surface debugging information and enable inspection to solve problems in their applications efficiently. This thesis extends and applies techniques from these works to help inexperienced developers discover design patterns and constructs supporting web UI features. Gliimpse provides animated transitions between the rendered UI and its source; Telescope (Chapter 4) extends this idea to link JavaScript to its queried elements in a browser's web page. Mimic and Theseus log analytics and invocation counts from recordings of UI interactions; Unravel (Chapter 3) and Telescope extend this idea to help users differentiate which lines of code were more active than others. Telescope and Isopleth

(Chapter 5) surface calls to library and REST API's, building from ideas in RESTful service extraction [**88**]. Beyond this body of work, this thesis contributes filterable views of function invocations, compositing relevant JavaScript and HTML together with visual links, and visualizing the JavaScript call graph with invocations, relations, and labels.

Web developer tools included in major browsers help experienced developers build and debug their applications but can be overwhelming for learners trying to gain insight into professional development patterns. Chrome Developer Tools (CDT), Firebug, and Safari Web Inspector provide rich suites of tools for debugging, inspecting, and live-editing methods. DOM breakpoints trigger direct navigation to responsible JavaScript methods when elements in the DOM are dynamically changed [**5, 45, 43**], which provides an entry point into searching for the logic responsible for updating the DOM. While these inspectors offer robust debugging workflows, the primary limitation of these tools emerges as a barrier to learning, where too much source code obscures how constructs of the language are used to achieve an effect. Complex websites often have thousands of lines of JavaScript bundled in minified form for optimized transfer. Beyond overwhelming amounts of code, these inspectors are designed to support debugging workflows instead of opportunistic discovery. Using the tools to gain professional experience involves tediously following long call chains and reverse engineering complex asynchronous event-binding networks. The event-driven asynchronous nature of JavaScript obscures how logical components are related to react to the UI from the user's point of view. This thesis both augments and extends the CDT environment to transform websites into learning materials.

2.1.3. Surfacing Call Graphs

Computing a well-defined call graph of a program's internal constructs, relationships, and function invocations helps construct accurate and comprehensive learning materials in RALE. Prior tools use forms of instrumentation to help inform developers how software works but typically use either static analysis or limited runtime analysis to determine program flow, leaving complex asynchronous operations unobserved. The contributions in Chapter 4, Telescope, rely on Fondue's [**53**] ability to surface an accurate call graph. Telescope traverses the call graph to calculate which sources to display in its composite view (e.g. library code, invoked-JavaScript, and DOM-querying JavaScript).

The call graph contributions in Chapter 5, Isopleth, support advanced calculations used in its scaffolded sensemaking affordances. Isopleth sits in a unique space, focusing on dynamic asynchronous call-graph navigation for comprehension. Its call graph calculation is inspired by Lencevicius et al's work on query-based debugging [**52**], but instead of requiring users to query traces, Isopleth's facets provide low-effort ways to reshape the graph while providing a simple abstraction for defining new facets. WhyLine's tracing techniques [**48**] are similar to Isopleth's serialized deanonymization technique, but Whyline cannot capture the important asynchronous connections often present in web applications. Isopleth uses the ID's from each method to backtrace complex asynchronous relationships, then provides affordances to visualize the complex relational links.

2.2. Minimizing Learning Barriers

Programmers often experience barriers in learning new programming concepts, and a primary claim of RALE is to minimize information barriers while supporting personalized

exploration of professional website code. Specifically, the three systems in this thesis help users overcome three of Ko et al's Six Learning Barriers: Design, Coordination, and Information [**49**].

The Design barrier involves the inherent cognitive difficulties of solving a programming problem, the Coordination barrier involves determining how to combine constructs or technologies to achieve an outcome, and the Information barrier involves knowing where to look for clues about a program's internal behavior. In Chapter 3, Unravel contributes a technique to overcome the Information barrier by filtering and promoting relevant JavaScript and HTML to inspect. In Chapter 4, Telescope provides a way to overcome the JavaScript/HTML coordination barrier by visually linking JavaScript and HTML if they are logically connected. In Chapter 5, Isopleth helps users overcome the design barrier by providing mixed-initiative scaffolds to help make sense of complex feature implementations.

Further, this thesis explores strategies, interfaces, and methods for developers to quickly and easily overcome barriers caused by the unfamiliarity of code. This includes mental barriers that Gross and Kelleher describe such as memory failure, method interpretation, and lack of temporal reasoning [**35**]. Developers trying to overcome these barriers currently turn to web foraging for speed and ease in finding help [**9**] but can become frustrated with outdated or incomplete results. Following Gross and Kelleher's guidance on designing systems to identify functionality in unfamiliar code, we designed Telescope to "connect code to observable output" and "provide interactions to fully navigate code" (see Chapter 4). The event-based, asynchronous, and often overly-complex

nature of JavaScript implementations [**2**] increases these barriers, further discouraging those trying to learn from JavaScript source.

2.2.1. Visual Techniques to Overcome Barriers

Visual learning techniques help users to easily see the dynamic effects of their code to understand the properties of a program's external behavior relative to its internal constructs. For example, Gliimpse [**24**] provides animated real-time visuals that transition between markup languages and their rendered output. Users can see their code transform into its visual rendering as they modify it. PyTutor [**37**] provides users with a simple Python interface while showing them a visual display of internal program state and operations as they modify code in the interface. Bret Victor's "Learnable Programming" [**89**] allows users to interact with a running program, modify its underlying code, and see the effects of modifications as they are made. Most modern web browsers provide affordances for users to find responsible source code through visual breakpoints in the DOM [**5**], JavaScript beautifiers [**31**], and HTML change highlights [**32**]. This thesis addresses remaining difficulties in forming accurate mental models of design patterns and techniques used to create professional web code, such as breaking down complex function relationships, visualizing JavaScript operations on the DOM, or minimizing the overwhelming effects of large volumes of code.

2.2.2. Interactive Techniques to Overcome Barriers

Existing tools contribute design techniques to highlight, filter, and curate parts of code responsible for an effect. Theseus [**53**] provides "hit-counts" and detailed call stack logs

for lines of JavaScript in an extended Brackets IDE. FireCrystal [**67**] records UI interactions and plays them back in a view coupled with a JavaScript inspector that highlights active lines per each frame in time. Scry [**15**] extends the Safari browser to record UI interactions, then provides a timeline view where users can view DOM state changes and the JavaScript/CSS trace involved in each change. Clematis [**2**] visualizes episodes of cause-effect JavaScript events and their effect on DOM state through an expandable timeline view. Tutorons [**38**] automatically generates context-relevant, on-demand micro-explanations of code in an editor. Gidget [**51**] interactively coaches players learn programming by working with a bot named Gidget to debug problematic code. WebCrystal [**17**] gives users a way to quickly access HTML and CSS information from a webpage by selecting questions regarding how a selected element is designed. Whyline [**47**] allows users to ask questions about a runtime then visualizes answers in terms of runtime events directly relevant to a programmer's question. Dinah [**36**] supports beginners in selected code causing graphical output through statement replay and temporal navigation.

2.3. Scaffolding Mixed-Initiative Sensemaking

The third primary claim of RALE is that it should scaffold mixed-initiative sensemaking to help users walk through unfamiliar complexities in the surfaced resources. This calls for techniques and affordances to support learners during their sensemaking process while providing them with cues to engage in multiple modes of the web programming discipline such as architecture, implementation, and refactoring.

2.3.1. Sensemaking and Learning Scaffolds

In order to build new understanding from a programming example, a learner must first make sense of the code structure and functionality. In the learning sciences, sensemaking refers to the process of understanding a new example or artifact by generating representations that explain what is known or understood [**91, 72**]. While experts leverage templates and formal representations of programming constructs to make sense of and solve problems, these patterns are not apparent to beginners [**1, 92, 57, 62, 18, 19, 22**]. Furthermore, learning from complex examples requires understanding not only the individual components, but also how they coordinate to solve a problem [**49**]. This requires both conceptual knowledge and expert strategies for constructing an understanding of a problem by examining evidence, testing hypotheses, and reflecting on findings [**95, 94**].

The learning sciences provide guidelines for scaffolds in RALE (supports and affordances) that can help beginners bridge this knowledge gap to make sense of complex examples. This literature suggests that tools designed to support sensemaking should build on learners' intuitive understanding by using representations and language that connect to their knowledge [**56, 72**]. Tools should also be organized around the semantics of the discipline [**72**] and provide opportunities for learners to reason about the structure of code, and not just how it works [**70**]. Finally, tools should provide opportunities for learners to inspect professional code in different ways [**72**]. Providing multiple ways to visualize the code helps learners build dense, interconnected conceptual representations [**10, 78, 3**]. In Chapter 5, these learning goals are implemented in Isopleth to enable RALE for professional websites.

2.3.2. Program Comprehension

The design of RALE extends a rich body of research in computer program comprehension by contributing new interaction techniques to code comprehension. Specifically, this body of work examines how programmers support cognitive tasks such as thinking and reasoning about the structure of code [**87, 13, 85, 90, 83, 69**]. Several comprehension theories classify how programmers understand new code, such as (1) top-down from domain to source code [**13**], (2) bottom-up from statements to abstractions [**83**], (3) beacons from familiar code with plan decomposition in unfamiliar code [**85**], and (4) bottom-up through control-flow abstraction from microstructrues to macrostructures to form a situational model [**69**]. Most similar to Pennington's theory [**69**], detailed in Chapter 5, Isopleth allows users to navigate and filter pre-labeled microstructures and follow control-flow through relational links in a graph. Isopleth's sensemaking affordances aid the formation of Pennington's macrostructures by breaking down complex website logic into consumable pieces and allowing users to explore and modify these pieces in a personally meaningful way. Similar to Soloway's theory [**85**], Telescope aids program comprehension by providing learners with visual line connectors between modified DOM elements and active JavaScript functions (see Chapter 4). Telescope's visual line affordances draw a learner's attention to certain code constructs in HTML and JavaScript, acting like Soloway's beacons for discovery.

2.4. Scaling Learning Resources

With millions of active learners looking to the web for online learning and the fast pace of innovation in web programming, teachers and content authors meet only a small

portion of the ever-expanding demand for learning materials. The primary goal of RALE is to transform inspiring professionally-produced websites into opportunities for learning with no dependencies on authoring. This section details related work around RALE's fourth claim to scale the conversion of examples into learning resources without additional authorship or maintenance.

2.4.1. Technical Dependencies

Chapter 3 and Chapter 4 introduce the API Harness and Wisat architecture to capture JavaScript functionality on the open web, but unlike related methods they require few dependencies or user installation to scale. The Mozilla Remote Debugging Protocol [**64**] allows developers to access the JavaScript event loop and observe execution, but the Mozilla RDP does not store a full history of function invocations, which is necessary for call graph calculation in Telescope and Isopleth. Lieber et al's Fondue instruments all functions in the JavaScript source to monitor execution, but requires users to override their system settings to allow a proxy server to intercept website sources [**53**]. While Unravel is limited in its abilities to surface runtime information (i.e. DOM-modifying JavaScript only), a core design characteristic of Unravel is to be lightweight and immediately usable on public websites. Similar to Unravel, Maras et al's source extraction technique and Burg et al's Scry offer limited views of JavaScript based on static analysis of dependencies and DOM-querying JavaScript, respectively [**60, 15**]. Alternatively, the Wisat architecture surfaces complete runtime information about all JavaScript functions via one-click user activation to provide accurate scaffolds and fully comprehensive views into source code (used in Chapters 4 and 5).

2.4.2. Extending Web Foraging

One set of tools has been designed to support professional developers as they make sense of complex code using resources provided continuously via the web. Brandt et al explored how programmers leverage online resources to support the development process, opportunistically transitioning between web foraging, learning, and writing code [**9**]. They built on this work to develop Blueprint, a web search interface integrated into a development environment to support searching for relevant code examples efficiently [**8**]. Fast and Bernstein designed Meta, a Python language extension that allows programmers to share and compare their implementation approaches and provides recommendations for improvements based on crowd data [**26**]. These approaches effectively support opportunistic sensemaking by leveraging context and online resources to provide relevant programming scaffolds. However, they are designed to support expert developers; they do not provide affordances to build on beginners' understandings [**56**] or help beginners reason about the structure of code [**70**].

CHAPTER 3

Unravel: Rapid Web Application Reverse Engineering

This chapter presents the first application in RALE, the Unravel system, which provides techniques in overcoming challenges in finding relevant HTML and JavaScript code for a UI feature in a complex professional website. This chapter has adapted, updated, and rewritten content from a paper at User Interfaces Systems and Technology 2015 [**39**]. The source code for Unravel is openly available [1]. All uses of "we", "our", and "us" in this chapter refer to coauthors of the aforementioned paper.

Professional websites with complex UI features provide real world examples for developers to learn from. Yet despite the availability of source code, it is still difficult to understand how these features are implemented. Existing tools such as the Chrome Developer Tools and Firebug offer debugging and inspection, but reverse engineering is still a time consuming task. We thus present Unravel, an extension of the Chrome Developer Tools for quickly tracking and visualizing HTML changes, JavaScript method calls, and JavaScript libraries. Unravel injects an observation agent into websites to monitor DOM interactions in real-time without functional interference or external dependencies. To manage potentially large observations of events, the Unravel UI provides affordances to reduce, sort, and scope observations. Testing Unravel with 13 web developers on 5 large-scale websites, we found a 53% decrease in time to discovering the first key source

[1] Unravel Github `https://github.com/NUDelta/Unravel`

behind a UI feature and a 32% decrease in time to understanding how to fully recreate a feature.

3.1. Motivation and Contributions

Developers can learn from professional websites, but the barriers to understanding unfamiliar code [**49**] hinder the potential for authentic learning [**81**]. Without documentation for UI features of complex websites, one must search for curated examples or attempt to reverse engineer the website to discover how a feature works. Examples may not be available for unique features or may only provide partial solutions. Professional websites combine many web technologies to present unified interfaces that are not straightforward to disassemble. Reverse engineering UI components such as a photo carousel, search autocomplete, or table filter is difficult, because it involves cyclical HTML inspections to follow element changes and find-all queries in JavaScript files for references to DOM elements. JavaScript often executes asynchronously out of order, making it difficult to identify which lines of JavaScript to start examining [**2**].

Current approaches including record & replay in the DOM and JavaScript tracing have inspired this chapter (e.g. [**5, 12, 14, 24, 67**]) as they showed that recording and tracing changes in-context gives developers a better understanding of what's happening in the source code [**14, 36**]. With some context clues about where to begin looking, junior developers are more likely to overcome barriers that would otherwise prevent them from beginning a first attempt at reverse engineering [**49**]. But beyond source exposition, existing tools lack affordances to show the most relevant lines of source code. Complex features may consist of hundreds or thousands of recorded function invocations; without

additional affordances, the inefficient process of searching, inspecting, and debugging to gain understanding is tedious and time-consuming.

Unravel aids the reverse engineering of websites by providing comprehensive yet targeted views of JavaScript invocations, HTML changes, and included libraries (see Figure 3.1). Unravel enhances Chrome's existing developer toolkit by linking all HTML and JavaScript components to their corresponding inspection panes for quick examination. Unravel works on all websites without interfering with existing functionality. For example, a developer can navigate to a landing page, record a parallax effect, and watch Unravel identify which lines of JavaScript were executed, which DOM elements were modified, and which attributes were modified per each element.

The main conceptual contribution of Unravel work is the idea of *tracing, identifying, and organizing the most relevant functions and DOM elements manipulated to support reverse engineering and understanding interactions on complex professional websites*. Unravel aggregates changes monitored from within a website and provides affordances to reduce, scope, and sort observations. As users repeat their desired interaction, call counts related to their feature bubble up, turning relevant sources into obvious choices for the user to examine. Complex UI features can invoke an enormous number of method calls and HTML changes. Navigating unstructured lists of change events for inspection is counterproductive. Unravel aggregates similar JavaScript call-stacks and HTML changes, increasing counts with each occurrence. Unravel's change panels are continually sorted by highest counts first, bubbling the most changed element or most called trace to the top. Affordances are provided to constrain observation scope to specific DOM sub-trees and

throttle large sets of function invocations generated in loops, such as scaling an image on each pixel scrolled.

The fundamental technical contribution of Unravel is *an observation agent that deploys an API harness for observing and recording UI interactions from within a website*. The API harness is an approach for monitoring an application's interaction with an API through a removable recording adapter placed between the application and the API. Unravel's observation agent publishes HTML changes and uses the API harness to monitor calls to the document API. While previous work was able to record and replay events, these solutions depended on access to a remote debugging API. Unravel's observation architecture only depends on native JavaScript and HTML, widening its application domain to other UI inspection toolkits.

In the rest of this chapter, we introduce Unravel and its main components for tracking HTML changes, tracing JavaScript method calls, and identifying libraries. We detail the observation agent and techniques for organizing and presenting trace information; evaluate the benefits of reverse engineering with Unravel; and conclude with a discussion of design principles, limitations of our approach, and a brief look at the next chapter.

3.2. Unravel

Unravel is a Chrome Developer Tools extension that provides affordances for discovering and navigating relevant UI source code through three main activities: recording source code activity triggered by a user's interaction with a web page, refining the scope of source code under observation, and linking lines of source code to corresponding inspection and debugging panes for further analysis (see Figure 3.1).

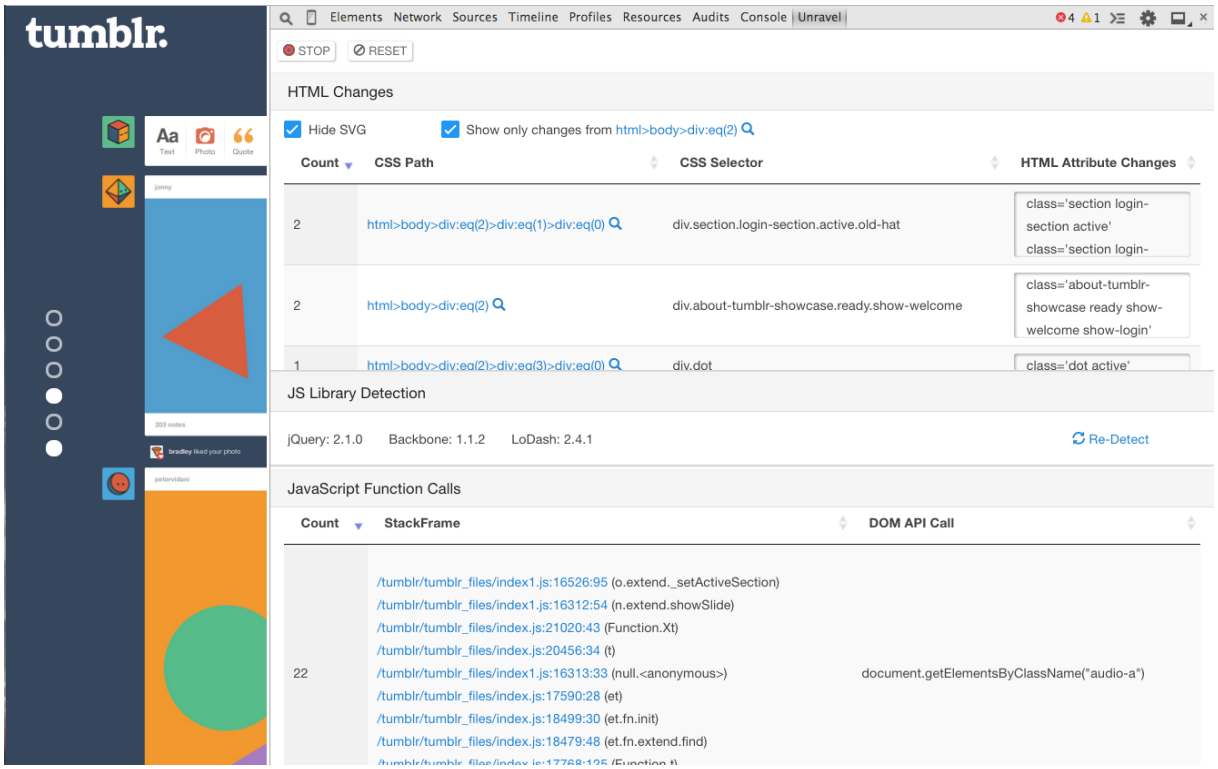

Figure 3.1. The Unravel recording interface consists of the HTML Changes pane (top), the JS Library Detection pane (middle), and the JavaScript Function Calls pane (bottom). Unravel has two controls for recording/stopping and resetting change detection (top). The HTML Changes pane shows the total count, CSS Path, CSS Selector, and HTML Attribute Changes for each element change. The JS Library Detection pane shows libraries detected. The JavaScript Function Calls pane shows the total count, stack frames, and DOM API call for each JavaScript call-stack recorded.

3.2.1. Unravel Feature Design

To inform the design of Unravel, we conducted a small exploratory study that observed the existing approaches for reverse engineering web pages. The study consisted of two senior and two junior developers for 20 minutes each, who were asked to reconstruct an animated feature from Tumblr on their own page. We observed participants repeating the animation frequently while inspecting the HTML to see changes. We watched participants slowly

scan through numerous JavaScript files to discover source code causing the animation. One participant said, "I just want to know how they achieved the effect, but it's not entirely clear from the web inspector."

Unravel's features were designed to help address frustrations and inefficiencies expressed by the test participants. The Unravel HTML Changes feature was designed to record and present modifications to lessen repeat behavior (see Figure 3.2). The JS Function Calls feature was designed to capture JavaScript traces with links to executed line numbers in JavaScript files, making it easier to skim active source code (see Figure 3.3). While no inefficiencies were observed related to JavaScript libraries, we noticed many non-native functions appearing in JavaScript traces. We decided to add library detection to inform the user about the presence of frameworks, polyfills, shims, or syntactic sugar (see Figure 3.4). Unravel's three views are presented as one inspection interface to highlight relevant source code supporting a feature.

3.2.2. Tracking HTML & CSS Changes

Without Unravel, current methods for detecting changes in HTML elements involve setting DOM breakpoints or watching for changes in element inspectors. Stepping through hundreds of attribute changes and looking through a DOM tree becomes time consuming. Unravel aims to streamline searches by providing a list of changes instead.

The Unravel extension begins to track HTML changes that occur in the website as a user starts a new recording. With each user interaction in the website, changes are streamed into the Unravel console under the HTML Changes section (see Figure 3.2). A DOM element's attribute and sub-tree modifications are then viewable in list form with

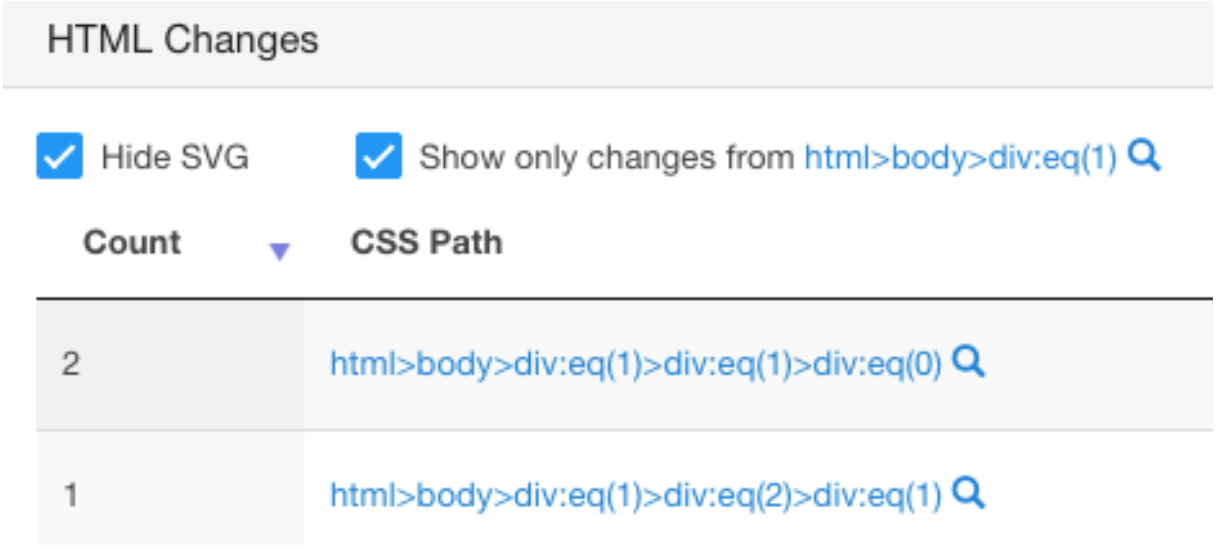

Figure 3.2. Within the Unravel HTML Changes pane, users can opt into hiding changes from SVG elements (top left). Users can constrain Unravel's observation scope by selecting an HTML element to observe (top right).

direct links to structural and CSS inspection in the CDT elements pane (see Figure 3.1). While Unravel does not capture preloaded CSS or CSS pseudo-classes like `:hover`, it monitors CSS class and style changes in HTML attributes such as changing opacity, toggling a class, or modifying WebKit attributes.

3.2.2.1. An Unravel HTML Change Record. Each record in the HTML Changes in the Unravel tool contains:

- Change Count: how many changes were recorded for the HTML element
- CSS Path: a unique CSS selector based on the element's DOM tree location that links to the corresponding node in the CDT Elements Pane
- CSS Selector: a CSS selector based on common query patterns including id, class, and name

- HTML Attribute Changes: a list of changes to the element's attributes ordered oldest first

An example user Alice wishes to discover how a modal window is hidden after clicking an × icon. She clicks *record* in Unravel and watches for changes while clicking the ×. Alice stops the recording and looks at the changes listed in the HTML Changes Pane of Unravel. She notices that the list is presorted by highest count of changes first. The first record shows a div with CSS selector `div#modal`. She clicks on the record to see what it is referencing in the actual website and elements panel. Chrome highlights the element in the panel and in the website. Alice confirms it is her element and examines the attribute changes, listed as `class="modal-front"` followed by `class="modal-hidden"`. Alice learns that removing class modal-front and adding class modal-hidden caused the desired effect.

3.2.3. Tracing JavaScript Method Calls

The bottom panel of Unravel lists JavaScript call-stacks captured while recording (see Figure 3.3). Unravel listens for calls to `window.document` and reports JavaScript traces involved in querying and manipulating the DOM. Every stack frame of each call-stack is linked to its corresponding file and line number in the CDT JavaScript inspector.

Each record in the Unravel JavaScript Changes pane contains:

- Call Count: how many times a call-stack was invoked
- Stack Frame(s): the call-stack leading to a document query
- DOM API Call: which document API method was invoked

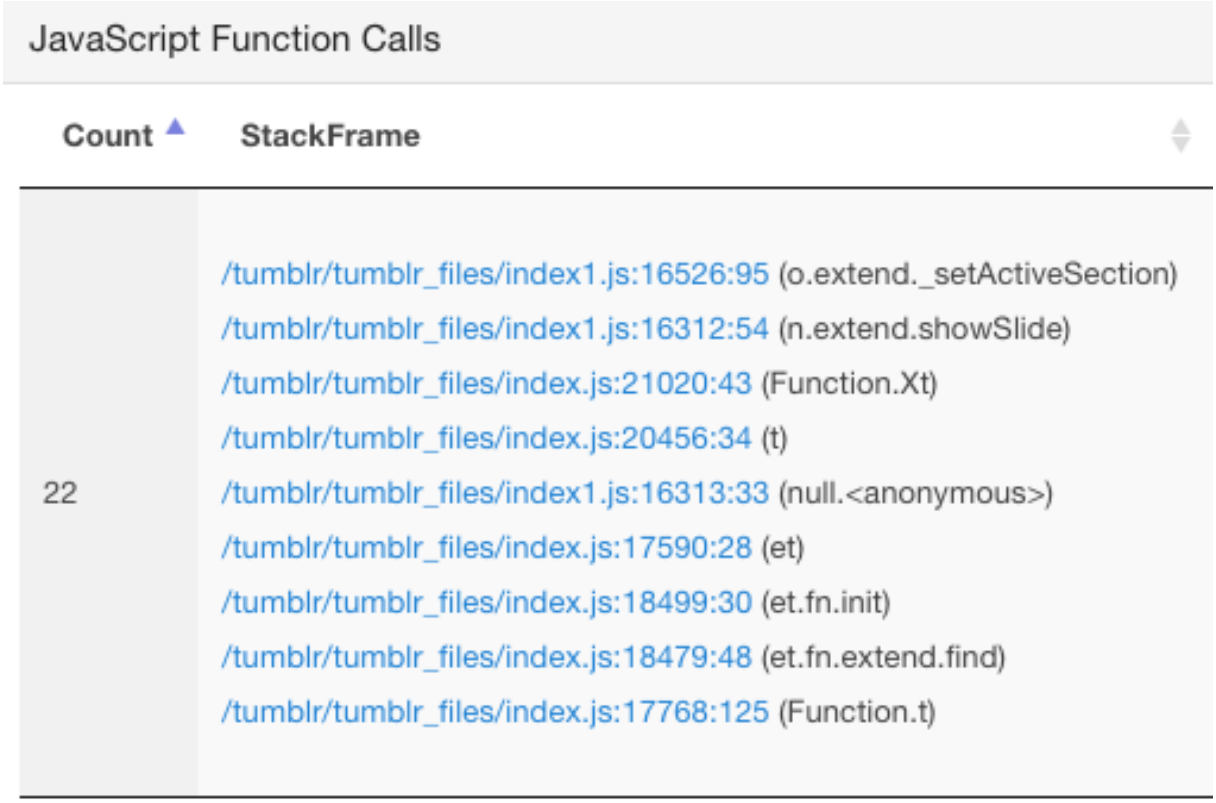

Figure 3.3. The Unravel JavaScript Function Calls pane has captured a call-stack that was executed 22 times. A stack frame with a method called _setActiveSection on object o.extend initiated the call-stack (top), which arrived at a document query for elements with class "audio-a" (shown in Figure 3.1 bottom right).

An example user Carol wishes to better understand how a web application's card-flip effect reveals new data when scrolling down in the interface. Carol initiates a new recording session in Unravel and begins to see stack frames captured in the JavaScript changes pane. Carol stops the recording and notices a call-stack was captured. Carol clicks the first frame in the call-stack and is linked to the CDT JavaScript inspector for `index1.js` at line `16526:95`. She immediately notices a function `_setActiveSection` that contains logic to change the `translate3d` style attribute of a div element. With the first clue,

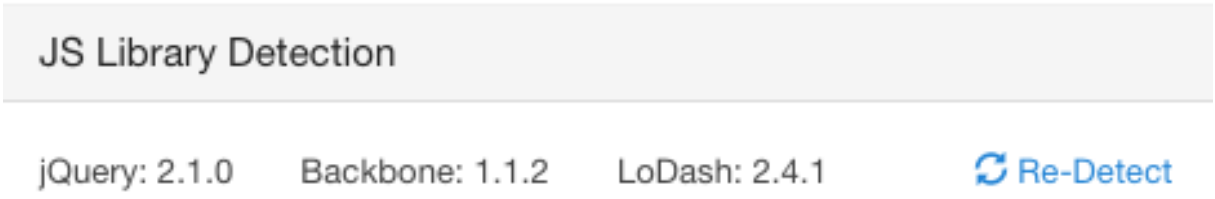

Figure 3.4. The Unravel JS Library Detection pane requests detection for libraries when Unravel starts and as users select re-detect (right). Re-detection is an affordance provided for libraries added after the initial page load. In this figure, jQuery, Backbone, and LoDash were detected.

Carol returns to Unravel to search for how data is loaded. Carol skims the methods and arguments of additional stack frames and finds a method called `fetchCard`. She clicks the stack frame and discovers an XHR request contained a callback that triggered `_setActiveSection`.

3.2.4. Identifying JavaScript Libraries Used

As a precursor to examining source, a list of libraries active in a website prepares the user to understand source in context with the libraries. This may help them to reproduce code for their own use without the frustration of missing libraries. Further, this provides users with clues to how features are implemented using the libraries.

Unravel detects JavaScript Libraries immediately upon launch and lists the libraries with their corresponding versions (see Figure 3.4). An option to re-detect libraries is provided for websites that use a lazy-loading strategy for installing libraries into the application scope.

An example user Bob wishes to discover how a stock-ticker web application easily reformats numbers in many variations. He opens Unravel and finds many sources using

a `numeral()` function. If Bob tried to invoke `numeral` in his own application, he would discover that it is not included in native JavaScript. Using Unravel's library detection, Bob sees that Numeral.js version 1.5.3 is present in the stock-ticker web application. Bob includes the numeral library in his application and is now able to use the same numeral conversion methods as the stock-ticker application.

3.3. Organizing and Tracing Relevant Source Code

3.3.1. Organizing Large Volumes of Trace Information

Complex UI features can generate large volumes of HTML changes and JavaScript traces. Navigating through long lists of changes and traces fails to resolve the *Information Learning Barrier* [**49**], because the program's internal behavior may remain unclear despite a wealth of information. This section discusses four strategies Unravel provides to counter information overload: DOM Tree Scoping, CSS Path Aggregation, SVG Hiding, and Call-Stack Aggregation.

3.3.1.1. DOM Tree Scoping. Without affordances to reduce observation events, simultaneous UI effects can cause confusion. As a user records an interaction, other dynamic behaviors in the application could highlight source code not relevant to the user's interests. After selecting an HTML element to observe, users can opt for Unravel to scope future recordings to a single element and its subtree (see Figure 3.2). With the focus option selected, changes outside the scope of selection will be ignored.

3.3.1.2. CSS Paths and Selectors. HTML changes are recorded and reduced in real-time to the unique DOM tree path of an element. Continuous changes to one element's

attributes are rolled up under a single record in Unravel's HTML change pane (see Figure 3.2). Elements with the most changes bubble to the top. While DOM tree paths can be queried, they can become quite long and difficult to read. Unravel provides simpler selectors by combining the elements tag, id, class, and name if present (see top middle of Figure 3.1).

3.3.1.3. Throttling Repeat Calls. During a preliminary study with an Unravel prototype, we discovered that users were being shown too many irrelevant HTML changes for pages that made use of animations. The users weren't interested in the animation logic itself, but rather DOM elements and interactions surrounding the animation. In the HTML Changes pane, users can select an option to hide superfluous animations (e.g. SVG transitions) (see Figure 3.2).

3.3.1.4. Call-Stack Aggregation. Similar to the HTML Changes feature, JavaScript traces are recorded and reduced by unique call-stack. Continuous calls through the same set of methods are logged by increasing the call-stack count. During our pilot study, we observed users repeating interactions and leveraged this usage pattern to surface relevance in code. As users repeat actions, DOM elements that were changed and functions that were invoked bubble up a sorted list. Users then can sort lists by highest count first with stack frames ordered top-down. All of Unravel's columns are sortable, allowing users to quickly navigate through different perspectives of their recordings.

3.4. Implementation

In building Unravel, we sought to improve upon architecture from related systems to provide a scalable and portable implementation. Systems like FireCrystal and that of

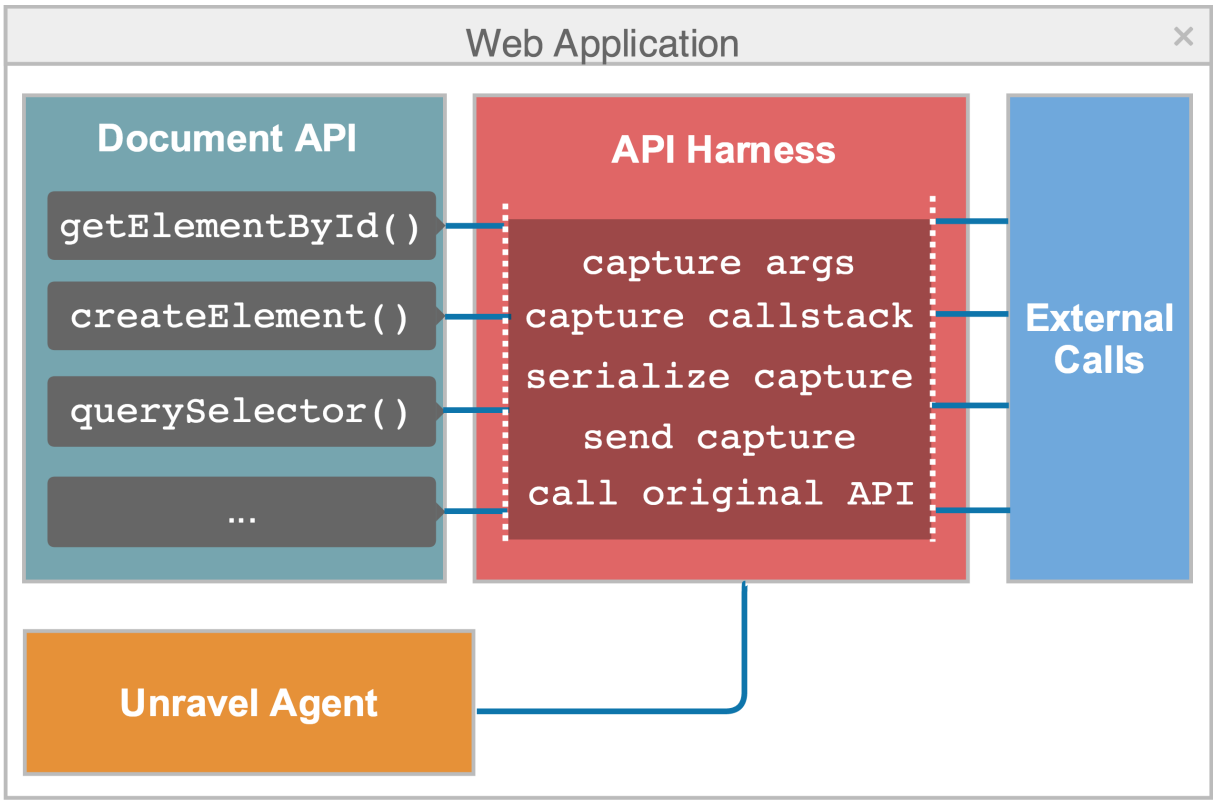

Figure 3.5. An API harness is placed on the document API. Unravel captures and serializes call-stacks and arguments made to the API. Normal interaction with the API resumes after the details of a method call are broadcast.

Maras et al depended on the Firefox Debugging API to query for sources involved behind UI feature [**59, 67**]. Both the scalability and portability of this strategy are limited to the constraints of the Firefox Debugging API. Theseus proposed a global method-wrapping policy for monitoring JavaScript traces that depended on a third-party server to alter sources [**53**]. We strived to build Unravel without any dependencies on external servers or environmental APIs so that it could scale to handling larger UI changes and share a reusable architecture for implementations in other UI toolkits.

3.4.0.1. API Harness. We introduce an API harness as a novel method for monitoring all interactions with an API by placing a removable recording adapter on top of the API. Unravel's agent applies the API harness to monitor call-stacks and arguments to `window.document` when the user begins a recording and removes it when the user stops a recording. By monitoring the document API, we can see the execution route and arguments of functions asking to query and change the DOM (see Figure 3.5). Data from the harness is sorted and reduced prior to appearing in the Unravel's JavaScript Function Calls view. Technical details are discussed in the next section.

3.4.1. API Harness

The API harness is a removable device installed during runtime that captures JavaScript method traces and arguments (see Figure 3.5). The harness implementation is straightforward: for each method in the API, save a reference to the original method and temporarily replace it with a new method that implements the following:

(1) Capture the call-stack invoking an API method.

(2) Capture arguments passed to the API method.

(3) Serialize the captures for transport.

(4) Propagate the capture to subscribers.

(5) Call the original API method with the incoming arguments.

Captures are broadcast from the harness without modification as method calls are made to the API, giving subscribers flexibility in processing the data. Unravel's API harness call-stack captures implement the JavaScript `Error` interface. As each method call is made to the document API, an error object containing a snapshot of the call-stack

is thrown and caught. This snapshot captures comprehensive execution traces from event handlers down to document queries. Unravel reduces and sorts its captures to simplify inspection for the user (discussed earlier). When a recording is finished, the API harness is removed by restoring the original methods to their respective endpoints in the API.

Alternative approaches to implementing an API harness either require external dependencies or aren't designed to monitor program execution. The Mozilla Remote Debugging Protocol [**64**] allows developers to access JavaScript threads and observe their execution but it is only available to extensions of Firefox. Lieber et al's Fondue wraps all functions in the JavaScript source to monitor execution, but exists as a separate proxy server that modifies a web page's JavaScript as it passes through [**53**]. Eagan et al's Scotty enables modification to non-extensible components during runtime, but it does not monitor interactions with those components [**25**].

Engineering trade-offs limit the capabilities of the API harness but give portability to its implementation. The harness must be able to modify public methods of the original API, it must be able to store references to the original method implementations, and it must be able to access callers and arguments. For example, an API harness would not be able to monitor an API reference that was closured in a private variable, because the harness requires public access to API methods. Despite these limitations, the API harness inspects program activity from within a program and operates without external dependencies. With minimal performance overhead, the API harness scales with API demand without causing interference.

3.4.2. HTML Observer and Library Detection

Unravel's HTML observation implements the JavaScript MutationObserver interface. When the observation scope is changed in the Unravel UI, new MutationObservers are created to monitor the corresponding subsections of the DOM tree. As the observers notice events, they are propagated to Unravel's sorting and reduction implementation. When each observation is received, its element's CSS path is calculated by determining the DOM tree location relative to parent and sibling nodes.

The JavaScript libraries are detected by a simple interface detection strategy: for each known library, the Unravel agent tries to invoke published interface methods from the library. We began with Hidayat's try-catch detection strategy [**41**], but extended it as we discovered libraries with identical identifiers and overlapping interface methods such as Underscore.js and lodash.js, both of whom have array methods like `_.reduce()`. If the test is successful, the agent detects the library version and returns the name and version number. There are many JavaScript libraries available, yet there is no published standard for declaring the library name and version from within the library. To detect all JavaScript libraries and display information about them is beyond the scope of Unravel, so we tested Unravel with support for the top 20 JavaScript libraries [**7**].

Website	UI Feature	Trigger
Tumblr	Card Flip View Change	Scroll, Click
Apple iPad	iPad Cover Change	Click
Flickr	Effect Sync to Video	Scroll, Click
Amazon	Product Carousel	Interval, Click
Kickstarter	Photo Carousel	Interval, Click

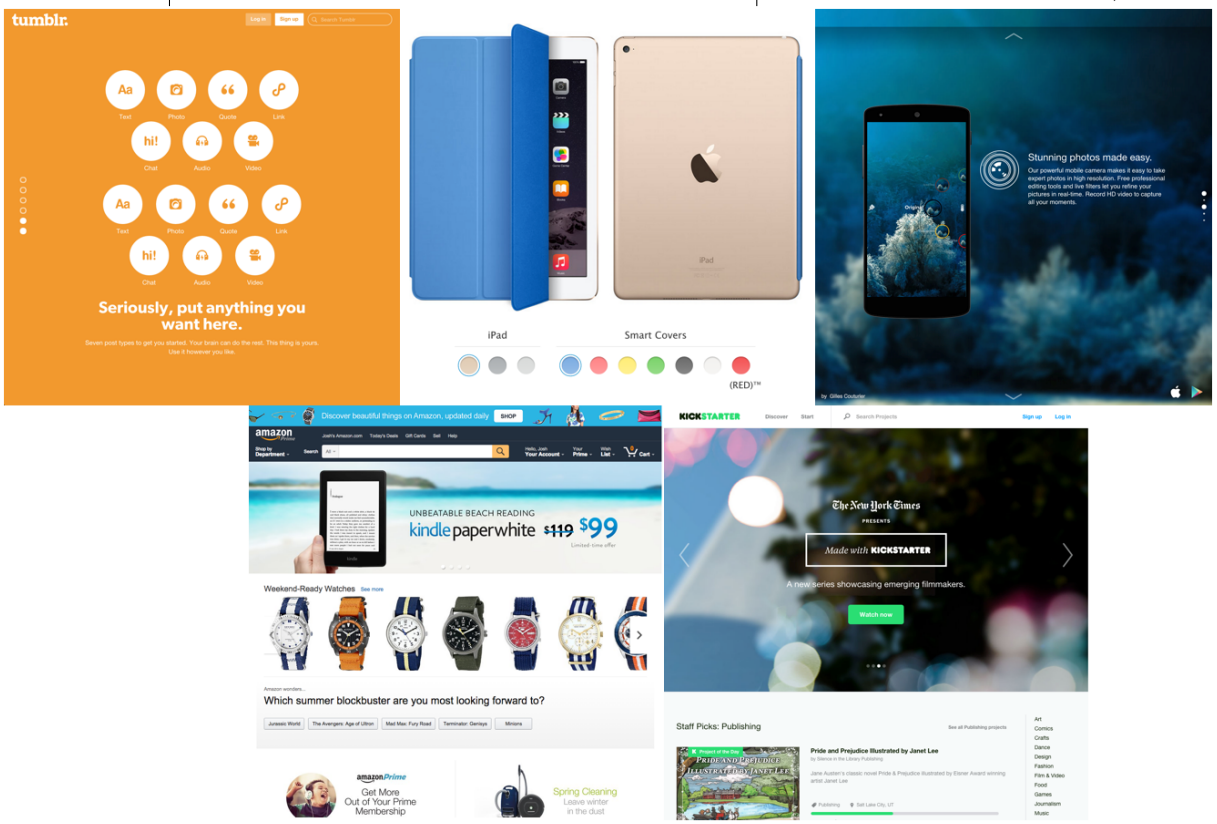

Figure 3.6. Participants reverse engineered 2 UI features from a set of 5. The top table lists each website with its corresponding feature and trigger under inspection. The screen-shots are of Tumblr, iPad, Flickr, Amazon, and Kickstarter (mid left to bottom right).

3.5. Unravel User Study

3.5.1. Method

Our study aims to answer the following research questions:

RQ1 How does a user's strategy for reverse engineering a web application UI feature differ with and without Unravel?

RQ2 How does Unravel affect the amount of time it takes a user to reverse engineer a web application UI feature?

RQ3 Which features in Unravel are the most effective while reverse engineering a web application UI feature?

RQ4 How do junior developers' use of Unravel and reverse engineering strategies differ from senior developers?

The target users of our study are junior web developers with less than one year of professional experience and senior web developers with greater than five years of professional experience. The study is a within-subjects design, where each user was asked to reverse engineer a UI feature in each of two websites from a pool of five, one website with CDT and one with CDT + Unravel (see Figure 3.6). CDT as the control requires no training or installation, and our initial study showed that both junior and senior developers could discover key sources using just CDT. 13 web developers, 6 junior and 7 senior, participated in study sessions lasting 45 minutes. Time was limited to 15 minutes each for each reverse engineering task with a 15-minute follow-up discussion. Each participant was compensated $20. The assignment of websites to participants was randomized, and the order of using Unravel first was reversed for half of the participants.

We chose UI features from five popular professional websites: Tumblr, Apple, Flickr, Amazon, and Kickstarter. While widely used, each contains a clever implementation. When scrolling down on Tumblr's homepage, a card flip effect peels away each page view. Selecting different iPad covers on Apple's product page fades through user choices without changing the iPad image. Flickr's mobile demo synchronizes changes on its virtual phone screen with background fades. Amazon animates its product carousel with easing transitions based on user selection. Kickstarter flips through its banner carousel with fades during pre-programmed intervals. Though not obvious, functionality in these features consists of changing CSS classes, modifying HTML positioning attributes, and loading media in subtle ways.

We taught users about the tool, verified their background, and recorded their tests to ensure result accuracy. Before starting the test, participants were asked to watch a two-minute demo to help them become familiar with how to use Unravel. While participants were recruited by the experience on their CV, they were asked to confirm their amount of professional engineering experience before starting the experiment. Each participant provided a screen recording with audio and click history for the entire experiment.

We tracked three key milestones for reverse engineering. The milestones correspond to events happening at certain times, but participants were encouraged to proceed at their own pace throughout the tests.

M1. Time to finding the first key source.

M2. Time to finding the second key source.

M3. Time to fully understanding how to recreate a feature.

These milestones were tracked via each participant's screen recording to assess understanding. A key source is defined as a high-level code snippet that provides critical-path functionality for a behavior such as a click handler that adjusts the opacity of a div. Some participants had enough experience to describe a solution without reverse engineering, but they were required to find sources to support their claims. Prior to performing the study, the test set of five UI features were fully reverse engineered to identify significant methods, line numbers, classes and variable names in JavaScript, CSS, and HTML. For each solution, two key sources were identified that users must find for each UI feature in order to fully defend how the behavior is functioning. Timestamps for M1 and M2 were logged if a user displayed a key source in view for three or more seconds. M3 was logged when a participant gave notice of complete understanding.

Study pre-tests revealed inconsistency between web applications caused by source minification and obfuscation. Some users knew of the Chrome Dev Tools "Pretty Print" feature that reformats JavaScript source to be readable, while others were confused by large undecipherable blobs of JavaScript. To remedy source minification, we cloned versions of the popular web applications, manually unminified their sources, and hosted them on a private mirror. Subsequent tests showed that mirroring unminified versions resolved the testing inconsistency.

Participants were given a short follow-up discussion to assess how using Unravel altered their strategy and understanding of web application engineering. Questions about specific features of Unravel were included to assess their qualitative value and provide opportunities for feedback on feature usability. Survey results were compiled into four categories: useful features, improvements, learning, and strategies.

Data recordings from each participant were analyzed for statistics on 25 distinct user activities in CDT and Unravel and the time signatures of the major milestones. User activities include actions with similar complexity to switching an inspector pane, inspecting an event handler, or setting a breakpoint. Paired t-tests for with-Unravel vs without-Unravel were performed across all the coded data in the screen recordings to check for significant differences. Distributions were analyzed on an aggregate to determine average milestone times and activity counts.

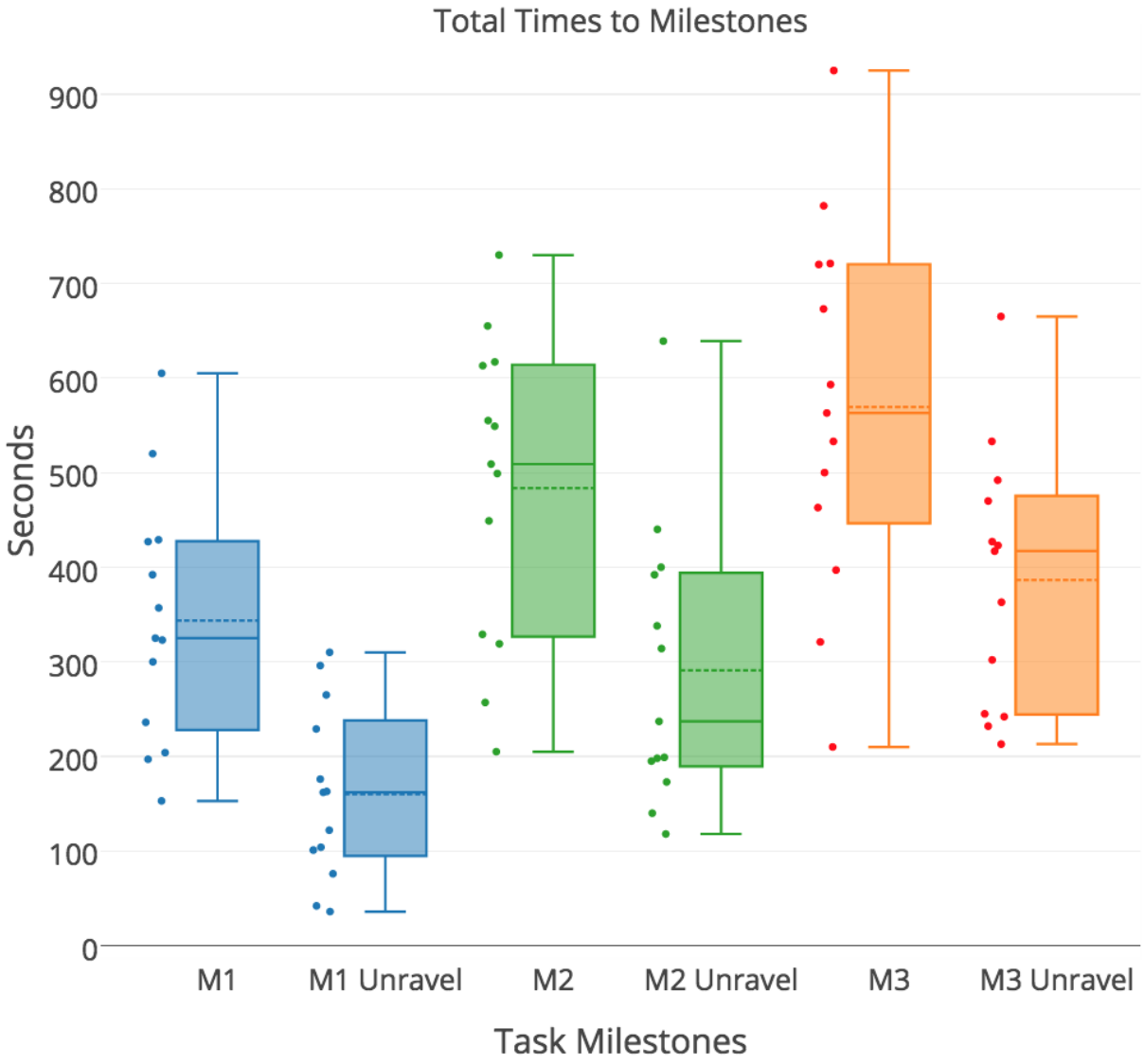

Figure 3.7. Results of the user study are compared in total times to milestones. Boxes indicate interquartile range. Means are shown as dotted lines and medians are solid lines. The box whiskers indicate range including outliers. There is a significant difference in each of total times for M1, M2, M3.

3.6. Study Results

3.6.1. How did Unravel affect task completion times?

Unravel significantly decreased time to all three milestones (see Figure 3.7). Developers achieved milestone I, finding their first key source responsible for the UI interaction roughly twice as fast with Unravel (53.4% time decrease, $t(13) = 4.2, p = 0.0012, \mu_1 = 184s, \mu_2 = 344s$) where μ_1 is CDT + Unravel. Developers achieved milestone II, finding their second key source 39.8% faster with Unravel ($t(13) = 4.533, p = 0.0007, \mu_1 = 291s, \mu_2 = 484s$). Developers achieved milestone III, reaching full understanding 32.1% faster with Unravel ($t(13) = 3.81, p = 0.0025, \mu_1 = 386s, \mu_2 = 569s$).

No significant difference was found in the split times between M1, M2, and M3 (see Figure 3.8). Developers had no significant difference between M1 and M2 ($t(13) = -0.24, p = 0.81, \mu_1 = 131, \mu_2 = 140s$) where μ_1 is CDT + Unravel. Developers had no significant difference between M2 and M3 ($t(13) = 0.33, p = 0.75, \mu_1 = 95s, \mu_2 = 86s$).

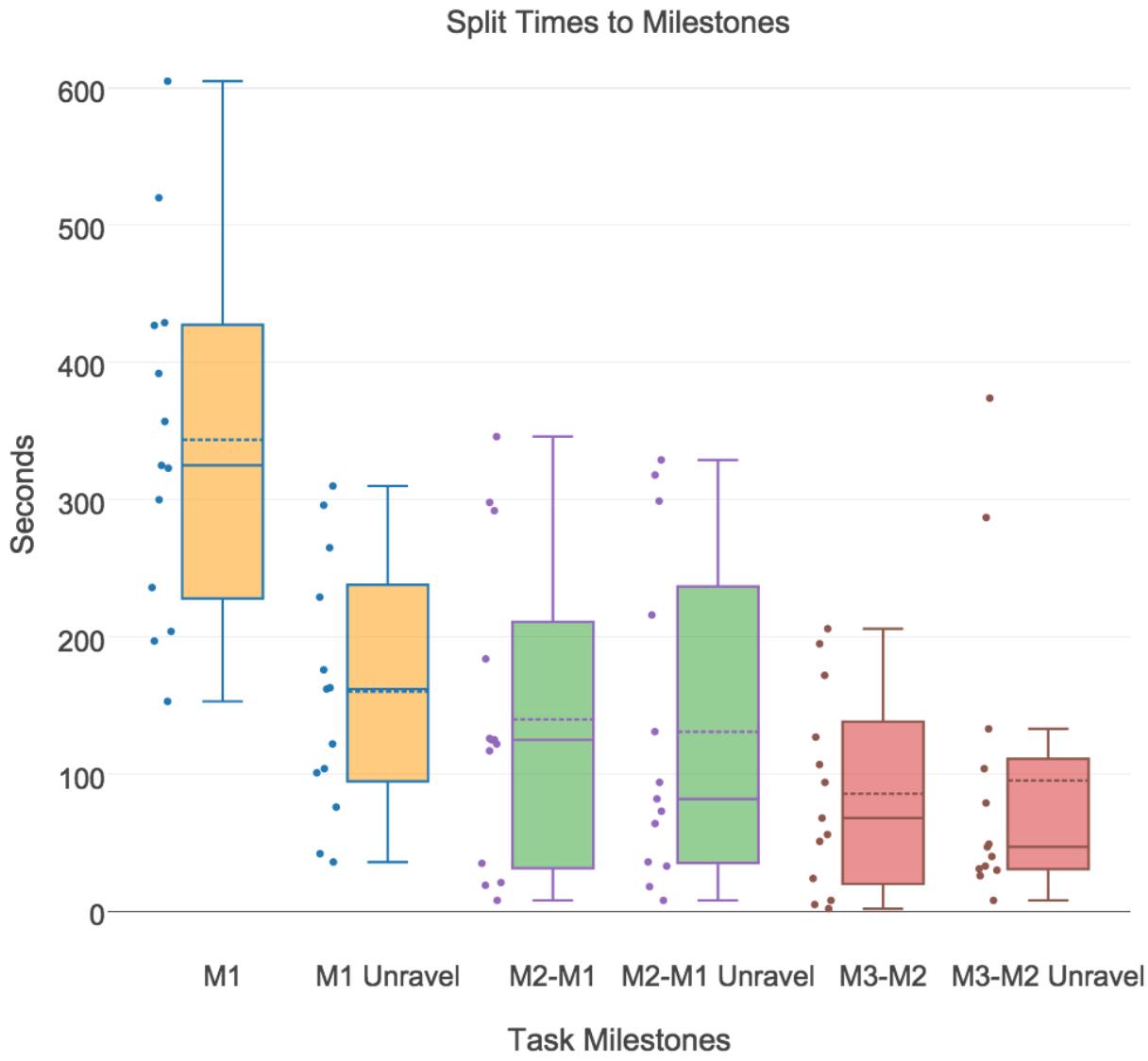

Figure 3.8. Results of the users study are compared in split times between milestones. There is a statistically significant difference between M1 and M1 with Unravel. However, there was no significant difference for the M2 or M3 split times. This means that Unravel was most effective for decreasing the time to first key source.

Differences in milestone times with and without Unravel are explained by variations in user interactions. Developers noted difficulty in finding a starting point during tests without Unravel, which increased their time to M1. Without significant differences in M2-M1 and M3-M2, some participants may have altered their strategy to depend on existing developer tools to find related sources. Total times to M2 and M3 show that no major inefficiencies affected overall time savings by using Unravel.

3.6.2. How did Unravel affect reverse engineering strategy?

Unravel significantly altered the reverse engineering strategy developers used when completing their tasks. Developers browsed an average of 2 JavaScript source files with Unravel compared to 10 without Unravel ($t(13) = 2.84, p = 0.015$). Developers searched for text in sources an average of 1 time with Unravel compared to an average of 9 times without Unravel ($t(13) = 5.6, p = 0.0001$). Developers focused on an element for inspection an average of 1 time with Unravel compared to 10 times without Unravel ($t(13) = 4.67, p = 0.0005$). Developers recreated the UI interactions an average of 5 times with Unravel compared to 11 times without Unravel ($t(13) = 3.45, p = 0.0048$).

3.6.3. How do junior developers compare to senior developers?

Six junior developers reached M1 without Unravel faster than seven senior developers and had no significant difference in other areas ($t(13) = 2.24, p = 0.05, \mu_1 - \mu_2 = 141s$), where μ_1 is for senior developers. Differences with less statistical significance include: senior developers set more breakpoints ($t(13) = 1.99, p = 0.09, \mu_1 - \mu_2 = 4$), senior developers were more likely to inspect network ($t(13) = 2.29, p = 0.06, \mu_1 = 1, \mu_2 = 0$),

senior developers inspected more elements ($t(13) = 1.8, p = 0.11, \mu_1 - \mu_2 = 6$), and senior developers inspected more event handlers ($t(13) = 1.84, p = 0.11, \mu_1 - \mu_2 = 3$). While senior developers used different CDT interface controls to reverse engineer, they desired a broader understanding of the UI feature in the context of the website. A senior developer stated, "I had an idea of how the feature worked before I started, so I wanted to see how the feature was situated in the application first."

3.6.4. Which features of Unravel were the most effective?

In the follow-up discussion, all 13 participants were interviewed for their opinion on Unravel's features, Unravel's weaknesses, reverse engineering strategies, and concepts learned. 10 out of 13 developers stated the JavaScript method traces were most helpful for them to understand a solution. The remaining 3 out of 13 developers stated the HTML changes pane was most helpful. 5 out of 13 developers found the library detection pane useful, while the other 8 stated it did not provide any help. 4 out of 13 participants noted that Unravel could be improved by integrating library detection into the JavaScript stack-traces to highlight the difference between library vs non-library source. 7 out of 13 participants stated a new programming concept they learned while reverse engineering with Unravel, such as using class-toggling to animate objects off screen or using scroll-to thresholds to activate events on a page. 5 out of 13 participants found constraining the scope of observation useful.

3.7. Limitations

Unravel only provides recordings of client-side traces and execution. Server-side source code typically isn't made available for external inspections, but there is an effort to study how to expose API endpoint and behaviors from front-end source [**59**]. Further, professional websites typically use source-code minification techniques to decrease the size of their files an average of 20% [**86**]. For our user tests, sources were manually unminified for participants. This feature can be added to Unravel with the use of JavaScript libraries like js-beautify [**54**], where sources would be parsed and reloaded in unminified form.

Our study did not attempt to identify UI features for which Unravel is not able to provide meaningful information. Unravel only observes JavaScript and HTML pertaining to changes in the DOM. Other JavaScript activity such as data-management, storage, and retrieval would not be visible in Unravel unless a DOM-query was involved in the same call stack (e.g. Unravel would surface functions appending data to the page from an AJAX response, but not the prior AJAX request). In our preliminary study, we discovered shortcomings from SVG transitions where elements had hundreds of positioning attribute changes each second. This flood of changes carries a risk of burying relevant sources. Pseudo-elements and CSS pseudo-classes are outside Unravel's scope of observation but can be easily discovered with existing inspection tools.

In-memory state storage techniques are outside the observation scope of Unravel. Unravel's API harness will not be able to monitor communication with privately closured references to an API. If a web application is designed to preload DOM API queries into memory on page load, Unravel will not capture the query in its API harness if it was not

actively recording at page load. A potential workaround is to detect these behaviors and inject the API harness and observation agents prior to page load.

3.8. Conclusion

Having demonstrated the effectiveness of Unravel for helping web developers reverse engineer professional websites quickly, we revisit techniques that contribute to Unravel's effectiveness.

3.8.1. Organizing and Presenting Large Volumes of Traces

Compared to the performance and interfaces of other source-tracking systems, Unravel is distinguished by its abilities to reduce, scope, and filter large amounts of source detection information in a way that highlights relevant data for the user. A participant stated, "Unravel was way easier to locate specifically where and when in the files the code was executed." We observed through the study that participants found relevant sources by looking at the top items in the HTML JS tracking panels in Unravel. A different participant stated, "Without a doubt I prefer Unravel over sifting through element changes in the Chrome Inspector."

3.8.2. Tracing UI Features to Relevant Sources

Advancing related work [**67, 59, 17, 12, 14, 36, 88**], Unravel introduces a reusable architecture that is both portable and scalable. Unravel serves as a recorder and reducer of meaningful information, with detailed inspection delegated externally. The implementation for this chapter was in CDT, but a participant asked, "Could we have this for Node.js?" While there isn't a DOM to observe, Unravel's JavaScript source tracing

and library detection would work in Node.js. For example, an API harness placed on the HTTP API could capture meaningful traces supporting a GET or POST request. The API harness and application agent allow Unravel's architecture to be reused in any JavaScript environment. The scalable nature of Unravel's architecture allows it to accommodate long recordings of complex features. A participant stated, "I don't even need to inspect, I just hit record and it goes. That by itself is great."

3.8.3. Toward Fine-Tuned Discovery in Web Applications

This chapter provides a contribution towards helping users quickly identify relevant HTML and JavaScript supporting a web feature, however the user discovery interaction involves back-and-forth navigation and only provides one-way inspection pointers into front end code. Unravel quickly helped users identify responsible source code, but its study reveals that Unravel had no noticeable effect in decreasing the amount of time to reaching understanding of the source code (M2 and M3). Returning to the goal of creating Readily Available Learning Experiences from production websites, the next chapter introduces the Telescope system to overcome Unravel's limitations and expand upon its contribution. Telescope provides users with composite views of low-barrier learning materials with support for two-way source discovery between the website feature and its relevant source code.

CHAPTER 4

Telescope: Fine-Tuned Discovery of Web Feature Source Code

This chapter presents the second application towards RALE, Telescope, which is an interactive platform for discovering how JavaScript and HTML work *together* to support a web feature interaction. This chapter contributes a technique to overcome limitations in Unravel's back-and-forth inspection style through dynamic links and interactive time and detail filtering. This chapter has adapted, updated, and rewritten content from a paper at User Interfaces Systems and Technology 2016 [**40**]. The source code for Telescope is openly available [1]. All uses of "we", "our", and "us" in this chapter refer to coauthors of the aforementioned paper.

Professional websites contain rich interactive features that developers can learn from, yet understanding their implementation remains a challenge due to the nature of unfamiliar code. Existing tools provide affordances to analyze source code, but feature-rich websites reveal tens of thousands of lines of code and can easily overwhelm the user. We thus present *Telescope*, a platform for discovering how JavaScript and HTML support a website interaction. Telescope helps users understand unfamiliar website code through a composite view they control by adjusting JavaScript detail, scoping the runtime timeline, and triggering relational links between JS, HTML, and website components. To support these affordances on the open web, Telescope instruments the JavaScript in a website without request intercepts using a novel *sleight-of-hand* technique, then watches

[1] Telescope Github `https://github.com/NUDelta/Telescope`

for traces emitted from the website. In a case study across seven popular websites, Telescope helped identify less than 150 lines of front-end code out of tens of thousands that accurately described the desired interaction in six of the sites. In an exploratory user study, we observed users identifying difficult programming concepts by developing strategies to analyze relatively small amounts of unfamiliar website source code with Telescope.

4.1. Motivation and Contributions

4.1.1. Deriving Authentic Learning Material

Telescope aims to support authentic learning [**81**] by generating low-barrier learning materials to understand code from professional websites of personal interest. Professional websites offer rich details missing from training examples, content that relates to the real world, and opportunities to think in the models of the discipline. However, despite the abundant availability of front-end code, website source code is difficult to read and can contain superfluous details that distract from learning core concepts.

Deriving learning material from websites presents design and technical challenges due to the magnitude and complexity of the underlying source code. A simple UI interaction may require only ten lines of JavaScript, but modern web applications can have tens of thousands of lines of code [**6, 77, 86**]. Bindings between HTML and JavaScript support an interaction, but it is difficult to determine how such bindings are constructed. A simple calendar widget, for example, could be created entirely in JavaScript and appended to the DOM with listeners, or it could be built in HTML and CSS with inline calls to JavaScript hooks. Embedding the widget amidst all its library or utility code in a minification build process blurs the location and scope of code most relevant to enabling the widget's

functionality. With prior tools [**53, 67, 39, 15, 2, 31, 5**], it is difficult to (1) capture the entire scope of JavaScript used, (2) identify the interplay between JavaScript and HTML, and (3) trim away inactive code and library code that get in the way of learning.

4.1.2. Introducing Telescope

We thus introduce *Telescope*, a platform that supports the discovery of website feature implementation by allowing the user to fine-tune a composite view of responsible JavaScript and explore visual links between JavaScript, HTML, and rendered UI components (see Figure 4.1). Telescope helps users generate low-barrier learning materials — less than two hundred lines of code — from tens of thousands of lines of complex website code. For example, a curious user could discover how an interactive map component achieves its dragging effect in JavaScript and HTML by setting Telescope's JavaScript detail level to minimum (dom-modifiers only) and time constraints before and after the click-and-drag. By clicking call and query markers in the interface, visual lines connect JavaScript methods to queried DOM elements, and corresponding DOM components are highlighted in the website. Telescope introduces three design principles to support the creation of learning materials from websites:

(1) *Single Composite View*: As a user interacts with a website, Telescope brings together relevant JavaScript for an interaction into a single, composite JavaScript view to resolve the challenges in finding all code relevant to a behavior in unfamiliar code [**35**]. Users can easily hide sources they deem irrelevant or alter the display order of script sources relative to their dependency load order.

(2) *Detail and Time Controls*: The user can scope relevant Javascript by call time and control the amount of detail they wish to see, ranging from showing DOM-modifying code excluding libraries to only showing all JavaScript present in the website. These controls address a critical need discovered through our human-centered design process, when we found users struggling to understand the code for an interaction when there is either too little or too much JavaScript to analyze.

(3) *Visual Links*: Visual links connect active JavaScript to lines of HTML and website DOM components to expose end-to-end functionality.

The technical contributions of Telescope support its design principles and enable using Telescope to examine website UI interactions across the open web in real time. Specifically, we introduce (a) the *Wisat architecture*, which supports source code tracing and instrumentation on public websites, and (b) the *Sleight-of-Hand method* (SoH), which swaps a website's client-side implementation during runtime with its instrumented counterpart. The SoH method transitions websites from a non-traceable state to a fully instrumented state, supporting live interaction traces as a user interacts with their website. The Wisat architecture then transmits runtime traces used to decide which JavaScript is displayed in Telescope's composite view and provides the linking data necessary for drawing connections between JavaScript, HTML, and website components. In the rest of this chapter, we introduce Telescope and its main components for tuning UI discovery and linking JavaScript and HTML source code. We examine Telescope's performance and study its effectiveness through a case study using Telescope on seven professional websites and an exploratory study with five users. We conclude with a discussion of design principles, limitations of our approach, and a brief look at the next chapter on Isopleth.

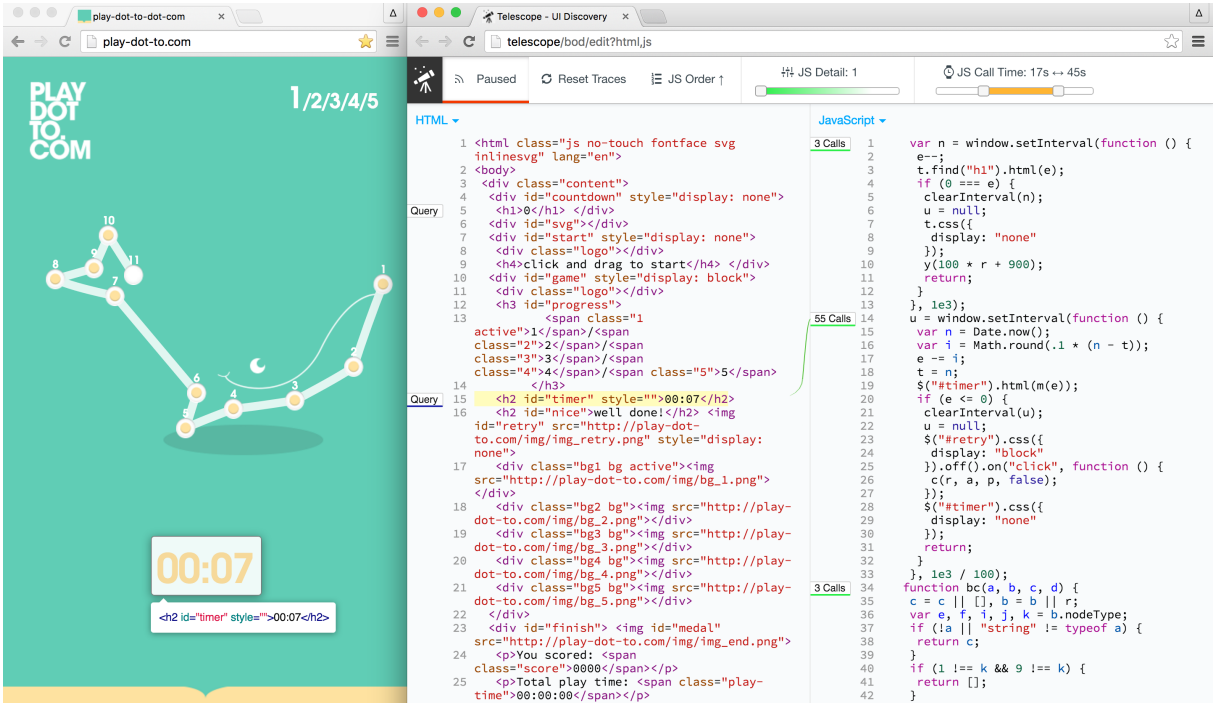

Figure 4.1. The Telescope interface is being used to discover how this HTML5 connect-the-dot game's timer works. The interface is paused to freeze the current view. The detail level is set at minimum, and the JavaScript call time is constrained between the 17th and 45th second of execution. The left Telescope panel (middle) shows a filtered HTML view, where an active element is highlighted and query markers denote that JavaScript queried those lines during the chosen time window. The right Telescope panel shows the website's JavaScript, filtered by time and detail. With the current settings, only the most relevant JavaScript is displayed: active non-library JavaScript which queried the DOM in the constrained time frame. A curved line is drawn to connect the JavaScript line to its DOM query.

4.2. Telescope

Telescope is a web-based platform for producing learning material to implement a UI interaction. By using the Wisat architecture discussed later, Telescope receives JavaScript runtime traces and DOM state changes from a website's UI during use. The user views all JavaScript for a website in a single composite view, condensed by time constraints, filtered by detail level, and ordered by last-loaded script file first. User-activated visual links connect JavaScript, HTML, and rendered components in the browser (See Figure 4.1).

4.2.1. Receiving JavaScript, HTML, and Trace Activity

A user launches Telescope by initiating website instrumentation from a browser extension. Once connected, Telescope begins receiving traces and its interface updates in real time to reflect the latest DOM state and an accumulation of JavaScript traces. Queried DOM elements are marked with a *Query* gutter marker. Active JavaScript functions are marked with a *Call Count* gutter marker, a technique we adopted from Theseus **[53]**. Telescope continuously analyzes call graphs to determine which JavaScript calls were involved in querying the DOM. If an active function is identified as being involved in a DOM query, it is marked with a green call marker instead of a colorless marker to highlight its significance. Depending on the detail setting, a user will see a certain subset of JavaScript in view with the corresponding call counts for that subset. As a user continues their interactions with the observed website, their function invocation counts will increase.

4.2.2. Tuning Telescope: Order, Detail, and Time

A core design goal in Telescope is to avoid overwhelming the user with large amounts of trace information by presenting the most relevant JavaScript together in a single composite view. Most of the websites tested in our case study, such as The New York Times "Snow Fall" article, have tens of thousands of lines of unminified JavaScript and hundreds of lines of HTML. Even a simple photo-slideshow change effect could have thousands of function invocations if embedded in MVC logic from a large JavaScript framework like Angular or React.

The controls in the header of the Telescope interface allow the user to fine-tune the source code activity during a UI interaction (See Figure 4.1). From left to right, the user has the ability to (1) pause/resume activity, and to reset the interface to a cleared activity state; (2) flip the JavaScript presentation order; (3) adjust the detail of JavaScript sources displayed; and (4) constrain the time of active JavaScript sources. We discuss each of these affordances below.

4.2.2.1. 1. Pause/Resume and Reset Traces. The Telescope UI updates continuously as the website's UI state changes to show live updates to source code execution. Users can see active JavaScript populate in view, as well as increasing call/query counts next to JavaScript/HTML lines. To freeze the capture state and ignore ongoing functionality, a Telescope user can pause the interface at its current DOM state and JavaScript trace collection. Users can browse and interact with other UI controls during this frozen state, but no new data will be displayed. Upon resuming, Telescope updates to the latest state of the website. Resetting Telescope empties its collection of JavaScript traces and synchronizes its HTML view with the latest DOM state.

4.2.2.2. 2. JavaScript Order. Early pilot studies revealed that relevant source is often found in scripts at the end of a website's load order. The interpreted nature of JavaScript combined with the disorganized nature of website script-loading leads web developers to load scripts with more dependencies last and fewer dependencies first [**4**]. As a consequence of this dependency pattern, our earlier prototypes often placed the most important high-level JavaScript hidden at the bottom, leaving relevant code out of view. Based on this observation, Telescope by default inverts the load order to display last-loaded scripts first as a heuristic. The composite JavaScript panel in Telescope displays scripts sorted as a whole, so the inner contents of scripts will remain in their original form. The JavaScript order control allows a user to invert the presentation order, e.g. to support cases where our heuristic may not apply.

4.2.2.3. 3. JavaScript Detail. Early pilot studies also revealed that simply showing users all active JavaScript code provided little value. To support discovery, our test users requested variable control over the detail visible. With Telescope's JS Detail slider, a user can control the amount of JavaScript visible. By default, Telescope slides detail to the left extreme (L1), which shows how higher-level JS achieves an effect using library APIs without showing library code. Low-level DOM API calls are often wrapped by libraries and would be hidden at this level, e.g. a jQuery call $(div) is displayed instead of the DOM API call. Sliding detail to the other extreme will reveal all of the JavaScript for a website. The detail levels include:

L1 (default): DOM API callers and parent callers, excluding library code. For example, call stacks to `document.getElementById` would be surfaced as well as calls to library API's wrapping `document.getElementById`. In this detail level, a jQuery library call like

`$(“#fooId”)` would appear and be marked with a green hit marker, but the internal calls within jQuery would not appear.

L2: Active JavaScript, excluding library code. This includes any JavaScript that ran in addition to the above level without showing library code. For example, functions updating objects in memory or storing cookies would appear here.

L3: Active JavaScript. In this level, all of the JavaScript that run during a user's interaction with the page will appear here. In other words, only dormant code is hidden.

L4: All JavaScript excluding library code. This level will show all JavaScript in a website (both active and inactive) while hiding library code. In other words, this level shows users what code was written specific to the website while hiding third party library code.

L5: All JavaScript. This level will surface all of the JavaScript for the page to show users the most comprehensive view of libraries and professional code working together.

4.2.2.4. 4. JavaScript Call Time. Telescope users can use timeline constraints to set a start time and end time to see which functions were executed during the specified interval. While JavaScript can execute asynchronously at arbitrary times, users can still slide the time constraints as a way to omit code outside a time interval, such as initial setup code or continuous interval functions. Dragging markers on the timeline reshapes the scope of time in percentage of width dragged. Constraining the timeline in this way helps users who wish to inspect multiple interactions they made with the website by constraining the inspection view to different points in time.

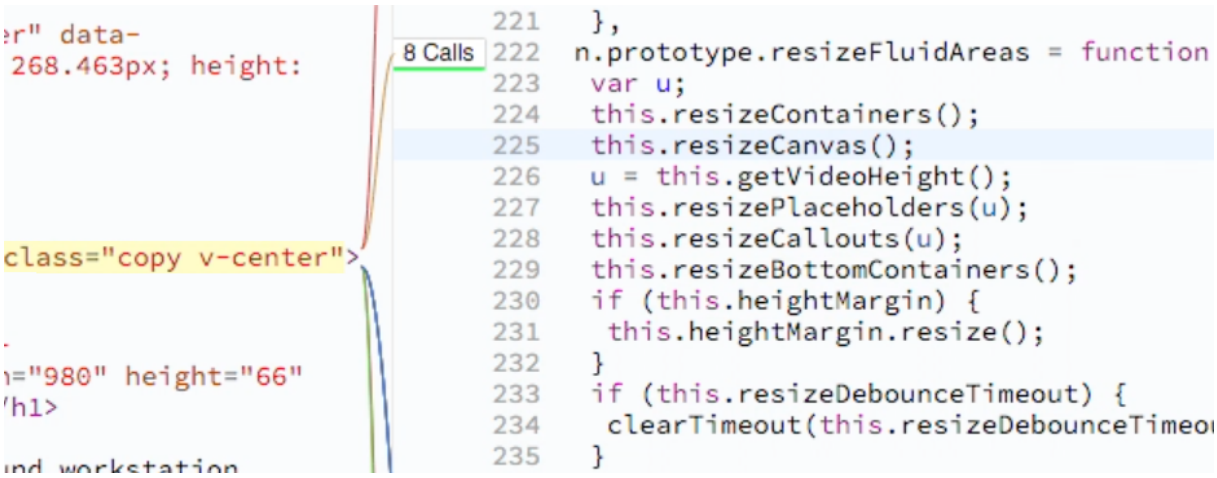

Figure 4.2. Clicking a Telescope HTML query marker from the Mac Pro website (left) shows lines to four JavaScript functions. In this view, a line leads to function `resizeFluidAreas`, which resizes elements on scroll.

4.2.3. Linking HTML, JS, and the Rendered DOM

Telescope provides bidirectional visual links between the HTML, JavaScript, and website DOM to provide end-to-end connections from source code to its UI output. Inspired by *Gliimpse* — which creates in-place visual transitions from code to UI and vice versa **[24]** — these links help form conceptual models of how JavaScript and HTML work together. But unlike Gliimpse, Telescope shows both the source code state and rendered UI simultaneously. Users can visualize how high level functions change many elements (see Case Study: Mac Pro) or how a single element event can trigger many function handlers (see Case Study: Dot-to-Dot). During Telescope sessions, *Query* markers appear in the HTML pane, and *Call* markers appear in the JavaScript pane. Clicking an HTML query marker draws lines to JavaScript functions which query the HTML line (see Figure 4.2). Clicking a green call marker (signifies DOM-query) draws lines to HTML nodes which were queried by the JavaScript line.

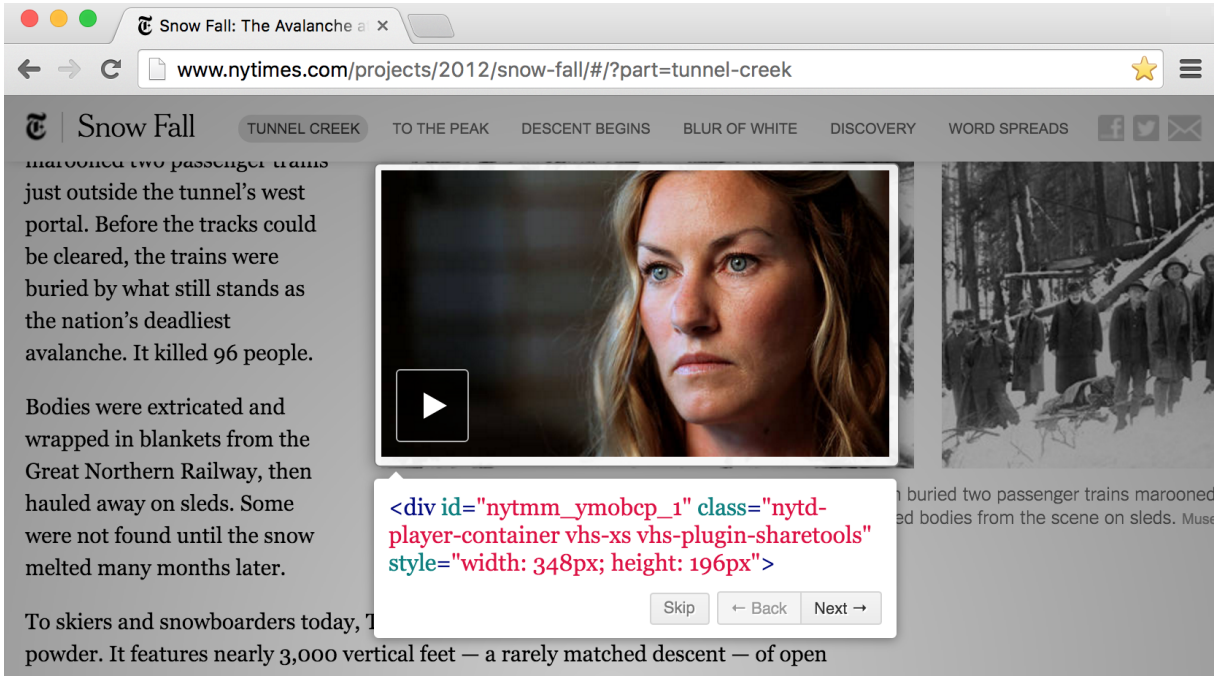

Figure 4.3. Clicking Telescope's code markers for the New York Times "Snow Fall" website highlights related DOM elements in the website. The DOM element's source is included in the highlight, connecting context to Telescope's HTML view.

Exploring HTML-JS links in either direction invokes a response in the website, where the rendered DOM nodes are highlighted in the foreground (See Figure 4.3). If multiple DOM nodes are involved in a query, a walkthrough is constructed in the rendered website that highlights each involved element in sequence. Conversely, if elements were deleted, the user is notified that the element is no longer in the DOM.

In early pilot studies, traces from library, tracking, and ad-content scripts caused confusion in understanding UI feature implementation. Telescope now hides many libraries and irrelevant scripts and provides affordances for users to hide other scripts they deem

irrelevant. By default, library scripts such as jQuery and Angular are hidden, as are popular advertisement and usage-tracking scripts such as DoubleClick and ScoreCardResearch. Users can then hide or show scripts selectively in two ways. Through the JavaScript dropdown view at the top of the JavaScript panel, users can see a complete list of the sources in view and selectively toggle their visibility. Alternatively, users can slide the detail slider to the far right to bring all sources into view (more library and ads) or far left to show only sources relevant to DOM manipulation (less library and ads, see Figure 4.1 top right).

4.2.4. Design Process and Design Insights

In the process of designing Telescope, we iterated through three software prototypes. Prototype 1 provided the ability to record an interaction and extract a subset of HTML, CSS, and JS into a web sandbox with visual output that can be shared with other users to explore on the web (e.g. for use in a group). Prototype 2 dropped sandboxed output and added affordances to selectively hide inactive code and sources. Clicking JS gutter markers exposed a function's callstack. Prototype 3 gained the Wisat architecture for continuous distributed tracing. After prototype 3, we trimmed features users didn't value and added controls for order, time, detail, and interactive links.

With each prototype we conducted a small pilot study to better understand how to help users overcome learning barriers tied to unfamiliar code. Each study recruited a convenience sample of three junior developers who used the prototype for 30 minutes each and were paid $20. In this process we discovered four primary design insights:

- **Users need variable amounts of JavaScript to understand different programming concepts.** Each prototype provided affordances to selectively trim

down the JavaScript, but users were unsure what to trim and found it difficult to remember what they had trimmed from view. Users expressed desires to see both high-level code and low-level utility code at different times to establish a basic understanding of how the program works before looking into its details. We implemented the JS Detail control to adjust the composite JavaScript view to different detail levels such as more minimal for DOM-modifying code or more verbose for deeper discovery involving AJAX and MVC logic.

- **Users have varied processes for playing and inspecting.** Observed in all three studies, some users like to repeat their interaction several times before using Telescope, whereas others will create an interaction and jump to Telescope before it completes. Prototypes 1 and 2 had a static extraction technique that frustrated users who liked to alternate between playing and inspecting. Telescope now continuously updates both its HTML and composite JavaScript as user plays with a website, while also giving the ability to pause and constrain their historical runtime timeline.
- **Users benefit from visual links connecting code to observable output.** Similar to Gross et al's recommendations to *connect code to observable output*, we found that linking the JS and HTML contexts to observable output helped users understand JavaScript's relationship with HTML [**35**]. By the third prototype, our users were still having trouble understanding how the JavaScript and HTML related even though active-code highlights were provided in both panes. We added support to draw visual lines from either direction between HTML and

JavaScript. Upon drawing these lines, the DOM element is highlighted in the website to complete the connection between the code and its output.

- **Metadata and redundant filters overwhelm the user.** Throughout prototype iteration, we kept accumulating features which began distracting users from efficiently using Telescope. To promote simplicity, we cut away features that were distracting or provided little use to achieving the goal of promoting understanding. Features cut included call-stack inspection, the CSS pane, a DOM preview pane, code-hiding toggles, and other extraneous features.

4.3. Implementation

Telescope's implementation goals include deployability across the open web, full-scope JavaScript instrumentation, and multi-user session support. Unravel [**39**] and Scry [**15**] provide JavaScript traces on public websites but limit their inspection scope to DOM-querying JavaScript. Theseus [**53**] provides full instrumentation but requires a debugging proxy for setup on public websites. To support future empirical field research and promote user adoption, we seek implementations that are easy for users to install with minimal setup. Existing architectures from related systems are designed to only support single-user sessions.

In the rest of this section, we describe the *Wisat architecture* and *Sleight-of-Hand methodology* that together enable Telescope to bring source instrumentation and JavaScript trace analysis to public websites with minimal user setup. Building upon related systems, Telescope brings Fondue's source instrumentation to the open web, augments Theseus'

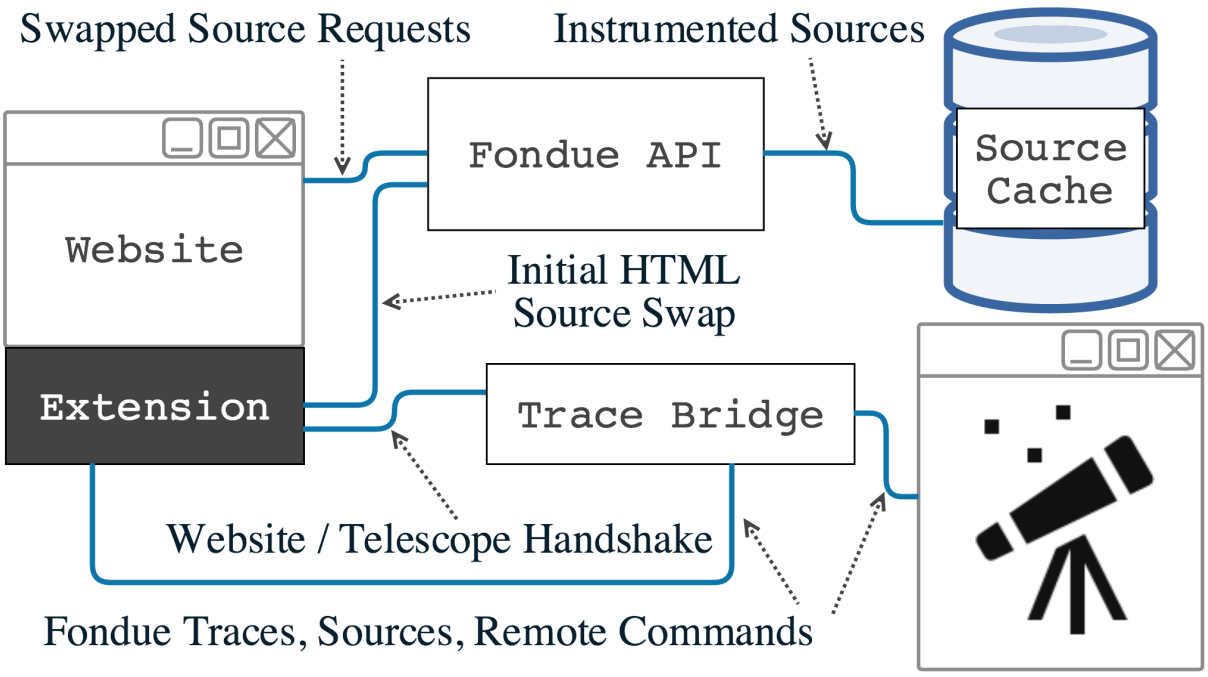

Figure 4.4. The Wisat architecture supports Telescope's ability to remotely process website interaction traces. A website receives its initial source swap via the Chrome extension. The website fetches instrumented scripts from the Fondue API (top), and the Chrome extension negotiates a two-way handshake via the Trace Bridge to connect it with its Telescope session (bottom). Upon successful connection, JavaScript traces and source data propagate continuously over the trace bridge.

active code markers with interactive links, and uses JSBin's collaborative online editor environment as a foundation [**53, 82**]. Telescope consists of a component-based architecture where new technologies can be swapped in or integrated later on.

4.3.1. Wisat Architecture

The Wisat (Web interface swap and trace) architecture supports Telescope's JavaScript instrumentation, trace propagation, source transmission, remote control, and sleight-of-hand source swapping. After a website is instrumented via the browser extension, Fondue

API, and source cache (see Sleight-of-Hand Method), the browser extension negotiates a two-way handshake between the website and Telescope interface via the *trace bridge*. Once connected, traces, sources, and remote commands can flow freely between the two, populating Telescope's code views and enabling remote DOM component highlighting (see Figure 4.4). Designed for web scalability, this architecture separates functional components so that each may be distributed across multiple load-balanced instances. The components of this architecture are defined as:

- Telescope UI: A website for receiving source trace activity from an instrumented website, fine-tuning source findings, and sharing with others.
- Fondue API: REST web service for JavaScript & HTML instrumentation and deobfuscation with caching, served over HTTPS to comply with mixed-content policies.
- Trace Bridge: WebSocket server for live cross-origin-compliant transmission of JavaScript traces, DOM changes, and commands between the website and Telescope interface.
- Chrome Extension: Agent injected into website to deploy the Sleight-of-Hand source swap, broker handshake with Telescope interface, and broadcast source activity.

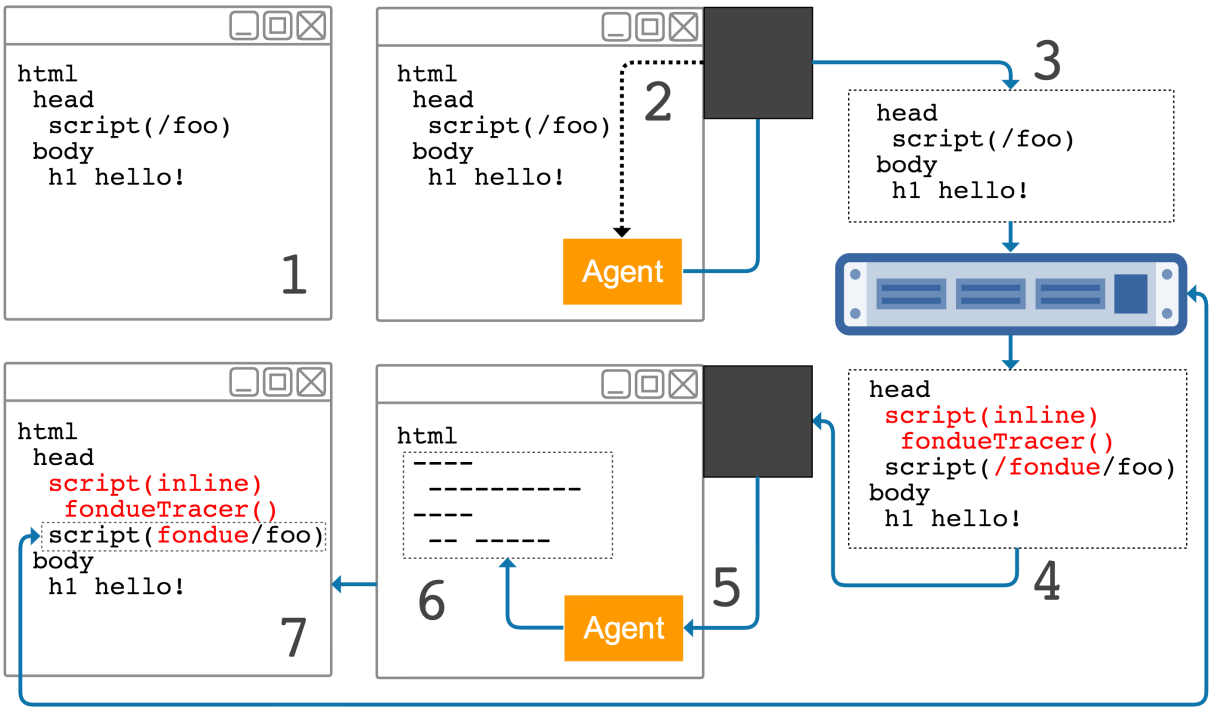

Figure 4.5. The Sleight-of-Hand technique pictured above is a 7-step process for instrumenting a website's source code via browser extension (black squares) and external instrumentation server (blue, middle right). After website load (1), the extension deploys an agent (2). The agent sends the sources for instrumentation via AJAX (3), which are returned (4), passed to the agent (5), and swapped for the originals, deleting references (6). The browser makes requests for the newly instrumented sources (7).

4.3.2. Sleight-of-Hand Method

The Sleight-of-Hand Method (SoH) expands upon techniques from Fondue [**53**] to bring source instrumentation to public websites. Current methods for JavaScript source tracing either trace only DOM-querying JavaScript [**39, 15, 14**] or require a man-in-the-middle debugging proxy [**53**]. Neither of these approaches fits with our goal to fully trace JavaScript execution and make setup simple. The SoH method — deployed from a one-click-install browser extension — implements full JavaScript traceability by swapping the

scripts of a website with their instrumented versions. The SoH process is outlined below (see Figure 4.5):

(1) Load a website and initiate SoH.

(2) Deploy a JavaScript agent into the website from a browser extension with bidirectional communication.

(3) Agent transmits the OuterHTML property of the root DOM element to the instrumentation API via extension, circumventing cross-origin policy.

(4) Instrumentation API returns the HTML with inline scripts instrumented and <script> tags with altered "src" attributes pointing to an instrumentation API URL.

(5) Browser extension passes the response to the agent.

(6) Agent clears the DOM and JavaScript state to ready the page for an artificial load process (without refreshing). It does so by overwriting the existing DOM with an empty root, iterating through non-native window object attributes while deleting them, and calling clearInterval on global interval indices 1 to 999.

(7) Agent inserts instrumented pieces of the DOM in a strict order to control script loading to simulate the script load order of the original site.

SoH leverages vulnerabilities enabled by browser extensions, circumventing source alteration protection by overwriting original sources [**16**]. An SoH is deployed from a browser extension with liberal permissions to modify the page and communicate with third party servers. It does so regardless of logged-in state or HTTPS encryption.

The SoH method works for many websites, but synchronized HTML/JavaScript workflows and Content Security Policies (CSP) can cause problems. For example, if a user

kept scrolling for more news to load in a Facebook news feed, the in-memory JavaScript would reflect news list additions and the HTML would reflect the same. If the SoH was initiated after scrolling for more news, the news list HTML would be correct, but the in-memory JavaScript would not have the news list additions. The result would be an odd-UI experience where interactions hit sync-error handling, such as moving the user back to the top of their news feed. Further, any scripts lazy-loaded after the SoH starts and before SoH ends would cause more UI oddities or potentially break the process. If breakpoints were injected into the page during the SoH process, it would simply pause the process. Implementing a whitelist CSP successfully blocks the SoH method, because it asks the browser to enforce a strict list of source domains [**93**]. However in Chapter 5, a workaround is provided to remove CSP headers from obstructing the SoH process.

4.4. Case Study

To better understand Telescope's capabilities and performance, we used it to identify relevant lines of code, key interaction methods, and implementation patterns across UI features on seven popular websites. This study aims to address the following research question:

RQ1 To what extent can Telescope reduce and scope lines of code for understanding complex feature implementations?

We chose websites with interesting and complex UI features that are not straightforward to understand, that have over ten thousand lines of code. Interactions of interest include a map-drag (XKCD), a scroll animation (Tumblr), a dot-drag (DotToDot), scroll-driven video sizing (NYT), a load-and-scroll-driven float (iPhone), a scroll-driven product

	Initial Stats			Website On-Load Activity				Website Interaction Lines of Code			
	Class	HTML LOC Total	JS LOC Total	HTML Queried	Active JS	Telescope Default JS	LOC Reduced	HTML Queried	Active JS	Telescope Default JS	LOC Reduced
XKCD 1110	L	77	11,023	31	9,378	4,730	57.09%	1	1,079	49	99.56%
DotToDot	M	34	12,910	3	9,697	5,534	57.13%	23	1,687	115	99.11%
iPhone SE	M	788	53,810	95	8,648	2,341	95.65%	20	907	84	99.84%
Tumblr	H	182	92,070	34	10,504	4,970	94.60%	1	1,839	52	99.94%
Mac Pro	H	794	33,631	248	10,095	1,877	94.42%	68	1,650	934	97.22%
Southwest Air	H	4,140	35,011	143	8,342	4,577	86.93%	1	1,825	53	99.85%
NYT Snow Fall	H	1,458	41,526	2	2,546	521	98.75%	30	522	150	99.64%
Average		**1,068**	**39,997**	**79**	**8,459**	**3,507**	**83.51%**	**21**	**1,358**	**205**	**99.31%**

Figure 4.6. Results from our case study show the amounts of code Telescope reduces, using time and detail filters to draw distinction between on-load setup code and interaction code. Each website's complexity class is provided (Small, Medium, High). The JS total lines of code (LOC), calculated after normalized unminification, are listed per each website (left) and categorized by all active JS LOC and the default DOM-modifying JS LOC with library code removed. In blue (middle, right) the LOC in Telescope's default view for on-load and interaction show the amount of reduction Telescope performs for the user while maintaining relevance. HTML LOC queried are listed, showing the small portion of DOM elements involved in each UI interaction. Interactions include a map-drag (XKCD), a scroll animation (Tumblr), a dot-drag (DotToDot), scroll-driven video sizing (NYT), a load-and-scroll-driven float (iPhone), a scroll-driven product show (Mac Pro), and a date-picker render and select (Southwest).

show (Mac Pro), and a date-picker render and select (Southwest). We classified websites as light (L), medium (M), or heavy (H) in proportion to their UI complexity and average number of function invocations. For each example, we tracked the minimum usage necessary to discover UI features on the website, while comparing against Unravel as a control.

4.4.1. Fine-Tuning Lines of Code

Telescope supported discovery on the seven websites with minimal tuning regardless of source code size (see Figure 4.6). We measured the lines of code visible in Telescope

during on-load and interaction, normalizing JavaScript and HTML with unminifying preprocessors. Telescope identified each site's large on-load setup processes (521 to 5,534, mean 3,507 LOC), allowing us to easily scope timeline constraints beyond the setup code to yield each interaction's code (49 to 934, mean 205 LOC). Besides the Mac Pro example, **running Telescope on all other websites with the default detail setting yielded 150 lines or less of code that sufficiently explained how the interaction was created in each site.** With 1 to 68 (mean 21) LOC of HTML queried during interactions, the HTML query markers offer a simple starting point for exploration.

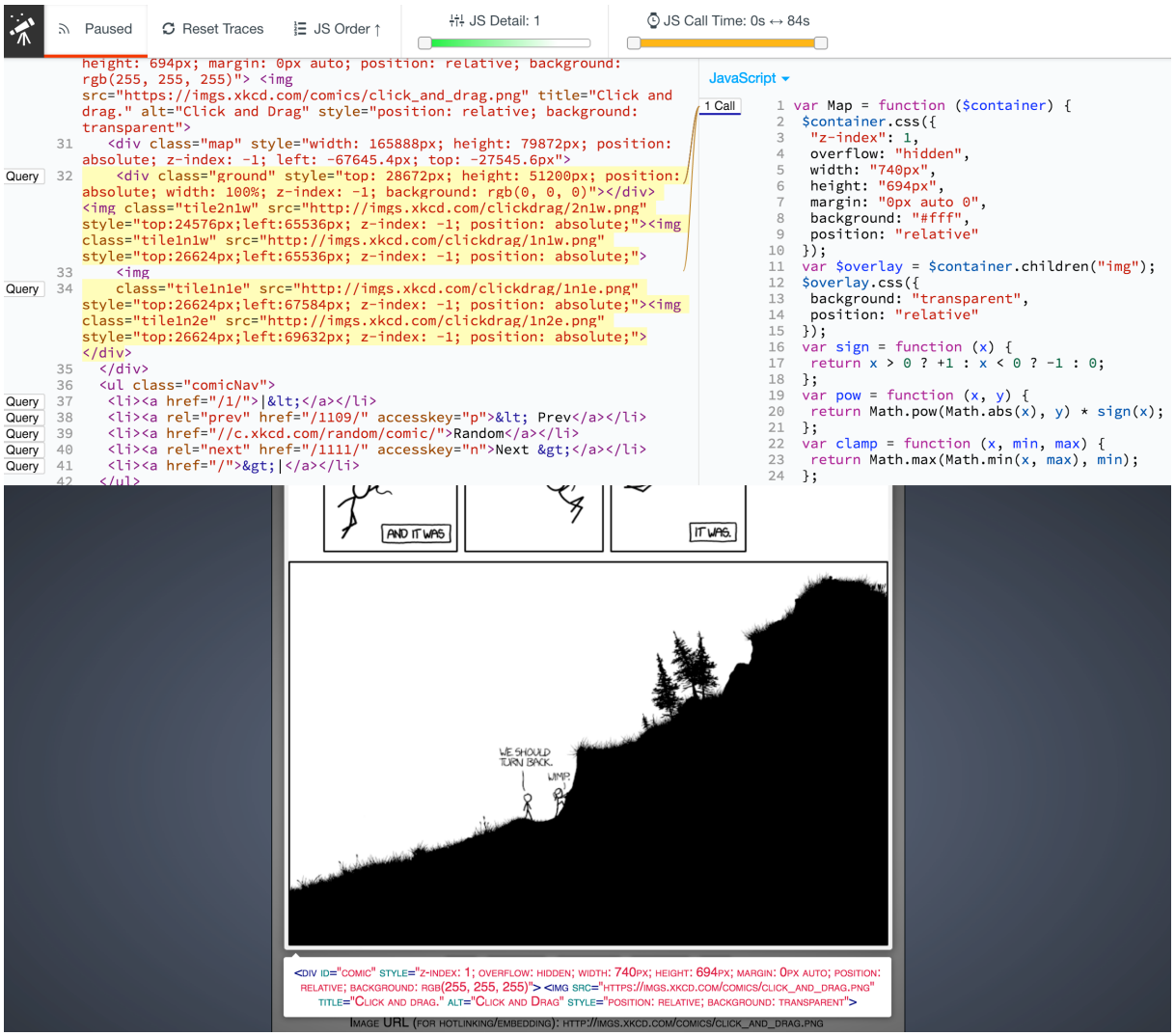

Figure 4.7. Telescope is being used to discover XKCD's map-drag implementation. A JavaScript call marker has been clicked next to the `Map` function, resulting in HTML line highlights and a DOM element highlight in the website.

4.4.2. Low Complexity Example: *XKCD 1110*

XKCD's interactive comic #1110 website presents a simple test scenario for Telescope with its relatively small codebase and direct UI interaction (see Figure 4.7). Telescope revealed a composite 49-line draggable map implementation (excluding library code). We quickly discovered functions `map`, `update`, and `drag` with Telescope's default settings. We examined the startup code and moved the timeline past startup to see the interaction. The map-drag effect is achieved by events bound on `mousedown` that track mouse position relative to a center start position. The map is a grid of image tiles with names representing their position, where images ± 1 away from the centered tile are loaded and set to visible, while others are hidden.

Using Unravel on the same interaction, we were able to easily find the same functions behind XKCD's map load, however we needed to look through 420 lines of JavaScript to find how relevant calls in separate files fit together. Unravel showed changes to the DOM caused by dragging the map, but it was difficult to determine the scope of JavaScript operating on the map. Setting DOM breakpoints through Chrome Developer Tools, we were able to step through function calls responsible for modifying the map.

4.4.3. Moderate Complexity Example: *Dot-to-Dot*

In analyzing the design award-winning Dot-to-Dot game, Telescope helped us to understand how the game connects the dots (see Figure 4.1). We sought to understand the code behind connecting a dot to another: dots appear, a line is drawn, and audio plays a dot sound. We didn't need to look far to find a `dot` class in the setup code, which was referenced later in the JS time 23s to 42s. The JavaScript code was heavily minified, but

Telescope expanded it in a way we could infer how functions operated even without their names. Function `c` activates a game round, function `y` starts the timer interval, function `o` draws a line invoking RaphaelJS, and function `v` handles dot clicks and dot animation. Sliding JS detail towards the middle we found a `pop.mp3` xhr request, where the response is stored in a variable and played via `SFX.pop()`.

In this scenario, Unravel provided hundreds of JavaScript inspection points and DOM changes. We inspected the top two most-called functions and quickly found the game's timer and dot-insertion logic by clicking through Unravel's inspection points. Using Chrome's search feature was more convenient than manually looking through the remaining Unravel results, so we ran find-all queries for RaphaelJS calls and set breakpoints to determine how game rounds began. Separating the game's setup code from runtime code was difficult with Unravel, because all of the JavaScript functions accumulate in one list that is only sortable by call count or function name.

4.4.4. High Website Complexity Example: *"Snow Fall"*

The Pulitzer Prize winning New York Times article "Snow Fall" stretched Telescope's technical ability with 41,526 lines of JavaScript and 1,458 lines of HTML along with a high volume of recurring background JavaScript execution. In this test, we sought to discover how the Steven's Pass flyover interaction was activated. We scanned through 300 lines of irrelevant ad and tracking code before finding the right Telescope settings. We set the JS Call Time to 41s to 73s and set the JS Detail to the middle, where we found relevant functions `videoBG.setFullscreen`, `checkArticleProgress`, and `percentTillNext` related to an HTML5 video player (see Figure 4.3). The latter two run on every scroll

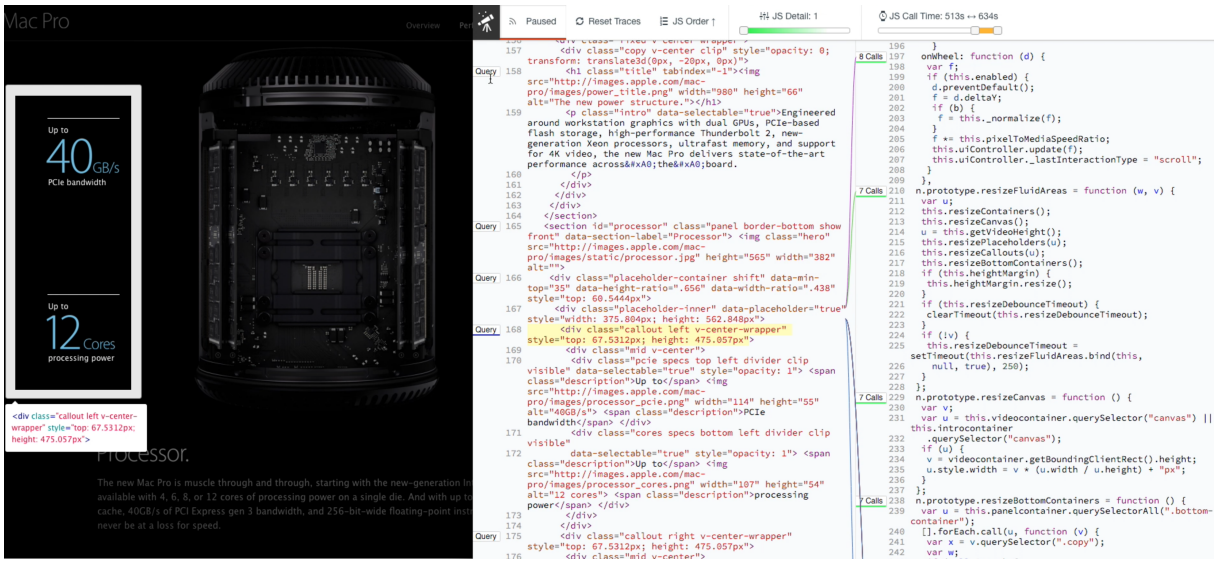

Figure 4.8. We evaluated Telescope's performance and source discovery on Apple's Mac Pro product demo website. While performance lagged during UI animation, Telescope accurately captured and reduced the source code view to show how the scroll-driven effect works. Above, an HTML line marker has been selected in the Telescope interface that draws lines to linked functions and highlights the DOM component.

event and the former is activated when the article progress reaches the "Tunnel Creek" narrative. We found related HTML elements `div.nytmm_video_player`.

In comparison, Unravel quickly revealed results pointing to functions responsible for setting full screen and initiating video playback, but like in the previous case, the magnitude of function traces occluded the search for other meaningful functionality. We were unable to quickly find the remaining functional pieces for checking the article progress and activating new sections.

4.4.5. High Complexity Example #2: *Mac Pro*

The interactive product page, which disassembles an Apple Mac Pro on user scroll, tested Telescope's performance limitations but revealed insight into the website's design (see Figure 4.8). The initial product-rising animation was captured in Telescope, logging 30k+ function invocations. We scrolled down to activate the Mac Pro's disassembly animation and tuned Telescope's JavaScript time to exclude on-load code and any code after our interaction. We disregarded 400 lines of code before finding the appropriate settings. We found an MVC architecture with event-driven-design, where a `sectionController` and a `clipController` listens for events relative to a `timeline` with functions like `pauseTimeline`, `getVideoHeight`, `resizeFluidAreas`, and `resizeCanvas`. While the clever video playback and container resizing became more evident, we found misleading code that queries and resizes canvas elements when there are none.

Similar to the previous two cases, Unravel found hundreds of changes and traces, with the topmost being calls to trigger, enable, and update sections via an `onWheel` handler. Discovering components of the MVC architecture through Unravel was extremely difficult in this case. In sorting by JavaScript invocation count and DOM query count, Unravel highlighted portions of the MVC most active in DOM modification. This resulted in pointers to view logic, but model/controller logic was difficult to surface.

4.4.6. Runtime Performance

Telescope performed without significant delay on four sites but experienced intermittent UI blocks on three dynamic sites with hundreds of UI transformations per second. While Telescope is primarily a retroactive inspector, it continuously receives trace information

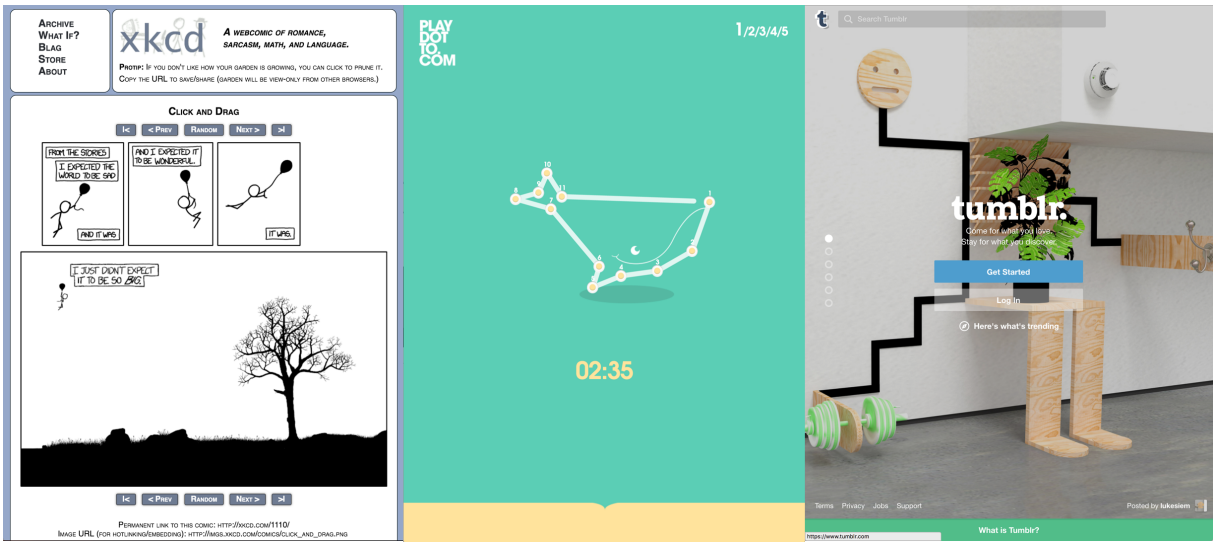

Figure 4.9. We observed Telescope's use while discovering a map-drag interaction on XKCD (left), a dot-connect interaction on Play-Dot-To.com (middle), and a scroll animation on Tumblr (right).

from websites. With each new trace, Telescope recalculates hit counts and surfaces DOM-querying calls out of library call stacks. This computation happens in Telescope JavaScript UI, thus slowing down its UI updates. With a relatively small codebase and heavy JS use for SVG modification, the Dot game caused UI blocking in Telescope for 2-3 seconds during some SVG transformations and line renderings. We noticed UI blocking for 3-4 second intervals for "Snow Fall" as well as some UI delay as all of the startup code traces were transmitted. The Mac Pro website incurred the most significant UI performance delays for up to 20 seconds while traces were being processed. In the future, Telescope's performance can be optimized by collating and garbage-collecting repeat invocations (see Chapter 5).

4.5. Exploratory User Study

Having demonstrated Telescope's capabilities, we evaluate Telescope's use to answer the following research questions:

RQ2 What web programming design patterns are users able to elicit using Telescope?

RQ3 What usage strategies do users employ while discovering a web interaction with Telescope?

4.5.1. Method

We conducted an exploratory study with five student software developers at Northwestern University to understand how they can use Telescope to learn from professional websites. Three of the developers stated they had at least 3 months of professional web development experience through internships. The other 2 stated they knew enough to create website and setup simple JavaScript interactions with libraries like jQuery or Bootstrap. Each user was interviewed about their technical experience and trained to use Telescope for 5 minutes on toy examples. They were then asked to explore 1–3 websites on their own in the time remaining. Sessions lasted 45 minutes each, and each participant was compensated $20. Each participant provided a screen recording with audio for the entire test.

We chose three websites and interactions from the seven in the case study (see Figure 4.9) that had fun or clever dynamic UI's whose implementation involved at least two functional UI transformations. For each website we observed how users reacted to aspects of code we identified as highly relevant to the UI interaction through prior review. We prompted users to think aloud during their interaction and periodically asked them open ended questions such as, "What can you tell me about the way the feature is constructed?",

"What coding lessons or decisions can you identify?", and "How does Telescope help in understanding this feature's source?"

4.5.2. Results

In our exploratory study, Telescope helped junior developers quickly identify coding design techniques and programming concepts in the unfamiliar code underlying professional websites, while also inspiring additional discovery. This section addresses our research questions with results from user observations and think-alouds during user testing.

4.5.2.1. RQ2: Web Design Patterns Recognized. All five users identified front-end web design patterns including lazy media loading, mouse position tracking, class-toggled effects, library usage, and animation. A user said, "Seeing what this is helps me know how to approach this problem design-wise (code design)." Four users found an example of lazy-loading and mouse position tracking in XKCD's map viewport by moving the JS timeline constraints past the startup code activity and watching the JavaScript call counts while repeating the map-drag interaction. Two users discovered class-toggled effects by watching the HTML view change during Tumblr's scroll effect, then clicking the HTML query markers to see what JavaScript queried the section element; however, these users did not look into the CSS properties related to the effect. All users identified instances of library usage in the Dot game's RaphaelJS line-drawing or each site's jQuery references. Two users found how to construct simple animation through Tumblr's use of jQuery animate.

Seeing in-context front-end architectural patterns working together helped users learn from examples. Users identified patterns for interactive UI including event-driven design,

function closures, and state maintenance. Before using Telescope on the XKCD map, a user said, "I know how to make event handlers, queries, and I know the syntax of JavaScript, but I'm missing the *how* of making them work together for a feature like this draggable map." Telescope enabled this user to find multiple patterns in XKCD's comic. Users intuitively found the nature of function closures in JavaScript in scenarios like XKCD's `update` function callback, which contains a `map` variable declared outside the function scope but is referenced without declaration inside the function scope. Users found alternate implementations of state maintenance: storing active state in HTML attributes on Tumblr, or storing the game state in an in-memory JavaScript object via references to `this` in Dot-To-Dot's `Dot` object.

4.5.2.2. RQ3: Telescope Strategies. We discovered a mix of strategies for interacting with Telescope that our users employed while learning from a UI with Telescope: *constrain-expand*, *copy-paste*, *watch-and-wait*, and *step-constrain-step*.

The *constrain-expand* strategy helped users focus on relevant code and other users curious about library code, external dependencies, or background code. One user said, "The detail control is crazy, because it lets me see just what modified the DOM or I can bring in background code too." *Constrain-expand* was typically used after the user gained a significant understanding of the interaction and wanted to validate their assumptions of hidden variable references or function declarations.

The *copy-paste* strategy emerged when users either tried to play with a portion of code themselves or wanted to see the external media referenced by JavaScript and HTML. The XKCD and Dot-to-Dot websites load external images and audio, which are referenced in Telescope's HTML view. Users copied links to the media to view them as whole files

outside the interface. Users copied portions of JavaScript code to an external IDE to see which variables were declared in scope and which ones were not.

We also observed users adopting two other strategies that were less successful in our test. With *watch-and-wait*, users watch the Telescope interface update without adjusting any controls. This made it difficult for users in our test to find interaction code amidst setup code, but could be effective when used on websites with little setup code. Another strategy is *step-constrain-step*, where users narrow the timeline min and max to examine one second of execution at a time. This made it difficult to see calls from high order functions which span multiple seconds, but it was effective in reducing noise from background functions.

Users were able to quickly and easily locate relevant source code for complex interactions. Averaging less than four control toggle changes to find code pertinent to their interaction, users excelled in parsing through fine-tuned views of JavaScript. Three of the users continued exploration past their goal to discover additional coding concepts. One user said, “Once I found that Raphael was being used, I wanted to dig deeper to see how it was configured to make a line wobble.”

Developers with less JavaScript experience chose Telescope’s HTML pane as a reference point, whereas developers with more experience spent time carefully gaining insights from JavaScript implementation decisions. Telescope’s line drawing features helped less experienced developers explore JavaScript from an HTML reference point they felt familiar with. A user said, “This would become my starting point over forums/tutorials — I might even use it on a tutorial’s solution instead of reading the tutorial’s example code."

Telescope's detail expansion feature helped developers with more experience learn architectural decisions about the code. Less experienced developers focused on understanding how to recreate the effects in the default, least detailed view.

4.6. Limitations

4.6.1. Instrumentation Scope and Applicability

While Telescope currently supports UI discovery on many popular websites, some limitations prevent it from working on all websites. Scripts that are loaded via lazy-loaders can escape Telescope's instrumentation if they are not present on the page when the Sleight-of-Hand method takes place. Lazy script loaders use URLs in strings to append to scripts to the DOM asynchronously. Telescope will capture and rewrite sources at the time of its invocation, but scripts loaded later are beyond the rewrite scope. However, Telescope does capture calls to load the scripts. Lazy intercepts can be added to Telescope in the future through request blocking and source redirection.

Telescope only instruments and monitors the top-level website frame. Subsequent or nested iFrames were omitted in this project, as iFrames are typically used to embed external content. Future versions of Telescope can recursively traverse the DOM to instrument and listen to traces from iFrames.

While calls to their API's are captured in Telescope, the rendering logic underlying HTML5 Canvas, OpenGl, Flash, Silverlight, and Java Applets are not visible to Telescope. Instrumenting these technologies through website source rewriting is currently not possible.

4.6.2. Performance

Unlike Unravel, Scry, and FireCrystal, Telescope depends on third party servers and lengthy instrumentation processes for large files. The performance overhead required for source instrumentation is considerable on modern hardware and exceeds the capabilities of web browsers. A rich UI might contain fifty thousand lines of code, which can require up to three minutes to instrument. While instrumented files are cached to speed up repeat-loads, future versions of Telescope could optimize the instrumentation process for larger script transformations by indexing and caching common file subsets like modules and libraries.

Telescope was unable to capture UI interactions on several test sites due to memory limits and website implementation techniques. Telescope sessions for the Netflix and Spotify web players exceeded the browser's memory limitations, resulting in truncated trace data. Amazon's use of iFrames, Airbnb's content security policy, and Forecast.io's app cache script loading prevented Telescope from collecting meaningful trace data. Telescope successfully displays interactions from Google web products, but we found their minification techniques especially difficult to read due to the minification of HTML attributes in addition to JavaScript. In future work, memory problems can be overcome by disabling source tracing and logging for portions of a website until needed, CSP's can be filtered out by debugging proxies, and given enough interest, a crowd of experts could help identify minified HTML attributes.

4.6.3. Code Explanations

Telescope instruments and examines only client-side code and does not curate or explain the code. Further, Telescope does not process or interpret CSS. Existing tools like Theseus and Scry help users discover how server-side code is executed and client side CSS transformations alter the DOM rendering [**53, 15**]. Future versions of Telescope could incorporate technologies like Tutorons in order to explain the code in the context of active traces [**38**].

4.7. Conclusion

Having demonstrated the effectiveness of Telescope for helping web developers discover implementations underlying UI interactions, we revisit techniques that contribute to Telescope's effectiveness.

4.7.1. Design Principles for Understanding Unfamiliar Code

The design of the Telescope platform evolved from three prototypes, each shaped by user feedback. Initially we aimed to deliver a code-extracting tool for delivering all code behind an interaction to the users, but providing code by itself was of little value. A participant said, "I can finally see everything that happened, but I don't know what it means." Each subsequent iteration incorporated techniques to present JavaScript and HTML to the user in a way the didn't overwhelm them, which shaped Telescope's three design principles: (1) Bring together relevant JavaScript for an interaction into a single composite JavaScript view. (2) Give the user control over the amount of JavaScript detail they wish to see for any given time frame. (3) Provide affordances to visually link functionality end-to-end,

connecting active JavaScript to queried HTML and components in the rendered website. Evaluating the current prototype showed success in helping junior developers understand UI's. All users were able to identify UI engineering concepts in unfamiliar code, and seeing architectural patterns in-context helped users identify how programming techniques can be used to construct a system.

4.7.2. Enabling UI Discovery

Advancing related work [**53, 67, 39, 15, 2, 5, 31, 32**], Telescope's live tracing and source view constraints helped users identify and understand code supporting an interaction. As a user interacts with a website's UI, Telescope receives trace information and processes it into HTML and JavaScript views for the user. The display of these views are controlled by JavaScript load order, detail, and time constraints. Default settings show the user a focused view of JavaScript responsible for modifying the DOM. Clicking code markers draws lines connecting JavaScript to HTML, helping the user see how JavaScript manipulates the DOM for a desired outcome. Evaluating the UI discovery in our case study, we found that the source code needed to understand a complex UI behavior is often 150 lines or less.

4.7.3. Toward Sensemaking Scaffolds

This chapter provides a contribution in creating interactive learning materials from professional websites for users wishing to become professional contributors in web development. But in creating the low-barrier learning materials, relationships between logical components in the code are either hidden or lost during source filtering. Using Telescope, learners

were able to find and recognize design patterns and developed strategies in finding new patterns with Telescope, but were not provided with scaffolds to make sense of components or relationships in the code. The next chapter introduces Isopleth and describes a new contribution for scaffolding users into a sensemaking process where they opportunistically compose an understanding of complex JavaScript artifacts and their relationships.

CHAPTER 5

Isopleth: Mixed-Initiative Sensemaking in Web Application Code

This chapter presents the third application towards RALE, Isopleth, which is a web-based platform that enables a mixed-initiative sensemaking process by combining system and user-generated content to support learners as they make sense of complex JavaScript features in professional websites. This chapter contributes a technique to overcome Unravel and Telescope's limitations in making sense of complex JavaScript artifacts whose relational structures are hidden or obfuscated. The source code for Isopleth is openly available [1]. All uses of "we", "our", and "us" in this chapter refer to coauthors of the aforementioned paper.

Unravel (Chapter 3), Telescope (Chapter 4) and prior approaches including Scry **[15]** and Theseus **[53]** reduce the complexity of professional code by surfacing relevant information and provide methods for walking through code in execution order. But in doing so, they lose the structure of how code constructs work together to implement a feature. Since JavaScript functions are often executed asynchronously, visualizations of execution order like those provided in Scry provide little information about the conceptual structure of web programs. One could understand the structure by walking through the entire execution path as they might when debugging, but this can contain thousands of steps for professional examples. Surfacing relevant information (e.g. top-level invocations, functions with high call counts) is a reasonable approach for identifying important

[1] Isopleth Github `https://github.com/NUDelta/Isopleth`

functional components, but hides lower-level functions that are the necessary bridges for understanding how components work together to produce a feature. In order to bridge the knowledge gap for inexperienced developers, scaffolds must provide rich representations of the underlying code structure [**70**] and support multiple ways of visualizing code to help learners develop expert models [**10, 78, 3**].

Isopleth is a web-based platform designed to scaffold novices as they make sense of complex JavaScript in professional websites. At Isopleth's heart is the JavaScript call graph that is produced by the learner's interaction with a feature on a professional website. In this graph, a node represents a set of collated invocations of a function, and an edge represents a parent-child call relationship or an asynchronous binding. The Isopleth interface, shown in Figure 5.1, supports three central activities that correspond to the three characteristics presented in the Design Arguments section: (1) learners can explore functional and event-driven relationships using the *condensed call graph* and *source frames*, (2) learners can view functionally-related slices of the call graph using *facets*, and (3) learners can manipulate these representations to reflect their current understanding. We describe each of these activities from the user perspective in the sections below.

5.1. Motivations

Isopleth presents a new method for scaffolding a learner's sensemaking process as they work to understand complex professional web applications. Before detailing our specific approach, we first highlight unmet needs that are not supported by current approaches that we aim to support with Isopleth.

5.1.1. Existing Tools for Professionals and for Learning from Simpler Examples

One class of tools has been designed to support professional developers foraging code from resources provided continuously via the web. Brandt et al explored how programmers leverage online resources to support the development process, opportunistically transitioning between web foraging, learning, and writing code [**9**]. They built on this work to develop Blueprint, a web search interface integrated into a development environment to support searching for relevant code examples from forums, blogs, and tutorials [**8**]. Fast and Bernstein designed Meta, a Python language extension that allows programmers to share and compare their implementations of utility functions [**26**]. These approaches support developers finding and reusing code snippets (i.e., foraging), but are not explicitly designed to support learning the underlying programming concepts and design patterns. Specifically, they do not provide affordances to help learners build on their existing understanding [**56**] or reason about the structure of code [**70**]. Moreover, in an effort to make examples easier to find and reuse, these tools generally support foraging from curated examples (e.g., tutorials) and utility functions. They do not make readily available learning experiences out of professional web applications, which are more difficult to understand in the absence of additional sensemaking scaffolds but which embed a richer and more diverse set of professional web development practices.

Another class of tools has been designed to help aspiring developers understand localized snippets of code during the learning process. Systems such as Tutorons [**38**], Gidget [**51**], WebCrystal [**17**], Whyline [**47**], and Dinah [**36**] provide question-answer workflows to help learners resolve questions about a program's state or effects. Source

visualization approaches such as Gliimpse [**24**], PyTutor [**37**] and Bret Victor's learnable programming [**89**] contribute techniques to expose cause-and-effect relationships and state actualization. While many of these tools provide effective sensemaking scaffolds for supporting aspiring web developers, they are designed for supporting learning from curated or simplified examples. These tools can help with learning to write functional code, but progressing further requires learners to engage with more complex, professional examples that provide opportunities for learners to think in the modes of the discipline, e.g., *by reasoning about how multiple code components coordinate to achieve a feature, that is beyond the scope of these tools.*

5.1.2. Existing Tools for Learning from Complex, Professional Examples

Closest to our work, tools such as fireCrystal [**67**], Scry [**15**], Unravel [**39**], and Telescope [**40**] are designed to help (experienced) developers discover how features in complex professional examples are implemented. FireCrystal, Scry, and Unravel support developers performing *feature location tasks*, that is, identifying code most responsible for producing an interactive visual effect on a website. Built for examining complex web applications, these tools make it easier to reference the specific JavaScript, HTML, and CSS involved in changing the DOM. For example, FireCrystal and Scry allow users to record UI interactions of interest, and provides a timeline visualization for users to explore how state changes in response to JavaScript calls. Unravel takes a different approach, whereby users scope which elements to observe and the system automatically reduces observations by aggregating and displaying most likely to be relevant sources first, as ordered by call counts. While useful for finding entry points into features of interest, these tools are insufficient for

helping developers uncover the conceptual structure of web programs. First, these tools only instrument code that queries the DOM, but much of what makes a feature work is in its implementation lies beyond DOM-touching code in events, timing, data-retrieval, and data management. Second, by slicing code by points in time during execution, these tools hide how functions are bound, passed, returned, and invoked asynchronously across time, which is necessary for learning how to implement the feature. Finally, complex interactive behaviors on professional websites can contain visual effects that produce hundreds or thousands of visual changes in a split second (e.g., see histography.io); for such behaviors, the task of locating a particular change through navigating a timeline is akin to locating a needle in a haystack.

Overcoming some of these shortcomings, Telescope **[40]** identifies all code relevant for a feature to provide developers with a comprehensive yet condensed view of the code. Telescope collates top-level invocations across source files into a single view, and provides affordances for tuning this view to see details beyond DOM-touching code. While this approach makes it possible for more experienced developers to understand features in complex professional examples by making visible all of the most relevant code, it hides the lower-level functions that provide the necessary bridges for less experienced developers to understand how components work together to produce a feature that more experienced developers can readily infer based on their prior knowledge. While Telescope can also be used to surface lower-level details, the resulting sea of 'relevant' source code can quickly become overwhelming without additional scaffolds to help learners identify meaningful, functional components and understand how they connect.

We expand below on the challenges in learning implementation techniques across three specific areas of web development with these existing tools: event bindings, web application design, and dynamic interactive features. While these areas represent different classes of problems in frontend web development, they all require composing an understanding of the relationships among functions and components that together realize a feature:

- *Event bindings*: Event bindings in JavaScript are used to realize a wide variety of interactive behaviors (e.g., show-picture-on-scroll), and can vary in complexity from simple DOM event listeners to complex constructs like async callback queues. Understanding these concepts in unfamiliar code is difficult with existing tools. First, the relevant source code can be spread across many files, making it difficult to find. Second, async bindings are not apparent, making it difficult to determine if one component is related to another. Tools such as Telescope [**40**] surface the source code that were invoked during a UI interaction, but they do not provide the relational links between constructs (i.e. async bindings) to help users make sense of the source.
- *Web application design*: Modern web applications implement design patterns with architectural decisions that are difficult to discover in unfamiliar code without helpful sensemaking scaffolds. For example, a search feature may populate previous searches into its autocomplete bar by issuing queries to a server tied to a user's history, or adopt a lightweight, per-user caching strategy that queries a user's localStorage. One challenge to surfacing software design patterns is the separation of concerns; a pattern may be implemented as classes and methods distributed across files, and referenced in many distinct file locations. Another

challenge is that understanding a software's design often involves not only finding logical components, but understanding their relationships to form a conceptual idea of a larger pattern. Existing tools such as Scry [**15**] provide ways to step through code execution, walk through files, and see what code is being called as the web UI changes, but these tools primarily support localized feature location tasks (i.e., finding the code that changes the DOM) but do not support making sense of relationships across multiple components such as setup code, event bindings, AJAX calls, and DOM queries. Lastly, the inspection interface of Scry and other related tools are mostly read-only, limiting the learner's ability to manipulate the underlying representation while forming a mental model of the code under inspection as would be helpful.

- *Dynamic interactive features*: Websites such as Histography.io, the New York SkyLine article, the Making it Big Article, and Stripe's landing page make use of dynamic interaction and visualization techniques that interleave graphic transitions and content transformation that often prove to be the most difficult to reverse engineer. First, such complex features often integrate several technologies, including HTML, Canvas, CSS, WebGL, visual media, and JavaScript. Given each technology's ability to make objects appear to move dynamically in a web view, a learner's starting assumptions about how the interactive feature is implemented could vary greatly from its actual implementation, both due to its complexity and the fact that there are often many logical ways to achieve the same effect. Second, complex interactive behaviors on professional websites can contain visual effects that produce hundreds or thousands of visual changes in a

split second (e.g. on scroll or drag), and each function invocation may only reveal a small portion of the larger purpose of the function. While existing tools such as Telescope, Scry, Unravel, and fireCrystal [**40, 15, 39, 67**] provide affordances to follow JavaScript and see its links to HTML modification, it remains difficult to see how data flows through the functions and how functions relate to one another amid large numbers of function invocations.

5.2. Contributions

Beyond Unravel and Telescope (Chapters 3 and 4), additional challenges remain in helping inexperienced developers make sense of complex JavaScript relationships, design patterns, and code constructs. Unravel supports identifying features in the source code of professional websites, and CDT allows users to inspect features with a profiler [**33**]. Telescope can extract web features into reduced examples that remove complexity. However, beginners lack the conceptual knowledge required to make sense of the undocumented code and JavaScript call graphs provided by these tools. Learning to implement web features is particularly challenging because they are composed of many small components working together, and each tool provides a limited view of how these components work together. Existing tools are not designed to extract programming concepts for learners or highlight how solutions are structured to support the development of expert models of programming constructs and strategies.

To address these challenges, *Isopleth*, a web-based platform that helps learners navigate and filter call relationships in front-end JavaScript implementations interactively so that they can develop conceptual models of complex code examples (see Figure 5.1).

First, Isopleth highlights how solutions are structured by exposing hidden functional and event-driven relationships between code components through a *condensed call graph* and detailed *source frames*. These affordances help learners understand how functions are bound, passed, returned, and invoked asynchronously. Second, Isopleth extracts programming concepts for learners by leveraging automated techniques to surface *facets*, or code constructs defined by inputs and outputs. Instead of requiring developers to conceive of their own queries of the call graph, facets provide default patterns that allow learners to easily identify or exclude code for handling setup, mouse and keyboard events, AJAX calls, and DOM changes. Third, Isopleth enables learners to manipulate the provided representations of code by rearranging invocations and composing their own invocation labels on the call graph, and adding comments and editing code in source frames. These affordances provide further support for learners to understand the various components of a complex code example and their connections, and to use the understanding they have built to further their investigation.

The core conceptual contribution of Isopleth is the idea of *designing sensemaking scaffolds to help learners build conceptual models of how components coordinate to produce functionality in complex professional code.* Isopleth embeds sensemaking scaffolds—or supports and affordances specifically designed to aid the sensemaking process—to help learners produce their own understanding of code components and their relationships as they *interactively* explore, label, and filter code. Existing approaches for learning from complex, professional web examples [**39, 40, 67, 15**] help to surface the most relevant code (e.g., Telescope [**40**], Unravel [**39**]), and provide methods for walking through code in execution order (e.g., Scry [**15**] and fireCrystal [**67**]). These tools reduce the complexity

of exploring professional code, but in doing so, they make it difficult for learners to discover the structure of how code constructs work together to implement a feature. While experts may be able to recover this structure using their existing conceptual models—thereby benefiting from the simplicity of presentation afforded by such tools without loss—learners who do not have this structure in mind can struggle to make sense of professional examples without sensemaking scaffolds that are explicitly designed to help them build such conceptual models. This paper contributes a set of sensemaking scaffolds that are grounded in research from the learning sciences and program comprehension to address these conceptual knowledge gaps for learners.

The core technical contribution of this work is a *Serialized Deanonymization (SD) technique that places unique identifiers in all functions in a web application's JavaScript source to trace how functions are bound, passed, returned, and invoked asynchronously.* SD contributes a general method for tracing dynamic callbacks that is distinct from existing precise heap tracing techniques for call graph analysis (e.g., see **[47]**). While related toolkits can already fully instrument JavaScript code in professional web applications **[40, 53, 34]**, they do not capture asynchronous bindings and do not link a function invocation to its declaration context. SD identifies these missing links and adds them to the call graph, which reveals a complete picture of code activity between declaration and invocation to the learner to support their understanding of how web features are implemented. In addition to SD, the paper contributes techniques for reliably detecting and visualizing facets that connect function chains across time that are responsible for particular aspects of functionality. Since function names, function bodies, and variable names are often unreliable or misleading determinants of facets, a facet detection approach is

introduced that uses inputs and outputs to test for arguments or return values in function invocations. Further techniques are introduced for bubbling up detected facets so that they can be properly visualized even when library internals are hidden from learners to reduce complexity.

5.3. Theoretical Framework and Design Arguments

In order for aspiring web developers to understand and learn from complex professional web applications, we argue for a set of sensemaking scaffolds missing in existing tools that novices need to discover how components work together to produce a feature of interest. We present in this section a theoretical framework for designing sensemaking scaffolds that help learners make sense of complex code that draws on research from the learning sciences and in program comprehension. We then present our design arguments that use this theoretical framework to inform a set of core characteristics that Isopleth implements which embed these sensemaking scaffolds.

5.3.1. Theoretical Framework for Making Sense of Complex Code

Sensemaking refers to the process of building understanding by generating representations that explain what is known or understood [**91**]. Early work on sensemaking from information science [**23, 76, 71, 65**] focused on how individuals develop complex and accurate representations, often in the context of information-seeking and search tasks. As an extension of these early ideas, later work considered the specific challenges that learners may face in making sense of examples and artifacts, not only for seeking information but also for building conceptual knowledge in a domain.

In the context of understanding code examples, a rich body of work in *program comprehension* [**87, 13, 85, 90, 83, 69**] examines how programmers make sense of the structure of code and its functionality. While experts leverage templates and formal representations of programming constructs to make sense of and solve problems, these patterns are not apparent to novices [**1, 92, 57, 62, 18, 19, 22**]. Learning from complex code examples is particularly challenging because it requires understanding not only individual components, but also how they coordinate to solve a problem [**49**]. Novices not only lack conceptual knowledge, but also the expert strategies for constructing an understanding of a problem by examining evidence, testing hypotheses, and reflecting on findings [**95, 94**].

The learning sciences provide guidelines for scaffolds (supports and affordances) that can help novices bridge this knowledge gap and adopt more effective strategies in order to make sense of complex examples. This literature suggests that tools should be organized around the semantics of the discipline to help learners adopt expert strategies and approaches [**72**]. Tools should also build on a learner's intuitive understanding by using representations and language that connect to their knowledge and provide expert guidance to overcome gaps in conceptual knowledge [**56, 72**]. Finally, tools should provide opportunities for learners to inspect the underlying representation in different ways [**72**]. Providing multiple ways to visualize the underlying data helps learners build dense, interconnected conceptual representations [**10, 78, 3**]. Through the design of Isopleth, we considered how guidelines for designing sensemaking scaffolds, originally designed to support sensemaking during scientific inquiry [**72**], can be applied to support learning from complex professional code examples.

In addition to drawing on theories for scaffolding sensemaking from the learning sciences, Isopleth is designed to support learners applying a flexible set of program comprehension strategies. Several comprehension theories describe how programmers understand new code, using strategies such as (1) top-down from domain to source code [**13**], (2) bottom-up from statements to abstractions [**83**], (3) beacons from familiar code with plan decomposition in unfamiliar code [**85**], and (4) bottom-up through control-flow abstraction from microstructures to macrostructures to form a situational model [**69**]. Isopleth supports learners adapting such strategies to understand complex call relationships as they interactively navigate a JavaScript call graph to make sense of complex code constructs and hidden asynchronous relationships in professional web code.

5.3.2. Design Arguments

In this work, we present a tool that is specifically designed to support novices in making sense of complex professional websites. As described in the previous section, existing tools, primarily designed for experts, are not effective at helping novices build the conceptual knowledge required to make sense of complex code. To overcome these obstacles, we designed Isopleth around the set of design guidelines and theories presented in our theoretical framework. In particular, we incorporate a subset of the relevant guidelines proposed by Quintana et al. [**72**] for designing software scaffolds to support sensemaking in the domain of scientific inquiry. We also incorporate ideas from theories of program comprehension to support sensemaking in this particular learning domain [**69, 85**].

In this section, we describe each of these guidelines, discuss the related obstacles for novices, and present the high-level design characteristics that we incorporate into Isopleth to overcome these challenges.

5.3.2.1. Guideline: Organize tools and artifacts around the semantics of the discipline. Quintana et al. recommend that software systems organize tools and artifacts around the semantics of the discipline to help shape the learner's understanding of disciplinary knowledge and practices [**72**]. Since expert practices rely on domain knowledge that learners lack, they need support in understanding, recognizing, and applying these practices during sensemaking [**75, 72**]. As a result, effective scaffolds organize information in disciplinary ways to help learners approach problems the way an expert would.

Most tools designed to support the exploration of complex professional code target experts rather than novices, and as a result they do not focus on making disciplinary information explicitly visible to users. Connections such as asynchronous relationships are not visualized in systems such as Telescope [**40**], Unravel [**39**], fireCrystal [**67**], and Scry [**15**], making it challenging for novices to uncover the structure of a web program and build a conceptual understanding of how the pieces fit together. Even without asynchronous functionality, novices may lack effective expert strategies for examining how functions connect to one another, e.g., by carefully examining the input and output values between connected functions. Moreover, while existing tools aim to surface the most relevant code by hiding certain details, the 'irrelevant' code hidden by Telescope and the back-end functionality hidden by fireCrystal, Unravel, and Scry are often crucial for understanding how components coordinate to achieve functionality.

These challenges highlight a need for tools to surface and help learners understand the hidden and asynchronous relationships among code components. This helps to expose the disciplinary information required to build an accurate understanding of professional code.

1 Expose hidden functional and event-driven relationships between code components.

Isopleth implements Characteristic 1 through a *condensed call graph* and detailed *source frames*. The condensed call graph colors nodes and edges according to their semantic meaning and makes all connections, including asynchronous ones, visible to the learner. For example, this can help a learner understand the end-to-end logic involved in the infini-scroll feature of a blog website where photos are continually added to the bottom of the page by discovering the exact functions that bind DOM modification as an asynchronous response to mouse scrolling. To avoid overwhelming the learner, library code is hidden by default but connections and links among code components through library code are preserved by bubbling the links up to the nearest non-library components for display in the call graph.

The source frames further organize information around the semantics of the discipline by showing the input, output, and bindings for each individual function. Learners can also view connected source frames side-by-side to study how two functions connect to one another. In this way, source frames construct the expert practice of thinking about functions in terms of their inputs and outputs by making them visible and explicit to learners. Together, these affordances make disciplinary knowledge and practice accessible to learners to help novices build a conceptual understanding of web program structure.

5.3.2.2. Guideline: Use representations and language that bridge learners' understanding. The previous guideline focused on helping learners adopt expert conceptual models and approaches. In contrast, this guideline focuses on helping learners connect their prior knowledge to the sensemaking task at hand. Quintana et al. recommend that software scaffolds use representations and language that connect to a learner's intuitive understanding, and embed expert guidance in situations when learners lack the background knowledge required to engage in a particular practice [**72**]. Effective scaffolds describe complex concepts in ways that build on what learners know from their own experience. They also use visual representations that organize content and functionality in ways that encourage learners to focus on conceptual understanding rather than surface details.

While existing tools are effective at reducing code complexity by providing access to relevant snippets, they are not designed to bridge from learner understanding or embed expert guidance. Unravel [**39**] and Telescope [**40**] surface relevant code, but do not help novices reason about the core aspects of functionality and how they relate upon locating the code. Experts may use their conceptual understanding to quickly identify these aspects and effectively explore how the code example is structured, but novices do not have the knowledge needed to apply these strategies. This highlights a need to help novices bridge between their intuitive understanding and expert approaches. FireCrystal [**67**] and Scry [**15**] provide an entry point into code based on visual outputs, which is intuitive to novices and may help bridge understanding. However, these systems do not provide affordances to help users learn how the identified code snippets connect and interact with

other code in the web application. Learners can easily struggle to reason about how bindings and callbacks interact and cannot effectively trace program flow through a complex example using these tools.

These challenges suggest that web inspection tools should help learners use their intuitive understanding of how interactions cause visual effects (e.g., mouse and keyboard events), and organize code into functional slices that reflect how experts think about functionality (e.g., where events are bound and where AJAX calls are made). To do this, we propose to:

2 Provide visual organizers that allow learners to view slices of code that are functionally related.

Isopleth implements Characteristic 2 through *facets*, visual organizers that give novices an entry point into complex code that is based on their own intuitive understanding of the functionality. For example, a novice who is interested in understanding which code constructs are used to react to her keyboard strokes on a search bar or clicks the search button could use a *facet filter* to surface code associated with a mouse or keyboard event. These facets help bridge beyond the novice's understanding by displaying a functional slice that includes all functionality associated with the event, showing not just how an event was triggered, but also where it was bound and what functions were called in response. This encourages novices to think conceptually about how multiple components connected through the same facet interact to create a given feature. Facets also embed expert guidance by providing slices that surface content like setup code and AJAX calls, by displaying *facet labels* on nodes in the call graph. This encourages novices to explore functionality using expert approaches they might not know to consider. For example, a

novice may expect the search autocomplete to query for data using AJAX, and use facet labels to identify AJAX-related code and how it connects to other code components. In these ways, facets bridge from a novice's intuitive understanding and provide affordances that help guide novices to explore code according to expert strategies.

5.3.2.3. Guideline: Use representations that learners can inspect in different ways to reveal important properties of underlying data. Finally, Quintana et al. recommend that software scaffolds allow learners to view and interact with multiple representations of data to help them reveal its underlying properties and understand cause and effect relationships [**72**]. In the domain of scientific inquiry that Quintana et al. study, learners work with representations like tables, graphs, equations, simulations, and diagrams to make sense of scientific phenomena, often editing the underlying data to explore cause-and-effect relationships. In our domain of program comprehension, developers instead leverage representations like the textual display of the code, call graphs that show program flow, and diagrams that show relationships between classes [**87**]. Echoing Quintana et al. program comprehension researchers have also recommended that tools to support program comprehension should provide multiple views of the code [**87**].

Extending Quintana's guideline, we consider the need for learners to manipulate representations directly as they build their understanding of how a professional code example works. Previous research has shown that the process of building mental models of code functionality is iterative; understanding is built up in progressive layers and changes over time [**83**]. However, we are not aware of any tools that allow users to directly manipulate code representations by grouping related functionality or adding comments and labels

to reflect their current understanding. Tools like Unravel [**39**], Telescope [**40**], fireCrystal [**67**], and Scry [**15**] help users locate or isolate relevant code, but do not support representation manipulation.

This highlights a need for tools to allow novices to externalize their mental models of code structure and functionality by manipulating representations to reflect their current understanding:

3 Support iterative manipulation of code representations to reflect a learner's understanding as it develops.

Beyond providing multiple representations of professional web code—including *facets*, the *condensed call graph*, and *source frames*—Isopleth implements Characteristic 3 by allowing users to manipulate these representations to further support the sensemaking process. As they explore the condensed call graph, novices can label nodes to signal their purpose and drag nodes to group them in ways that are semantically meaningful. Novices can also edit labels and code comments in source frames to externalize their understanding of functionality. Finally, novices can create custom facets to define new code slices that surface functionality of interest. These affordances allow novice learners to not only explore complex professional websites using multiple representations, but also to manipulate those representations to express their current understanding of the functionality and provide beacons [**85**] to help them externalize their mental models. For example, a learner attempting to understand how a game timer causes a game-over action may discover components such as game view updates and game timing throughout their sensemaking process, and add node labels and reorganize the call graph to describe these components and support their ongoing sensemaking process.

In the rest of this chapter, we introduce Isopleth and its affordances for supporting sensemaking through source code frame views, facet-filtered call graph, and extensible filters. We detail the serialized deanonymization technique; evaluate the extent Isopleth identifies and relates facets in professional websites through an in-depth case study; and conclude with a discussion of design principles and limitations of our approach. Isopleth takes an important first step towards the development of Readily Available Learning Experiences (RALE), a conceptual approach for transforming all professional web applications into opportunities for authentic learning.

5.4. Isopleth

Isopleth is a web-based platform designed to scaffold novices as they make sense of complex JavaScript in professional websites. At Isopleth's heart is the JavaScript call graph that is produced by the learner's interaction with a feature on a professional website. In this graph, a node represents a set of collated invocations of a function, and an edge represents a parent-child call relationship or an asynchronous binding. The Isopleth interface, shown in Figure 5.1, supports three central activities that correspond to the three characteristics presented in the Design Arguments section: (1) learners can explore functional and event-driven relationships using the *condensed call graph* and *source frames*, (2) learners can view functionally-related slices of the call graph using *facets*, and (3) learners can manipulate these representations to reflect their current understanding. We describe each of these activities from the user perspective in the sections below.

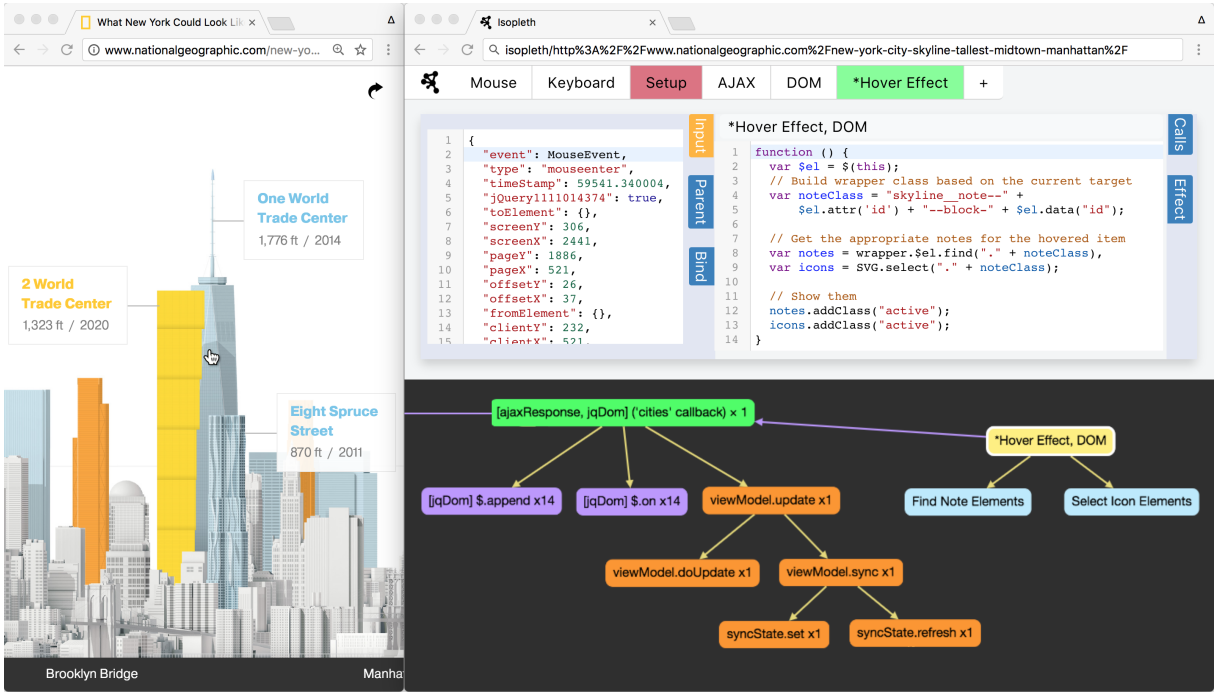

Figure 5.1. A learner is using Isopleth to understand JavaScript code constructs related to moving and scrolling their mouse on National Geographic's New York Skyline article. Once activated, Isopleth opens in a new window and continuously updates with JavaScript activity. The condensed call graph (bottom) and source frame views (middle) allow learners to explore functional and event-driven relationships between code components. The condensed call graph (bottom) displays a collated, filtered, labeled, and color-coded JavaScript runtime call graph that includes asynchronous links. Learners can manipulate these representations to reflect their current understanding by dragging and labeling nodes, editing and commenting on source code in source frames, and adding custom facet filters. By clicking on a node in the call graph, users can open source frame views (middle) which display specific function invocation states in the runtime with their inputs and outputs, parent and child calls, asynchronous declaration context, asynchronous binding, and asynchronous effect if present. Facets (top) allow learners to view functionally-related slices of code in the call graph; predefined facet filters include Mouse, Keyboard, Setup, AJAX, and DOM. Users can apply `or` and `not` operators to engage multiple facets to expose desired views. In this example, the learner added a custom "Hover Effect" facet, comments to the source code, and node labels as they made sense of components in the call tree.

5.4.1. Exploring hidden relationships through the condensed call graph and source frames

Isopleth helps learners make sense of complex relationships in JavaScript program flow through the *condensed call graph* and *source frame* views, shown in the bottom and middle panels of the Isopleth interface in Figure 5.1. Program flow is particularly challenging for learners to understand because JavaScript functions can execute asynchronously and often appear in a different runtime order than their initial source order [**2, 53**]. Further, JavaScript's functional nature means that functions can be passed by reference in arguments, return values, and closures. No previous system directly visualizes a JavaScript function's journey from declaration to binding during runtime, but this information is crucial for making sense of the conceptual design of web applications. Isopleth provides the first interface for visualizing and exploring these hidden conceptual relationships.

In Isopleth's condensed call graph, call trees are ordered from left to right by root-level invocation over time. Each node in the call graph represents an invocation of a function or a set of repeated invocations of a function in a unique call chain that have been collated into their most recent occurrence in the tree. Edges represent relationships among function invocations, and are colored to denote the relationship between the nodes. Parent-child (or caller-callee) relationships are shown in yellow; asynchronous parent-child (or declaration context, invocation) relationships in orange; and asynchronous binding sites that denote how functions are passed through call chains to produce an asynchronous effect in purple. To control what portion of the call graph is displayed, a learner can zoom and pan to identify code constructs of interest and use controls to show/hide library nodes and repeat nodes.

When the learner clicks on a node in the condensed call graph, Isopleth displays the function body in the source frame view (Figure 5.1, middle). The interface displays navigational buttons on the perimeter of the source frame view, which provides snapshots of related functions, arguments, and return values. Users can access a function's parent caller, child calls, asynchronous declaration context, asynchronous binding locations, as well as other functions the frame binds as effects. These affordances allow users to quickly access semantic information about each node in the call graph to make sense of their functionality and their relationships to other functions. When a learner clicks on an edge, both nodes touching the edge are highlighted and their respective source frames are displayed side-by-side so that learners can readily examine their source code next to one another.

As an example scenario, consider a learner Cindy who wants to understand the end-to-end logic involved in the infini-scroll feature of a blog website, where photos are continually added to the bottom of a blog after scrolling to the end of the page. Using Isopleth, Cindy sees nodes on the right side of the condensed call graph that modify the DOM. Clicking the nodes and examining the source frame shows how JavaScript queried and appended some elements. Following purple lines to nodes on the left (an asynchronous link to an invocation earlier in time), Cindy discovers the exact function that binds DOM modification as an asynchronous response to mouse scrolling, which helps her form a more complete understanding of the feature's implementation.

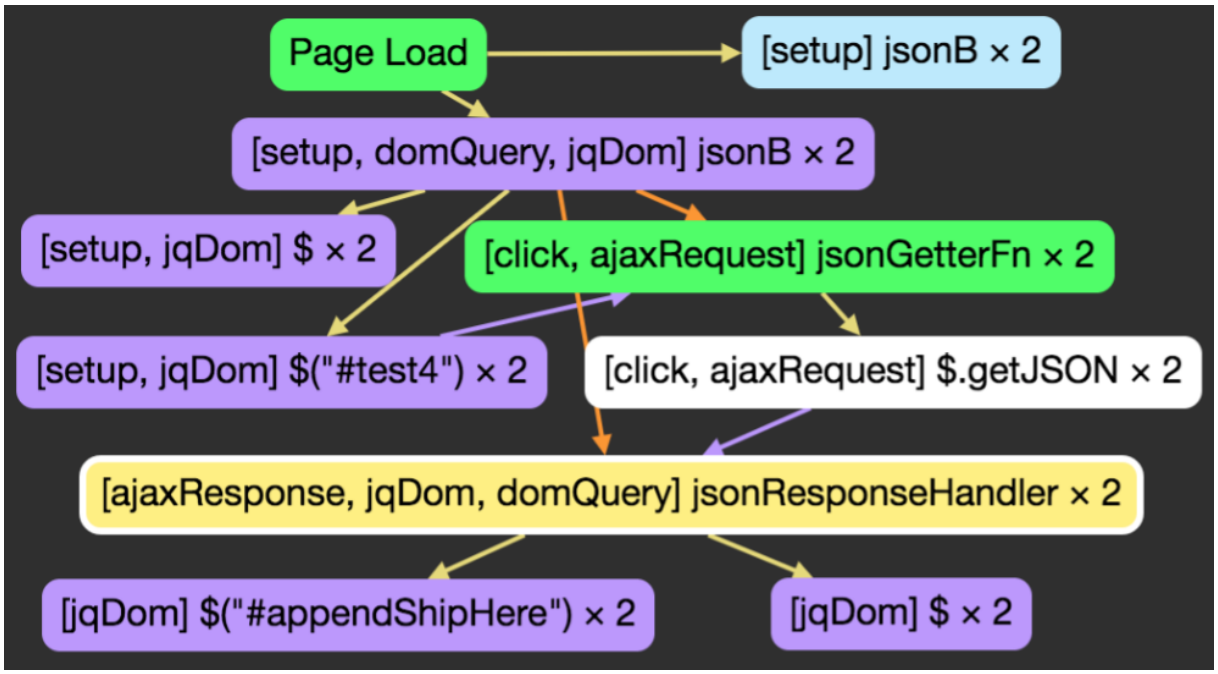

Figure 5.2. A cluster of related collated function invocations (with their invoke-counts) in a condensed call graph, manually organized here for display. Nodes are colored green for top level calls and yellow for currently-selected; other nodes are colored based on the facets they match: purple for DOM, white for AJAX, and blue for Setup. Edges in the graph are color-coded yellow for call relationships, orange for asynchronous declaration, and purple for asynchronous bind locations. In this toy-example of a lazy-loaded image, a click handler is bound on `#test4`. Upon clicking `#test4`, the handler makes an AJAX JSON request and binds `jsonResponsHandler` as the callback. The `jsonResponsHandler` queries the DOM for `#appendShipHere`, and adds the image.

5.4.2. Viewing functionally related code slices through facets

Facets provide methods for viewing functionally related code slices of the condensed call graph, exposing conceptual relationships between JavaScript functions. Isopleth includes a set of predefined facets that are shown by default, including Mouse, Keyboard, Setup, AJAX, and DOM (see Figure 5.1, top). Facets enable two affordances to support learner

sensemaking: (1) *facet labels* and coloring on the nodes in the call graph that denote their functionality, and (2) *facet filters* that can be used to view functionally related slices of the call graph.

Nodes in the condensed call graph are labeled and colored according to the default facets they match, e.g., purple for DOM, white for AJAX, and blue for Setup. Top-level calls and currently selected nodes are colored green and yellow, respectively. Nodes with multiple facets take the color of the last invocation type. Figure 5.2 illustrates a cluster of related, collated function invocations whose nodes are colored by their facets and whose edges are colored based on the relationship among the nodes they connect. Facet labels and coloring are designed to help learners quickly find nodes related to intuitive concepts like mouse and keyboard events, and also notice the features like AJAX and setup nodes that experts find important.

To further reduce the complexity of the call graph, Isopleth provides facet filters that allow learners to view functionally related slices of code. When a facet filter is selected, the call graph at the bottom of the Isopleth interface is filtered to display a subgraph that includes functions related to the selected facet, along with their parent and child relationships. This allows learners to quickly see all functions related to mouse and keyboard events, all setup code and AJAX calls, and all functions that modify the DOM. To avoid overwhelming learners, all library code related to a given facet is hidden by default. Learners can engage multiple facets to expose desired views by joining facet filters with `or` operators (upon left-clicking) and `not` operators (upon right-clicking). Facet filters are designed to highlight conceptual relationships between related functions that are not apparent in tools that visualize execution order exclusively.

As an example scenario, consider a learner Alice who wants to discover which code constructs are triggered when interacting with a search bar. Using the facet filters, Alice left-clicks the mouse and keyboard facet filters to activate an `or` condition and right-clicks the DOM filter to activate a `not` condition on DOM-querying nodes to focus on backend functionality. The call graph updates to show Alice keyup and click handlers as top level nodes that, respectively, correspond to functions that react to her keystrokes into the search bar and clicks on the search button. Examining these nodes and their descendants allows Alice to quickly see the distinct code constructs that support interactions with the search bar as well as any code that is shared and reused between these constructs.

5.4.3. Manipulating representations to reflect understanding

In addition to providing learners with multiple representations to support sensemaking (facets, the condensed call graph, and source frames), Isopleth supports learners manipulating these representations to reflect their current understanding. While exploring the condensed call graph, a learner can drag nodes to rearrange them in ways that have meaning to the learner. Node labels and source frame views are also editable, and update referentially. When examining source frames, learners can label the node, add comments to the code, name anonymous functions, and even refactor code. To support learners' building on their existing understanding, Isopleth referentially updates learner-inputted changes throughout the graph so that they appear whenever the source is referenced by other nodes. Moreover, Isopleth's interface provides edit cues, such as placeholder boxes and blinking code cursors, that signal such changes are possible and that encourage learners to make edits as would support their program comprehension strategy.

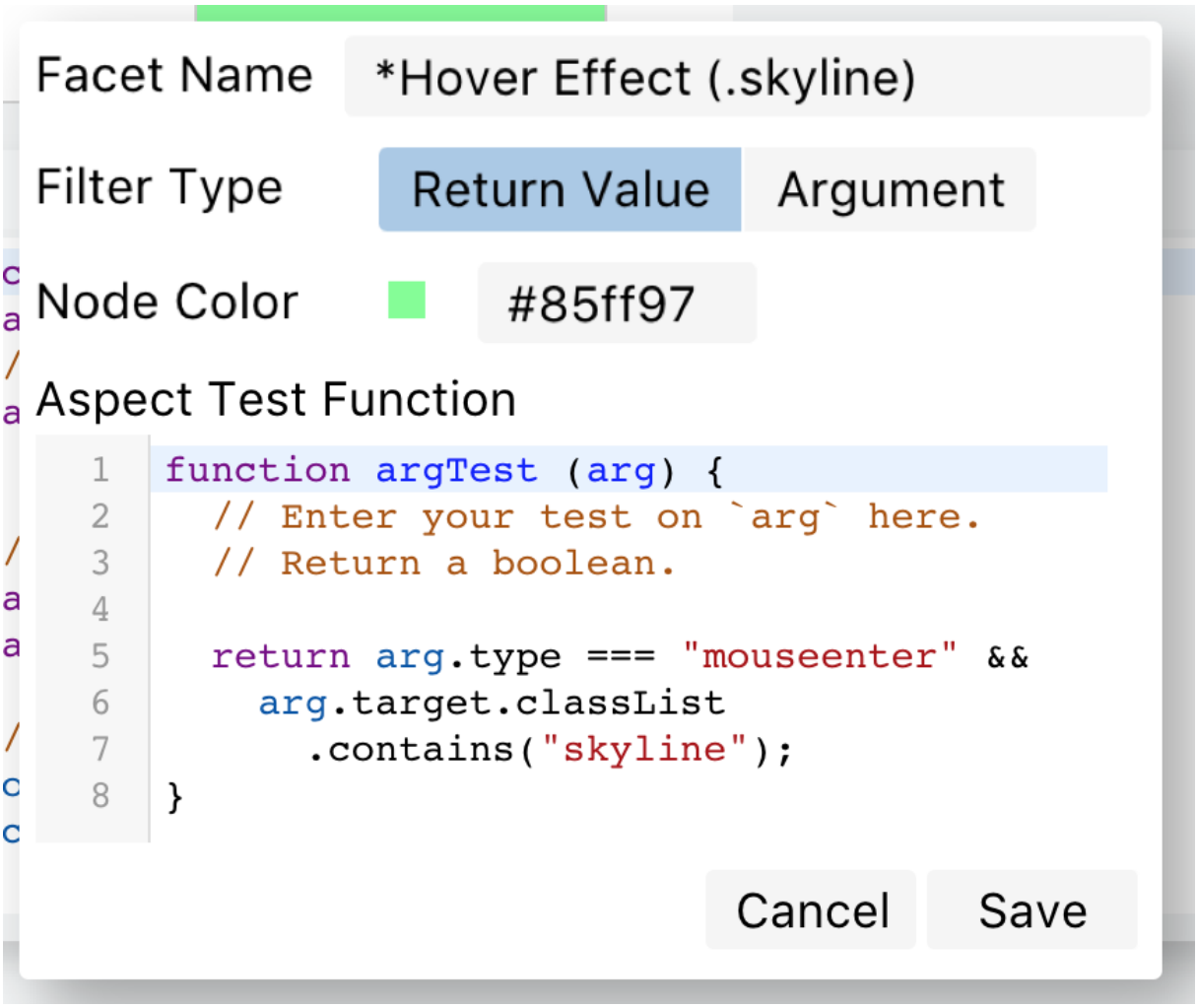

Figure 5.3. A learner is creating a custom facet filter through the facet creator view. Facets are functional input-output schemas; creating a custom facet thus involves writing a test for arguments and return values to identify function invocation nodes that match such conditions on the argument or return value. Learners also assign a node color for display in the condensed graph. In this example, the learner creates a custom facet to filter for code constructs responsible for the hover effect upon mousing over buildings in the National Geographic's New York Skyline visualization (the effect on the left of Figure 5.1).

Isopleth also allows learners to create custom facet filters that help them explore functional slices beyond the set of default facets (see Figure 5.3). Dynamic interactions with websites are initiated by either user inputs or scheduled inputs, and can produce corresponding changes to the DOM as outputs. Since the set of possible inputs and outputs is unbounded, it is infeasible to automatically identify all facets that might be relevant to a learner's sensemaking process on a particular professional example. Custom facet filters thus give learners the flexibility to create filters on inputs and outputs as would support their specific sensemaking process. For example, a learner can type the text "dog" into an autocomplete field on a professional website, and then create a custom facet that filters for "dog" as an input to a function to trace how the string "dog" is passed from an input, to an AJAX request, and finally into a result list to understand how the autocomplete search works.

As an example scenario, consider a learner Mark who wants to understand how the game timer causes the game-over action in an HTML Tetris game. He first defines a custom facet for timer events. He then finds the final timer event on the right of the call graph and notices 15 nodes underneath. He does not immediately understand the functionality of the top level node, so he clicks a few other nodes in the tree to find familiar code. Mark finds a node three nodes down and works through the source, adding comments about an object state being updated and labels the node "Game State Update." He explores and labels two other related nodes, and identifies a link between the game state and the timer methods. This helps him understand the higher-order design pattern of separating concerns, such as game view updates and game timing.

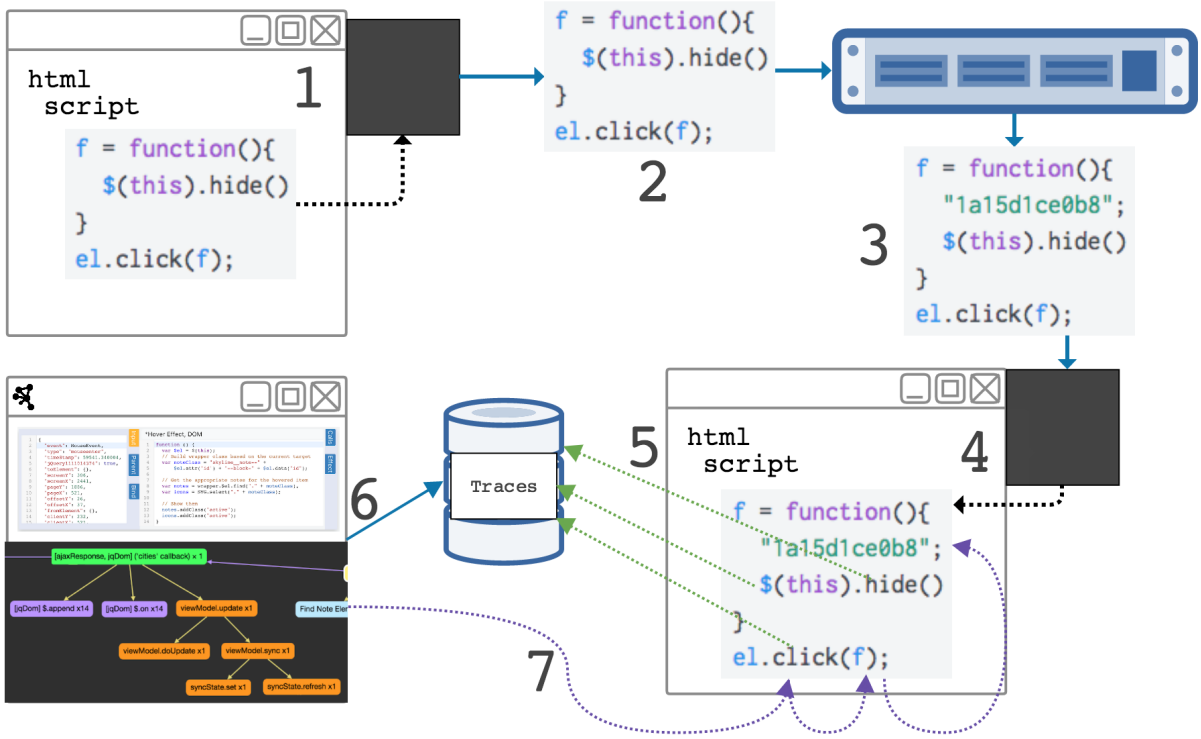

Figure 5.4. The Serialized Deanonymization technique pictured above is a 7-step process for tracing an anonymous JavaScript function's path from creation to invocation. (1) Website JavaScript is extracted and (2) sent to an instrumentation server. (3) UUID's are injected into all function bodies. (4) The source is injected into the page and (5) re-rendered, sending trace activity continuously to a database. (6) Isopleth queries traces for call graph calculation and (7) mines arguments and return values for function serials to discover how functions were passed and bound.

5.5. Techniques for Exposing Hidden Links and Identifying Facets

Isopleth supports sensemaking of web applications by automatically (1) exposing hidden links among code components and (2) identifying functionally-related facets that can be used to filter the call graph for the learner. In this section we present the technical methods that support these two core functionalities.

5.5.1. Exposing Hidden Links with Serialized Deanonymization

Tracking the lifecycle of a function from creation to invocation is especially difficult in JavaScript, which allows for functions to be declared, passed, invoked, and manipulated during runtime both synchronously or asynchronously. For instance, a function could be created and passed by reference through a complex library event system before being bound to a UI event. While related toolkits can already fully instrument JavaScript code in professional web applications [**40, 53, 34**], they do not capture asynchronous relationships to fully link a function invocation to its declaration context. Current tools can identify the declarative scope of the function and its calling scope [**53, 34**], but the function's journey from creation to invocation is missing. For instance, this makes it difficult to see the entire scope of code in popular event-binding callbacks common in JavaScript, e.g., `object.on(‘‘some event’’, anonymousCallback)`.

The goal of Serialized Deanonymization (SD) is to trace the lifecycle of anonymous functions. Our strategy is to add unique ID's to each function at instrumentation time, then record all instances of the function that appear in serialized arguments and return values at run time (i.e. from the Function.toString prototype, which provides the string representation of the function — including our injected UUID).

We detail SD in the steps below (See Figure 5.4):

(1) Initiate public website instrumentation using the Sleight of Hand (SoH) technique [**40**] to instrument a website's source code.

(2) Extract the source for instrumentation via website-instrument-swap-and-trace (Wisat) architecture [**40**].

(3) While applying Fondue tracer code [**53**] to the JavaScript source, for each function body in the JavaScript abstract syntax tree, prepend a unique ID as a terminated string expression to the function body.

(4) Reinsert the source via Wisat architecture and complete the SoH technique, rendering the instrumented source.

(5) Collect function trace activity, including logs of our newly added serials if present in arguments or return values.

(6) Load trace activity for call graph calculation.

(7) Make purple SD graph edges (See Figure 5.2) by backtracing function invocations through the logs of arguments and return values from other function traces.

5.5.2. Identifying Facets with Facet Tree Decoration and Node Collation

With current tools [**53, 15, 67**], we can see source code, individual variable states, and active lines at certain points of time, but this does not reveal meaningful *facets* that connect function chains across time or code constructs responsible for particular aspects of functionality. In order to support exploring a call graph based on facets (such as DOM, Setup, or AJAX), we need a reliable method for determining whether a function is related to a facet. Function names, function bodies, and variable names are often unreliable or misleading determinants of facets because programmers may struggle to create well-named variables [**30**] and minifiers swap variable names with short system-generated names that hold no semantic meaning [**6, 77, 86**].

To reliably identify facets, we define facet filters based on inputs and outputs through tests of arguments or return values in function invocations. For example, Isopleth's predefined facet filters look for `EventTarget` arguments for the Mouse facet, `onload` arguments for the Setup facet, and `XHR` objects in return values for the AJAX facet. Invocations that pass these tests are labeled in the call graph, and displayed along with their parents and descendants when a facet filter is applied.

Since JavaScript libraries typically wrap API concepts deep within legacy-supporting constructs [**44**] such as XHR formation (for AJAX) and MouseEvent binding (for Mouse), defining facet filters in the manner we described means that facets are often detected through library code. While Isopleth's frontend interface needs to hide such library internals from the learner to avoid unnecessary complexity, our facet filters must operate on these constructs in the backend to reliably identify facets. For example, learners need to know that calls to `$.ajax` are AJAX facets, even though internally these facets are often hidden in library wrappers around the JavaScript XMLHTTPRequest API.

To address this need, Isopleth propagates facet labels from high-level nodes to descendants and low-level nodes to ancestors; this helps users to see the responsibilities of a particular branching path in the call graph regardless of their search strategy. When Isopleth detects a facet in a return value or argument, it traverses the call graph to label nodes in a call-chain (i.e. a DOM query). For argument values, it begins the search at the node with the argument, and if the node is library code, it traverses descendents until finding non-library root nodes to mark with the facet, e.g. an AJAX response. Similarly with return value facet identification, if Isopleth detects the return value within library code, it bubbles the facet up the tree until finding non-library code to label with the

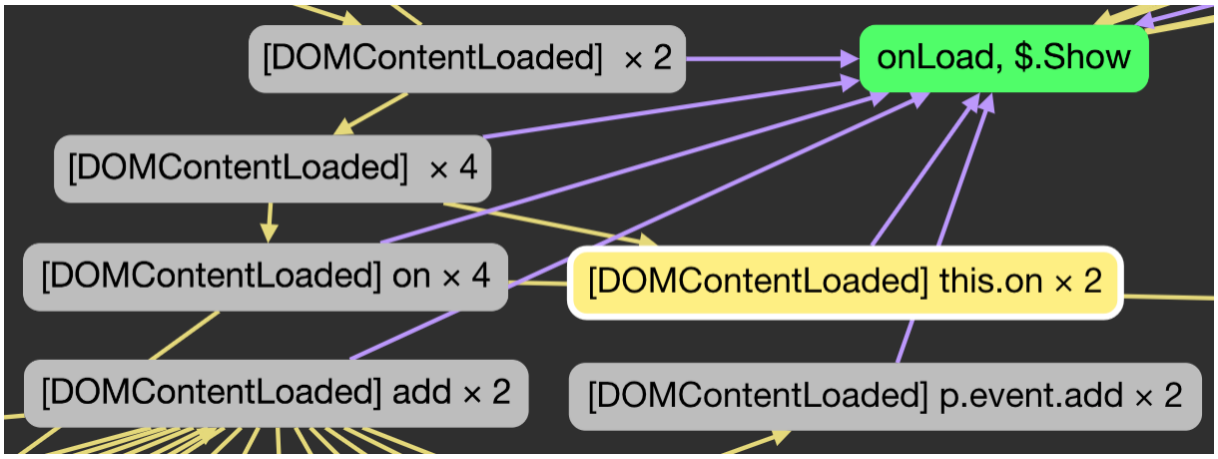

Figure 5.5. This figure shows how Serialized Deanonymization allows for a DOM-modifying facets to be bubbled up out of a library call. After removing library code filtering from the condensed call graph, we can see DOM-modifying functions existing inside the library (grey nodes; and the currently selected yellow node). The facet is bubbled out of library code to the green node that initiated the DOM changes (outside of jQuery) by following the asynchronous links (purple lines).

facet. Figures 5.5 and 5.6 provide two examples that show how facets are bubbled out of library code through asynchronous links (using SD) and through function invocations respectively.

5.5.3. Implementation

Activating Isopleth follows the same workflow as Telescope [**40**]. A user navigates to a website of interest in a browser (i.e. currently supported in Google Chrome), activates source instrumentation via a browser extension, and explores Isopleth at a newly launched URL. Isopleth then communicates with the instrumented website to gather traces and generate a call-graph. To do this, Isopleth uses the Wisat architecture [**40**] to instrument

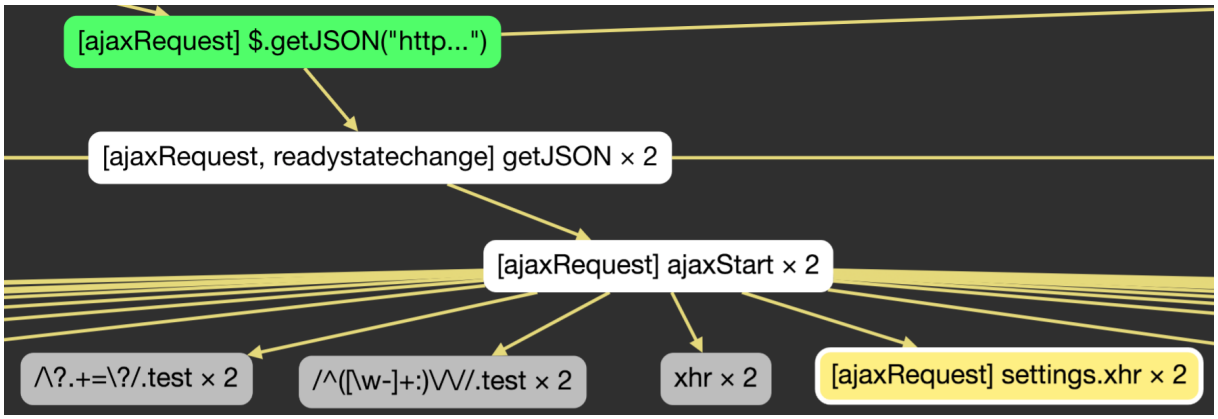

Figure 5.6. This figure demonstrates how facets are bubbled out of library code through function invocations. During call graph calculation, if a facet is detected in a library, we bubble the facet up to the first occurrence of non-library code to help learners identify the facet roles of library API calls. After removing library code filtering, we can see how the jQuery library API surfaces a `getJSON` wrapper-method (green node, not inside library code) which is decorated with the AJAX facet that was actually detected at a lower-level through the yellow node in the library code (i.e. `getJSON` actually delegates to the XMLHTTPRequest API through which we detect the facet).

websites and extends Fondue [**53**] by adding unique identifiers through the Serialized Deanonymization technique.[2]

To streamline source instrumentation on more public websites than was supported by Telescope, we also extended the Wisat architecture to block Content Security Policy (CSP) headers via Chrome network intercept requests.[3] CSP headers are sent in advance of the HTML and enforced by a web browser to prevent the modification of JavaScript outside of predefined domains or rules specified. Isopleth needs to modify the JavaScript in order to instrument the runtime on the page, and CSP headers block this functionality.

[2]Github: Isopleth's Fondue API `https://github.com/NUDelta/Isopleth/tree/master/fondue-api`
[3]Github: Isopleth's CSP Modifier `https://github.com/NUDelta/Isopleth/blob/master/chrome-extension/background.js`

By blocking CSP headers, we allow Isopleth to alter the source code and instrument the JavaScript for analysis prior to appearing in the call graph.

To produce the condensed call graph, we implemented the following techniques for detecting, collating, and throttling repeat call branches in the call graph. We collate nodes into their most recent call if they have identical source, identical parent source, and identical children source. Call links (direct and async) from each collated node are appended to the most recent called node. Arguments and returns values are collected in order and are viewable through the argument and return value buttons in the source frame view. In some feature-rich applications such as the New York Skyline article, we throttled the volume of collated nodes from thousands to tens to improve performance. A common use case is to decrease the volume of function calls from a mouse scroll binding, where each pixel scrolled yields a function call.

We wrote flexible serializers for non-serializable JavaScript types such as Events, DOM Elements, and Abstract types like Object and Array for display in Isopleth's source frame views as inputs and outputs. We also used a "beautify" process to unwind minified JavaScript and format it into readable code.[4]

5.5.3.1. Technical Limitations. By extending Fondue [**53**] and the Wisat architecture [**40**] for instrumenting source code on public websites, Isopleth and its Serialized Deanonymization technique inherits the limitations of these approaches. For instance, the system only tracks source activity from top level website frames; scripts loaded dynamically during a UI interaction will not be instrumented, and functions invoked from string via `eval` are not traced. Other browser rendering techniques such as Canvas, OpenGL,

[4]Github: UglifyJS beautifier `https://github.com/mishoo/UglifyJS2`

and Flash are not captured. However, JavaScript calls to these API's are captured and are surfaced to support sensemaking.

Isopleth's Serialized Deanonymization technique does not capture the path of functions passed via closured variable reference, string key reference, global object reference, or DOM element invocation reference (e.g. `onclick=‘‘MyFunction();’’`). Function invocations and asynchronous declaration context are still traced, but Isopleth's purple lines will not draw connections for functions passed this way. One way to overcome this limitation is to additionally track the state of variables over time using Fondue's instrumentation technique, which is currently not supported.

With our current implementation, Isopleth generally performs well on less complex web interactions (fewer than 5K function calls) with minimal interruptions to framerate or website usability. Performance starts to degrade from 5-15K function calls, as each function's arguments and return values are being stored in memory. Beyond 15K function calls, we added filters in Isopleth's runtime source to debounce redundant events and function calls (e.g. older library techniques of querying the URL hash every 200ms to detect hash change). Persisting these invocations to a database instead of memory could alleviate some of the bottleneck with memory pressure.

Many websites are using minification techniques when publishing their code, which can detract from Isopleth's usability because names often help clarify the role of properties, objects, and functions during sensemaking. While there is currently no standard for minification, we observed three primary categories of minification in the sites we studied:

(1) Minification renames JavaScript variables and functions, but object attribute names, HTML property names, and CSS class names are still available. JavaScript API methods such as `document.queryselector` are unmodified.

(2) Minification extends category 1 by "mangling" properties in objects, meaning that methods and shared properties defined in objects become obfuscated. HTML properties, CSS class properties, and JavaScript API methods are still unmodified.

(3) Minification extends category 2 by also replacing HTML properties and CSS class names, which obscures DOM queries. JavaScript API methods are aliased and references to those methods are replaced with the alias. Only string constants in templates and static content are still readable.

Given what is preserved through the minification process, we expect Isopleth to perform reasonably well with category 1 and category 2 minification but not with category 3 minification. Most of the websites in our studies fall into category 1, except Histography and Stripe in category 2. None of our evaluated websites were category 3, which we found were mainly used by larger product organizations such as Google and Facebook that have the resources to build in-house minification engines for obfuscating property names across HTML and CSS. To help make sense of heavily minified code, future work on Isopleth may integrate tools such as JSNice [**73**], which uses machine learning to rename variables based on their usage and context.

Isopleth traces all interactions between JavaScript and HTML/CSS, such as manipulating the DOM, adding classes, and manipulating CSS property values (e.g. `translate3d`). However, Isopleth does not support learners understanding concepts in HTML or CSS,

which, given modern advances to these languages, can themselves be used to create working UI interactions. Our recent work on Ply [**55**] addresses this limitation by providing a visual web inspector that supports novices learning professional web page features in CSS.

5.6. Case Study

To better understand Isopleth's capabilities, we conducted a case study to illustrate how Isopleth can be used to surface programming patterns and implementation techniques on complex professional websites. We selected 12 websites from a diversity of industries based on Alexa popularity rankings, the Webby awards, and personal interest; see Figure 5.7. As this case study is conducted by an expert JavaScript developer (one of the authors), our goal is to assess whether Isopleth provides the desired sensemaking scaffolds to surface a range of concepts and implementation approaches across diverse, complex websites, and not how novices may then use these scaffolds to build new understanding (which we will address via a user study with junior and senior developers that follows). In other words, we sought to first provide an understanding of the breadth of examples that Isopleth can potentially help a learner explore through its core characteristics, before studying how novices can learn new concepts when they use Isopleth.

As a reminder, Isopleth's core characteristics are:

- *Characteristic 1*: Expose hidden functional and event-driven relationships between code components (Condensed Call Graph and Source Frames)
- *Characteristic 2*: Provide visual organizers that allow learners to view slices of code that are functionally related (Facets)

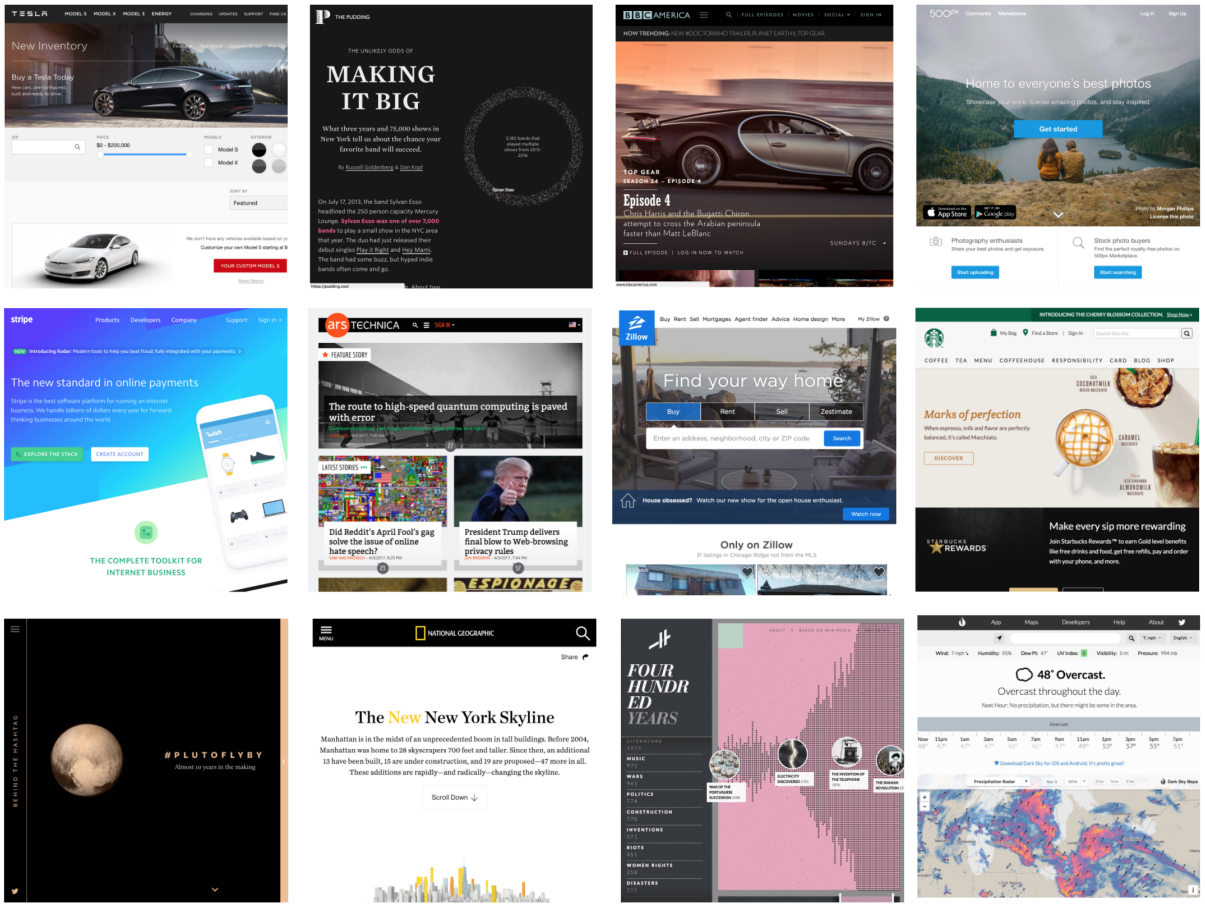

Figure 5.7. We studied Isopleth's ability to support sensemaking and elicit design patterns across 12 websites selected from a diversity of industries based on Alexa popularity rankings, the Webby awards, and personal interest. From top left to bottom right: Tesla, The Pudding's "Making it Big", BBC America, 500px, Stripe, ArsTechnica, Zillow, Starbucks, HashTagsUnplugged's "#PlutoFlyBy" article, National Geographic's "*New* New York Skyline" article, Histography.io, and DarkSky.net.

- *Characteristic 3*: Support iterative manipulation of code representations to reflect a learner's understanding as it develops (Moving Nodes, Node Labels, Code Comments, and Custom Facets)

This case study aims to address the following research questions:

- **RQ1** How do Isopleth's core characteristics support the process of making sense of complex code artifacts?
- **RQ2** What programming patterns/concepts can be surfaced (by an expert) through Isopleth across professional examples sharing similar and different features?

In order to address these two research questions, we used Isopleth to understand implementation techniques across three areas of web development: event bindings, web application design, and dynamic interactive features. While these areas represent different classes of problems in frontend web development, they all require composing an understanding of the relationships among functions and components that together realize a feature and may thus be more easily examined using Isopleth's core characteristics.

5.6.1. Result 1: Support Sensemaking of Complex Web Applications with Isopleth

We discuss in this section how Isopleth supported understanding techniques in event-bindings, web application design, and dynamic interactive features in professional web applications across three respective cases.

5.6.1.1. Understanding Event-Bindings. We used Isopleth to understand the event-binding patterns on websites including BBC America, 500px, ArsTechnica, Pluto Fly-By, and Starbucks while tracking the features implementing the sensemaking scaffolds that helped us. As an illustrative example of a common strategy we used, we describe the process by which we used Isopleth's core characteristics to examine BBC's show-picture-on-scroll feature.

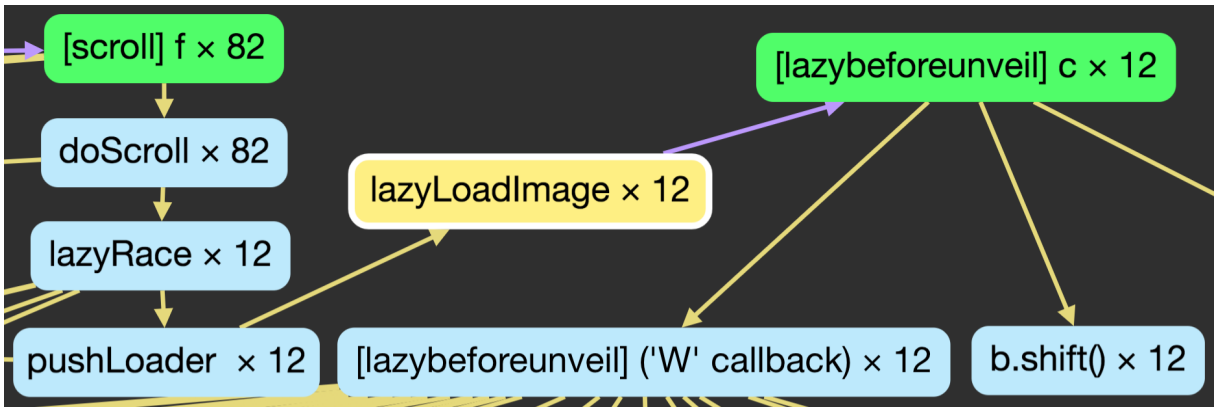

Figure 5.8. Isopleth's condensed call graph representation of BBC America's lazy-image-loading strategy. By reading pre-defined node labels and following functional (yellow lines) and asynchronous links (purple lines), we see that the scroll event (top-left node) was passed to an event handler responsible for the callback (top-right node). While examining source frames, we had renamed the scroll events' child calls to better describe what the functions do, such as implementing the scroll (doScroll), handling a race condition (lazyRace), and pushing an image load request to the browser (pushLoader). The crucial asynchronous link connects the disparate parts of code, which helped us to elicit the design pattern of appending images only when the user scrolls below the fold.

We loaded the initial view of the call graph to find function nodes organized by run time, colored by facet type, and collected by repeat occurrence. Facet labels and coloring on nodes in the call graph provide a visual organizer that helped us to immediately distinguish between setup, mouse, and DOM code and to consider what roles they each may play. From here, we explored DOM-modifying and Mouse facets first, which matched our intuitive understanding of the interaction on the page (i.e. the page changes as the mouse scrolls).

During the exploration, we used Isopleth to expose hidden functional and event-driven relationships between code components; see Figure 5.8. We found a `lazybeforeunveil` node on the rightmost portion of the graph, and worked from this node outward to

discover BBC's asynchronous image-loading strategy. By reading the function labels on its children (linked via yellow lines), we deduced that this node must be responsible for handling an async callback. Curious how the async activity worked together with mouse scrolling, we clicked on the purple line that linked this node to a node in a different call tree. This revealed connected source frames that showed bind-setter and bind-callback functions side-by-side, which helped us to understand how certain scroll points triggered an AJAX call to fetch an image, then inserted it into the DOM. By working backwards from the response to the request, we were able to discover how separate functions in different areas of the source code worked together to achieve this lazy loading strategy.

We also used Isopleth to iteratively label and organize code by our own understanding as we went along. For instance, we discovered that some functions were anonymous, so we manipulated the representation to match our understanding by renaming nodes according to their inputs, outputs, and function bodies as we understood them, like `lazyLoadImage`.

5.6.1.2. Understanding Web Application Design. We used Isopleth to understand the web application design of the Zillow homepage, Starbucks' login, Tesla's car picker, and DarkSky's city finder. As an illustrative example of a common strategy we used, we describe the process by which we used Isopleth's core characteristics to understand the design of the Zillow homepage, specifically how its home search autocomplete feature populates the user's previous home searches into the search bar. Our starting assumption was that after a number of keystrokes, a query would be issued to a server to fetch home results tied to a user's history. However with Isopleth we discovered a more elegant caching pattern.

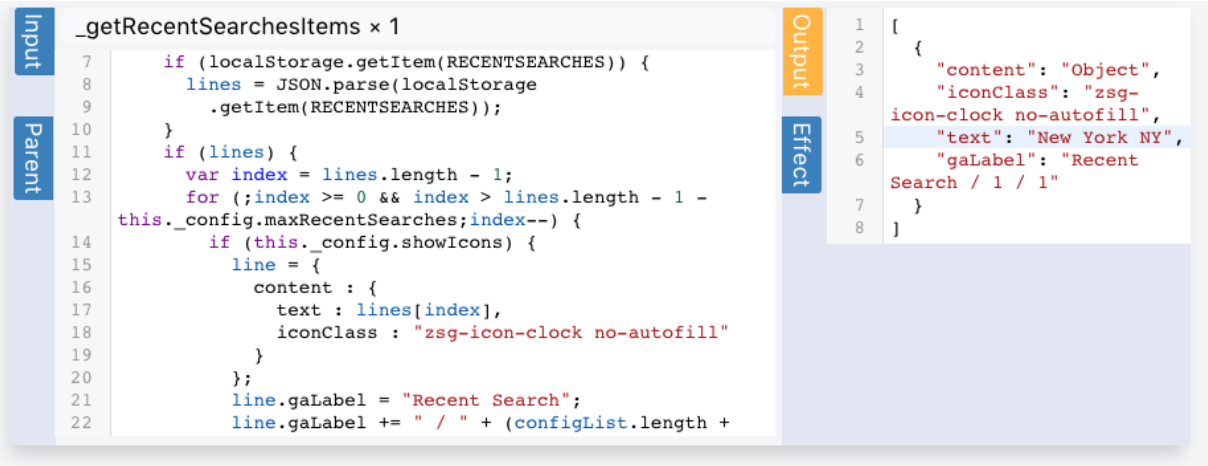

Figure 5.9. A source frame view found while learning about Zillow's recent search results feature in its autocomplete. The construct for loading previous searches is on the left and the captured return value is on the right. We were surprised to find recent searches stored in the browser's local store rather than the user's profile, or synced with the server.

We first used Isopleth's facets to conceptually organize the call graph to highlight AJAX activity and Keyboard activity to see whether links exist between server requests and keyboard events. We imagined that Zillow might save searches in a user's history on the server, but after seeing no AJAX activity linked to keyboard events, we wondered what other implementation approach it may have employed. Using a different view of the same data, we ignored the DOM, AJAX, and Setup facets to focus closely on calls directly related to keyboard events.

We then used Isopleth to expose hidden functional and event-driven relationships between code components. We found calls linked to keyup handlers and used Isopleth's source frame views to reveal underlying properties of arguments and return values for these method calls. In examining these calls, we discovered that typing characters into

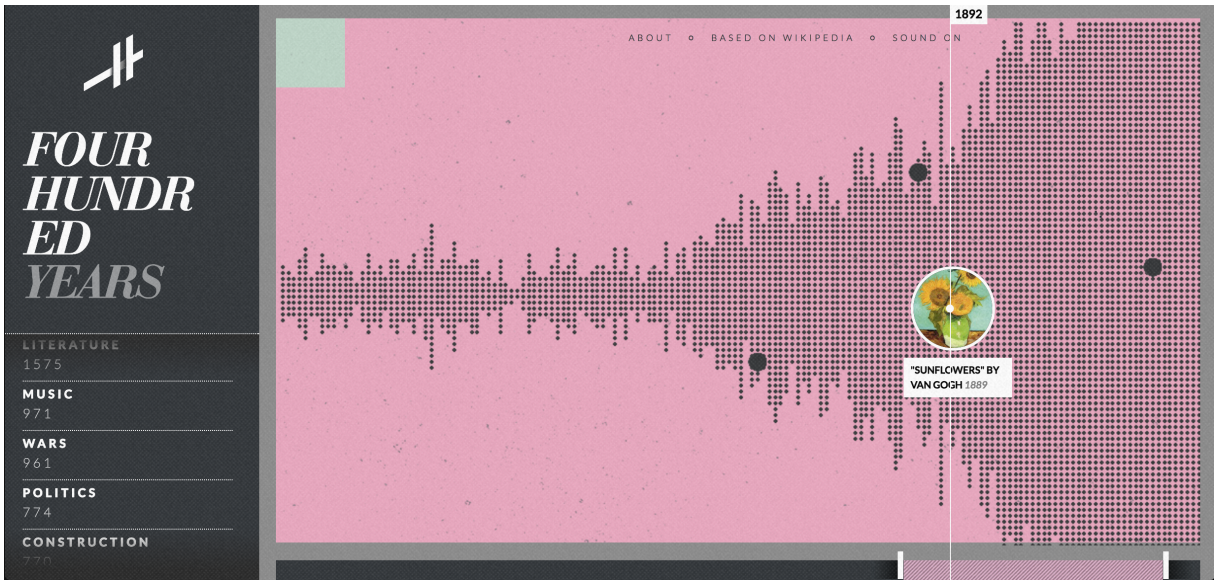

Figure 5.10. The most complex UI we tested was histography.io, which triggers thousands of function invocations in response to mouse movements. On hover, historical events on a timeline bubble up with randomly decaying dots.

the search box queried the user's `localStorage` for recent user search queries, thereby revealing an unexpected lightweight per-user caching strategy (see Figure 5.9).

To help expose the macrostructures in this example, we labeled nodes as we examined their sources and reshaped the call graph into common functional clusters to distinguish nodes responsible for storage, queries, and view updates. We realized that the recent search history feature is encapsulated in a component that is only available if the user's `localStorage` is enabled. Otherwise, it is simply ignored. The ability to reshape the call graph and label methods as we built our understanding helped us to work through large sets of invocations in Zillow's view, storage, and query code, a task which would have otherwise been daunting and tedious.

5.6.1.3. Understanding Dynamic Interactive Features. We used Isopleth to understand dynamic interactive features on Histography.io, the New York SkyLine article, the Making it Big Article, and Stripe's landing page. As an illustrative case, we describe the process by which we used Isopleth's core characteristics to understand Histography.io's highly dynamic UI, particularly its visual effects in response to mouse movement (see Figure 5.10). When moving the mouse, (1) a number of dots denoting historical events follow the cursor; (2) a vertical line indicates the date of events; and (3) a pop-up displays featured historical events. Using Isopleth, we were surprised to find that much of Histography's UI is rendered in WebGL via bindings to a JavaScript library called PixitJS.

Initially receiving a large set of events in the Isopleth call graph, we began by filtering the call graph using the Mouse facet. By quickly skimming through nodes and their sources (seeing "image" or "title" in the source), we were able to distinguish between code responsible for different aspects of the feature. Based on this functional understanding, we rearranged nodes into groups (i.e., "dots," "vertical lines," or "popup") that correspond to these aspects. Examining nodes related to popups, we learned that the popup was rendered in WebGL after fetching a cover image after a `settimeout` relative to the last mouse-mouse event timestamp.

Still unclear on how the vertical lines or dots were drawn, we needed to see some of the underlying logic in the corresponding nodes. Using source frames, we inspected their arguments and return values and found numeric primitives repeatedly used in their clusters of nodes. These numerics were tied to UI positioning, as they were passed to Pixit configuration variables like `position.x` and `position.y`. We created a new facet to detect numeric types in return values, and this enabled us to surface functions that

were operating (or not operating) on numeric data. By studying and labelling each of the smaller number of remaining numeric positioning node, we discovered that the vertical line was rendered in WebGL similar to the popup's positioning hook. The dots following the cursor used a simple random number generator, with values between 0 and 1 to simulate a messy cluster of dots around the cursor. Without Isopleth, it would have been difficult to distinguish and examine the various aspects of code responsible for the feature, and to learn about the interplay between JavaScript and WebGL across components that together realize the feature.

5.6.2. Result 2: Surfacing Design Patterns

Having described how Isopleth's core characteristics can be used to support the process of making sense of complex code artifacts, we discuss in this section the range of features and implementations approaches across websites that we (as expert users) were able to surface using Isopleth. Specifically, we describe how we used Isopleth to find (1) common patterns across similar features, (2) common patterns across different features, and (3) different patterns across common features.

5.6.2.1. Common Patterns across Similar Features. Isopleth's facet filtering, call graph, and source frames helped us discover consistent design patterns across similar features. For example, by examining the relational links among DOM Query and Setup facets in the call graph, we found that Starbucks, ArsTechnica, Zillow, and 500px used the same content swapping technique based on logged-in state. After filtering the call graph using the DOM Query facet, we inspected the arguments in Isopleth's source frame views to find that ArsTechnica, DarkSky, NatGeo, and Stripe add and remove a class "hidden" to

DOM elements to toggle their visibility. We also found 500px and PlutoFlyBy's animated scrolling technique when simply looking into the latest occurring invocation with a Mouse facet.

5.6.2.2. Common Patterns across Different Features. Isopleth's facet filters and source frames also helped us elicit consistent design patterns across websites with different features. Tesla's car picker and BBC's landing page each listen for a UI event that triggers an AJAX call, which loads JSON containing image URL's, which are appended to a template and rendered to the DOM. This lazy-load pattern emerged through an iterative sensemaking process between DOM, AJAX, and Mouse facets for both pages. 500px, MakingItBig, and NatGeo's scroll-based CSS transform animations were surfaced by using Isopleth's Mouse facets, then inspecting arguments and return values in source frame views. We identified loops operating on values modifying CSS `translate3d` positions to achieve a smooth GPU-enabled transition.

5.6.2.3. Different Patterns across Common Features. Isopleth's default and custom facet filters helped us to discover contrasting implementations for the same feature. Different patterns may be equally valid, but often the pattern highlighted the needs of the application domain, such as a socially integrated login on the BBC America site compared to a simple form-post login on Stripe. DarkSky, BBC, and Zillow's autocomplete search techniques were surfaced through Isopleth's Keyboard, AJAX, and DOM facet filters, and each of their implementations fits their domain. DarkSky's autocomplete searches local storage for previous searches and builds a URL query otherwise, fitting the site's simple design. BBC issues AJAX calls and populates templated results, fitting the site's reactive design. Zillow's search populates a result list, but builds a URL redirect to their

map interface, fitting their real estate shopping design. Each website's login technique varied, and while Isopleth helped reveal insights, some sites did not use JavaScript to support user login. 500px, Stripe, Tesla, Starbucks, and ArsTechnica simply redirected login actions without JavaScript. Isopleth revealed BBC's use of the social Janrain platform for an AJAX social login through its DOM and AJAX filters, however on successful AJAX login, BBC oddly refreshes their page. Isopleth's DOM, Keyboard, and AJAX facets along with a customized facet filter for login arguments showed that Zillow uses a refreshless login strategy via secure AJAX post and view update.

5.6.2.4. Surfacing Architectural Decisions. Isopleth's call graph helped us surface unexpected lower-level characteristics of websites such as identifying their JSON API, or revealing large amounts of dormant code from framework-bloat or analytics packages. 8 of the 12 websites have mouse-tracking analytic packages, which we noticed through high call counts in collated superfluous invocations related to mouse events. 3 of the sites use large frameworks including Angular, YUI, and React, with thousands of invocations in Isopleth's unfiltered call graph views during simple UI changes. Isopleth revealed excessive polling activity in un-collating its call graph, where 4 websites contain library code that polls `window.location` every 20ms for hash changes. Finally, by showing library code and filtering for AJAX facets, Isopleth streamlines the ability to surface how applications structure their interaction with a remote API.

5.7. User Study

After demonstrating that Isopleth can surface a wide variety of design patterns used in professional websites through our case study, we evaluated the system's ability to support

learners as they explore and make sense of professional code. We conducted a lab study with ten novice "junior" developers and four more experienced "senior" developers. While we were most interested in determining whether Isopleth effectively supports novices, we were also interested in understanding how learners with different levels of experience interact with the core Isopleth features. We aimed to address three core research questions through this study:

- **RQ1** Are learners' conceptual models of complex professional web features more accurate after exploring the code with Isopleth?
- **RQ2** How do learners use the Isopleth features during the sensemaking process?
- **RQ3** How do the sensemaking strategies used by junior and senior web developers differ?

5.7.1. Methods

Our lab study had a single condition; participants completed a pre-test, completed a sensemaking task with Isopleth, completed a post-test, and responded to questions about the experience of interacting with Isopleth. We describe the study methods in detail below.

Participants. Our participants included ten junior web developers with less than one year of professional web development experience but at least one professional internship, and four senior web developers with more than three years of professional experience. The ten junior web developers (seven male, three female) were undergraduate students at our university recruited through university email and Slack channels. The four senior web developers (all male) worked in industry in a large midwestern city, and were recruited

through referrals. We evaluated the experience on each participant's CV to ensure they met our "junior" and "senior" inclusion criteria. All participants gave informed consent for participation in the study.

Professional Examples. As part of the study, each participant explored the source code for an interactive feature on one of four popular websites: the National Geographic NY Skyline Article, Histography.io, BBC, and XKCD's big map. We selected these websites because we explored them in depth as part of our case study, and we knew that each involves a simple and intuitive interaction with a clever and complex underlying implementation. In the National Geographic New York Skyline article, scrolling horizontally causes a zoom effect on the skyline, and hovering over new buildings yields information about them. On hover in the Histography.io site, little dots follow the cursor indicating historical events and dates change for which year the user is navigating over. When clicking the header on BBC America's landing page, it expands and reveals images not present before. XKCD's big map strings images together to form a draggable map, allowing users to explore an extremely large comic through a normal browser-sized viewport. These examples provided nice opportunities for participants to take advantage of Isopleth's affordances; while each has a seemingly obvious implementation on first look, deeper exploration reveals clever and scalable design patterns that may be counterintuitive to novices.

Procedure. First, we confirmed each participant's degree of comfort with web development concepts by asking a series of basic questions such as "What is one way you can hide a DOM element?" The goal of these questions was to ensure that the level of experience reported on each participants' CV accurately captured their degree of understanding.

Next, participants completed a 10-minute tutorial during which a researcher taught them how to use the Isopleth interface. After learning about the interface, participants were asked to explore and interact with a toy example to demonstrate advanced features in Isopleth such as finding asynchronous bindings or creating custom facet filters.

Next, the participants selected one of the four professional websites to explore during the study. Participants were told which interactive feature to study, and were asked to play with the UI for that feature until they understood its functionality. Participants spent around five minutes interacting with the feature on average. After interacting with the website, we asked each participant to spend five minutes drawing a diagram of how they thought the feature was implemented, from responding to a user interaction through creating a visual effect on the webpage. This diagram served to externalize the participants' conceptual model of the feature functionality, and was used as a pre-test that reflected their understanding prior to interacting with Isopleth.

After this pre-test, participants were asked to use the Isopleth interface to explore the source code for this feature. We told participants their goal was to accurately describe how the feature was implemented. We gave each participant 25 minutes to complete the task, and told them they could stop whenever they were confident that they understood how the feature was implemented. During iterative user tests we found that most people completed this task in under 20 minutes, indicating that participants would have enough time to explore sufficiently. Participants were free to take notes on paper or in a text editor during the exercise and ask clarifying questions about how to use Isopleth.

Upon completing the sensemaking task, participants were given ten minutes to describe their new understanding. We first asked users to write pseudocode to describe

program flow from the start to the end of the interaction. Then, participants drew a diagram of their conceptual model of the feature implementation, which served as a post-test. During the drawing task, participants were shown their original diagram (pre-test) and were allowed to either draw a completely new diagram or modify and extend their existing diagram. After drawing the diagram, participants were asked to verbally describe (1) any differences between their prior and current understanding of the feature, (2) which Isopleth features were most helpful during their sensemaking process, (3) any programming concepts or design patterns they discovered that they did not know about previously, and (4) any features or functionality they wished Isopleth included.

Measures. To measure changes in participants' understanding of the interactive features after using Isopleth, we scored the accuracy of their pre-test and post-test diagrams, and then compared the accuracy scores for each participant. These diagrams externalized participants' conceptual models of the feature implementation at that point in time. To measure the accuracy of their conceptual models, one of the authors created *ground-truth diagrams* that represented the true implementation of each of the four web features. The author has more than five years of professional web development experience and produced the ground truth diagrams through a deep review of the source code, during which he relied on both Isopleth and his own conceptual knowledge of JavaScript. This author then evaluated participants' pre- and post-test diagrams by comparing them to the ground truth diagrams and counting the number of the correct components, relationships, and data flow elements that were present. This produced a score (percentage correct) for each pre-test and post-test diagram. To measure changes in accuracy over time, we calculated the difference in scores between each participant's pre- and post-test diagrams.

We made a distinction between participants who rejected, changed, or accepted their original model of the feature functionality as presented in their pre-test diagram. We considered participants to have *rejected* their original model as incorrect or invalid if they drew a completely new diagram or significantly altered more than half of their original diagram. Those who expanded their original diagram or changed less than half of the content *changed* their model. Finally, we considered participants to have *accepted* their original model after verifying that it was correct as those who added details but did not substantially change the content in their diagram.

To measure participants' usage of Isopleth features, we captured screen recordings during the study and logged all clicks during their interactions with the Isopleth interface. We used the log and video data to count the number of times each participant used each Isopleth feature during the sensemaking task. To capture participants' responses to the initial questions about web development and the final questions about the experience of interacting with Isopleth, we audio-recorded each session in full and transcribed all spoken text for analysis.

We analyzed our user study data to evaluate our three core research questions, first measuring changes in the accuracy of participants' conceptual models before and after using Isopleth, then exploring how participants used the Isopleth features to support their sensemaking process, and finally identifying differences in the behavior of junior and senior web developers. We present the results of each of these analyses in the sections below.

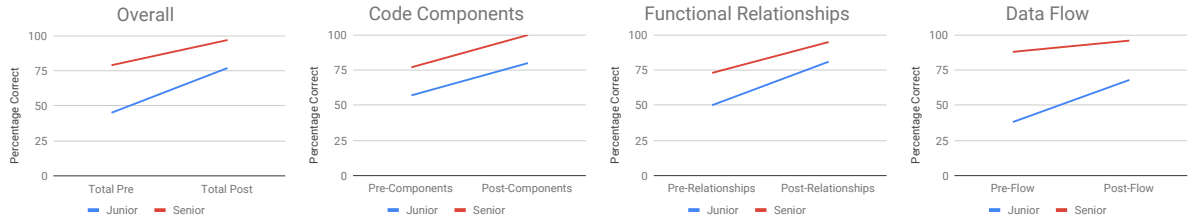

Figure 5.11. Graphs show the change in the accuracy of junior and senior developers' conceptual models before and after using Isopleth. The *Overall* graph shows the average change in accuracy, while the *Code Components*, *Functional Relationships*, and *Data Flow* graphs show the changes in accuracy for elements relating to those specific concepts. Overall, junior developers increased the accuracy of their mental models by 31%, and senior developers reached a near-perfect 97% accuracy.

5.7.2. Result 1: The accuracy of developers' conceptual models improved after using Isopleth

All of our participants improved their conceptual understanding of the feature implementations through interacting with Isopleth. The accuracy of junior developers' conceptual models improved by 31% between the pre- and post-test, and the accuracy of senior developers' models improved by 17%. A repeated measures ANOVA shows that the difference between pre- and post-test scores across all of our participants is statistically significant ($F(1, 13) = 4.72$, $p < 0.0001$) despite our small sample size. As shown in Figure 5.11, participants more accurately described code components, functional relationships, and data flow in their post-test diagrams. Unsurprisingly, senior developers performed better than junior developers on both the pre- and post-test. When drawing their post-test diagrams, nine out of ten junior developers either *rejected* or *changed* their original model, while all senior developers *accepted* their model.

Most junior developers' conceptual models changed substantially after interacting with Isopleth; of the ten junior developers, four rejected their pre-test model, five changed their model, and one accepted their model. First, this indicates that the junior developers were not able to construct accurate conceptual models of the functionality during the pre-test. As expected, spending a brief period of time interacting with the web feature was not sufficient to help these developers understand how it might be implemented. More importantly, we found that the accuracy of the junior developers' models improved substantially between the pre- and post-test; their post-test models were 31% more accurate on average. In addition to looking at overall accuracy, we separated out each diagram's accuracy according to the number of components, relationships, and data flow attributes each participant successfully identified. As shown in Figure 5.11, we saw large improvements across all three categories, demonstrating that Isopleth helped participants understand not only the components involved in the implementation of a feature, but also the relationships between components and the data flow attributes that help them coordinate. These substantial gains in conceptual understanding suggest that professional examples can provide an avenue for novice developers to learn authentic implementation practices.

The senior developers performed substantially better than the junior developers on the pre-test, and all four accepted their original models of the feature implementation. This indicates that the senior developers had sufficient background knowledge to predict how a feature might be implemented from the pre-test activity, highlighting the differences between junior and senior developers. However, Isopleth still helped senior developers improve the accuracy of their diagrams. Their post-test diagrams were 17% more accurate than their pre-test diagrams, reaching an impressive final accuracy of 97% on average.

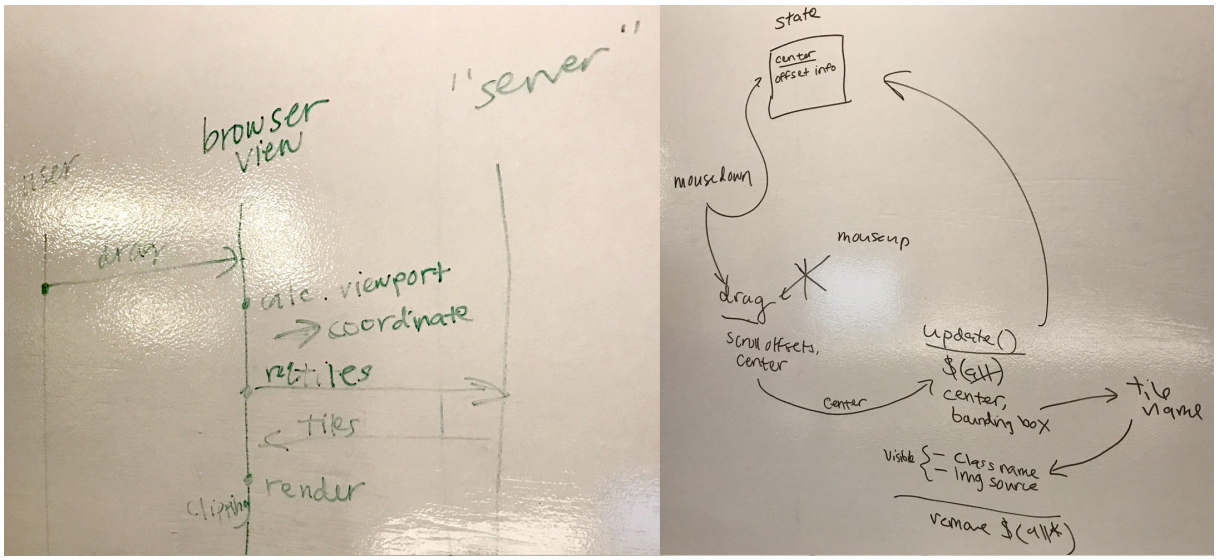

Figure 5.12. A junior developer diagrams their conceptual model before (left) and after (right) using Isopleth. The developer rejected their conceptual model in favor of a new model formed using Isopleth. Before using Isopleth the developer thought that XKCD tracks a drag, calculates the viewport change, re-tiles the images, and renders with clipping. After using Isopleth the developer found that drag coordinates are transformed into center-offsets which are used to load and unload map tiles dynamically based on filename.

Again, the improvements were distributed across components, relationships, and data flow attributes, as shown in Figure 5.11. This demonstrates that even though senior developers had a good initial understanding of the feature implementations, Isopleth was able to help them fill in the details needed to build a fully accurate conceptual model.

5.7.2.1. Rejected Model. Four junior developers rejected their original conceptual models after using Isopleth. In general, these developers presented vague and ambiguous conceptual models during the pre-test, but overcame misconceptions and provided much more detail in the post-test. As an example, consider the pre- and post- diagrams

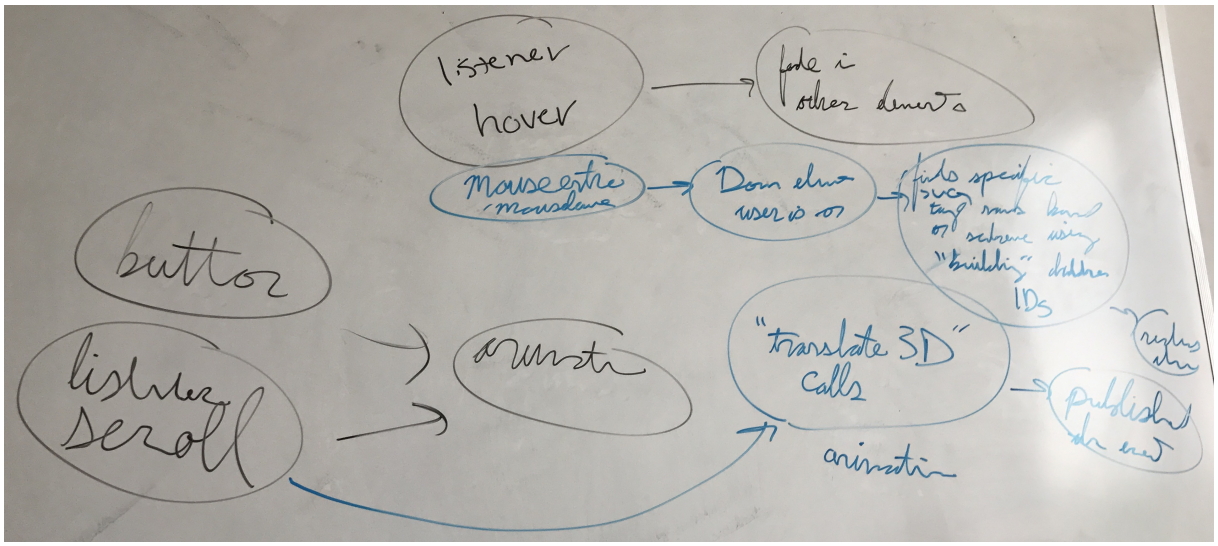

Figure 5.13. A junior developer diagrams their conceptual models before (black) and after (blue) using Isopleth. The developer changed their conceptual model by exchanging some components and relational attributes with more accurate representations. Before Isopleth the developer thought that the New York Skyline website listened to hover events to trigger an animation to show a building. After using Isopleth the developer found that the website listened to `mouseenter` and `mouseleave` instead of hover. They also discovered that instead of an animation, there was a DOM visibility attribute that was toggled based on querying a building's ID.

one junior developer drew for the XKCD click-and-drag map, shown in Figure 5.12. Before using Isopleth, this participant thought that the site tracks drag events, calculates the viewport change, re-tiles the images, and renders with clipping. Through Isopleth, the participant discovered an elegant technique where coordinates are transformed into center-offsets, which are then used to load and unload map tiles dynamically based on filename. This more detailed and accurate representation of the website functionality is clearly visible in the post-test diagram.

5.7.2.2. Changed Model. Five junior developers changed their original conceptual models after using Isopleth. In general, these developers described their hypotheses about

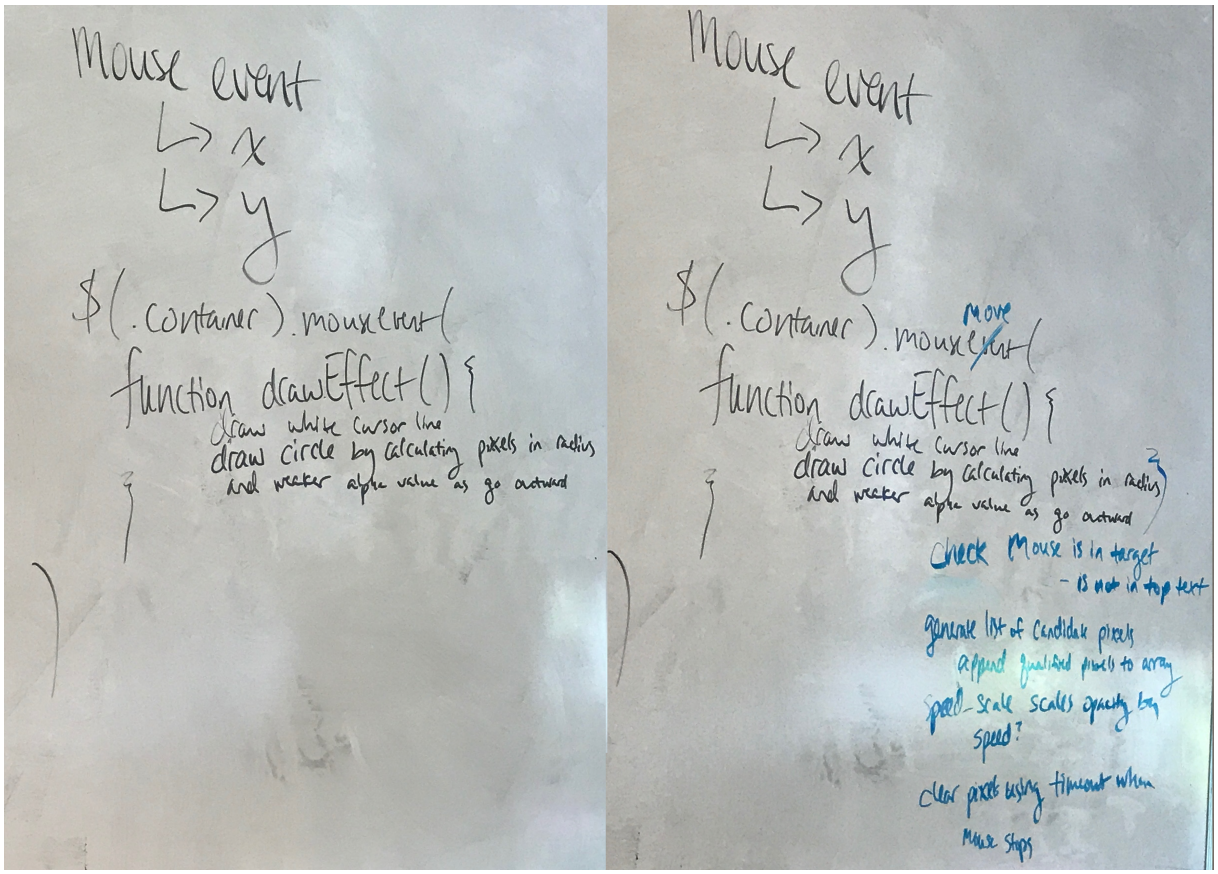

Figure 5.14. A senior developer diagrams their conceptual models before (left) and after (right) using Isopleth. The developer validated their jQuery-style pseudocode model and added specific details about Histography.io's cursor movement found in Isopleth's source frame views. Prior to Isopleth they described a DOM query and binding on `mouseevent` of the DOM item which triggers a function `drawEffect`; this function would then call functions to render a circle and pixels. With Isopleth, they discovered actual bindings to `mousemove` that were close to their pseudocode along with greater detail including validating the range of mouse movement, tying mouse speed to scaling, and cleaning up pixels when complete. The developer validated their key components, relationships, and data flow and added clever implementation details to their prior model.

the possible general architecture for the feature during the pre-test, and then used Isopleth to discover the details of how the website implemented this high-level approach. Consider

the example diagram shown in Figure 5.13, where the original pre-test diagram is shown in black ink and the post-test additions are shown in blue ink. During the pre-test, this participant thought that the New York Skyline website listened to hover events to trigger an animation to show a building. After using Isopleth the participant found that the website listened to `mouseenter` and `mouseleave` instead of hover. They also discovered that instead of an animation, there was a DOM visibility attribute that was toggled based on querying a building's ID. This junior developer had a basic understanding of the high-level pattern used to implement this feature (listen for an event, trigger a change in the UI), but was able to improve their conceptual model by exchanging some components and relational attributes with more accurate ones after using Isopleth.

5.7.2.3. Accepted Model. The five remaining developers (four senior, one junior) accepted their original conceptual models after using Isopleth. These participants all provided accurate depictions of the logical components, how they coordinated, and how the data flow functioned for their chosen website. As an example, consider the pre- and post-test diagrams one senior developer drew for Histography.io, shown in Figure 5.14. This participant used Isopleth to validate their jQuery-style pseudocode model and added specific details about Histography.io's cursor movement that they found in Isopleth's source frame views. Prior to Isopleth they described a DOM query and binding on `mouseevent` of the DOM item which triggers a function `drawEffect`; this function would then call functions to render a circle and pixels. With Isopleth, they discovered actual bindings to `mousemove` that were close to their pseudocode along with greater detail including validating the range of mouse movement, tying mouse speed to scaling, and cleaning up pixels when complete. Through using Isopleth, this participant was able to validate their

key components, relationships, and data flow and added implementation details that were missing from their original model.

5.7.3. Result 2: The developers used Isopleth's features to support their sensemaking processes

After seeing that interacting with Isopleth improved developers' conceptual models, we were interested in learning how the developers used Isopleth features to support their sensemaking process. In particular, we wanted to discover whether Isopleth's core characteristics supported participants in the ways we expected. In the following sections, we describe how each Isopleth feature was used; Figure 5.15 summarizes feature usage by junior and senior developers.

5.7.3.1. Condensed Call Graph and Source Frames. The condensed call graph and source frames were designed to expose hidden functional and event-driven relationships between code components to help learners understand how the components coordinate to implement a feature of interest. We also provide affordances that allow users to iteratively manipulate these representations, with the goal of helping them articulate their understanding as it develops. As shown in Figure 5.15, all fourteen participants interacted with both the call graph and the source frames, and nine changed node labels and edited source code during their sensemaking process.

All fourteen participants began their sensemaking process by exploring the condensed call graph and looking for nodes of interest; as one participant put it: *"in general I just looked around [the call graph] for functions or classes that looked familiar and dove in from there."* Many users started by looking at the most recent invocations on the far

Feature	Junior	Senior	Total
Call Graph	10	4	14
Node Click & Drag	10	4	14
Click Async Lines	4	3	7
Click SD Lines	4	3	7
Node Label Change	8	1	9
Begin on Right	7	1	8
Source Frame	7	3	10
Edit Source Code	7	2	9
Inputs Button	5	3	8
Outputs Button	1	3	4
Parent Call Button	2	3	5
Binding Fn Button	0	1	1
Fn Calls Button	4	0	4
Async Delegate Button	0	0	0
Facets	4	2	6
Mouse	3	2	5
Keyboard	0	2	2
Setup	0	2	2
AJAX	0	1	1
DOM	1	1	2
Custom Input Facet	3	2	5
Custom Output Facet	2	2	4

Figure 5.15. The number of junior, senior, and total developers that used different Isopleth features to make sense of the provided professional code

right of the graph; as one noted *"I really liked that it is timeline-based so I can retrace what happened and know some locations to start looking."* Upon identifying potentially relevant nodes, all users clicked on the nodes to study the associated code in the source frame view. A junior developer noted that *"you can see the parent, inputs, and calls"*; eight developers clicked on the input, output, and call buttons as a convenience to quickly find connected nodes. We observed that all developers moved back and forth between scanning the call graph for relevant nodes and reading code in the source frames during sensemaking, exploring how nodes connect and then diving into their code to better understand their roles. These usage patterns show how exposing disciplinary information about the relationships between code components helped our participants explore and make sense of complex professional code.

The majority of our participants manipulated the call graph and source frame representations during sensemaking. Nine developers (seven junior, two senior) edited the code displayed in the source frames by adding comments or renaming variables, and nine developers (eight junior, one senior) edited node labels. The two senior developers who made edits to source frames and labels only made a few updates, spending most of their time skimming code and verifying prior assumptions. Two of the junior developers also made only a small number of edits; these developers had lower pre-test scores (average 23%) and spend most of their time working to understand the code they were reading in the source frames. The remaining five junior developers with higher pre-test scores (average 53%) regularly manipulated the representations throughout their explorations, adding comments and renaming variables. Some of these participants also updated node labels as a way of marking completion (e.g. with the label "done") when they had finished

reading and understanding the node source. After updating a node label, one developer said *"now that one's done, I just have three others I need to look at before I know what's going on."* This demonstrates that these junior developers were using labels to represent their current understanding, the central aim of our manipulatable representations.

Most of the participants who manipulated the representations (seven junior) not only added labels and source comments, but also referenced these annotations at a later point during their sensemaking process, showing that these labels and comments served to document and communicate their current understanding of the code. These users all followed an iterative sensemaking process, but adopted a variety of different strategies. We observed (1) top-down iteration, referencing each parent call then jumping back to the top after reaching the bottom, (2) bottom-up iteration, visiting each child node then reviewing and comprising the parent until reaching the top, and (3) skim-and-save iteration, navigating connected nodes and only editing nodes of interest. There was no clear evidence of a single strategy being more effective over the other, and some users mixed strategies, for example starting with strategy (3) and then focusing in on a specific logical branch with strategy (1).

Overall, these usage patterns demonstrate that Isopleths' sensemaking scaffolds helped our participants build their conceptual understanding of the feature implementations. The developers used the call graph and source frames to explore relationships between code components, and many manipulated these representations to externalize their understanding as it developed.

5.7.3.2. Facets: Colors, Labels, and Filters. Facets were designed to provide learners with a way to view functionally related slices of code. Facets are conveyed both

through the coloring and labeling of nodes in the call graph and the facet filter buttons that allow users to view the subset of nodes related to a given facet. We also provided affordances for users to create custom facets to define and explore functional slices of interest, with the goal of helping users manipulate the provided representations to support their understanding. As shown in Figure 5.15, all of the participants interacted with the colored nodes in the call graph. In addition, six developers used the facet filter buttons during their explorations, and five explored building their own custom facets.

The participants who used the facet filter buttons fell into two groups: those who used the filters with intention to explore a hypothesis, and those who used the filters to gain entry points into the code. Three developers (one junior, two senior) clicked on facet filter buttons with a stated goal in mind, for example clicking the AJAX filter to show only nodes which exist in callstacks that either issued or responded to an async HTTP request. These users made thoughtful choices about when to apply a filter to view a different slice of functionality in the call graph. Furthermore, these were the only three users to successfully define and use a custom facet filter; the two other junior developers who tried to define facets gave up quickly. The custom facets these three developers created included filtering for return values in a range, filtering variable types from the arguments, and filtering for attributes in JSON objects. These filters allowed the users to significantly reduce the call graph complexity and hone in on nodes of interest. These three users had high pre-test scores (average 74%), indicating that they had the background knowledge required to take advantage of the predefined and custom facet filters to explore hypotheses.

The three other developers (all junior) clicked on facet filter buttons without a clear hypothesis related to the filter. Instead, these developers used the filters to gain entry-points into the code; one commented *"at the beginning, there were a lot of nodes, then I applied the filters and with the coloring it got a lot easier."* These users appeared to follow the behavior that was modeled during the Isopleth study at the beginning of the demo, clicking on DOM and Mouse filters to see associated code. For example, while exploring the BBC header open-close effect, one user clicked the DOM facet filter stating an abstract goal: *"I want to figure out the place where [the header] opens".* The user explored the filtered call graph for a while, then clicked the Mouse facet filter, and found a node that showed the source code `querySelector(‘.header-secondary’)`, and exclaimed *"this looks promising!"* In contrast to the developers who used filters to explore a specific facet-related hypothesis, these users used facet filters to reduce the complexity of the call graph and help them explore the source code. These three users had lower pre-test scores (average 34%), and most likely lacked the background knowledge needed to form clear hypotheses at the beginning of their sensemaking process. However, the facet filters were still able to support their sensemaking process by providing intuitive entry points into the code.

The remaining eight developers (six junior, two senior) did not use the facet filter buttons during their exploration. However, we observed that these users referred to the facet labels and coloring on nodes in the condensed call graph during their sensemaking process. For example, we saw users move their mouse from node to node in the call graph, reading the facet labels without clicking. We also saw some users scan the call graph for colors of interest (e.g. green for top-level, purple for DOM, etc.) before exploring a section

of the call tree. During exploration, one junior developer said *"I'm looking for a node, probably purple [DOM], because there will be a click then a DOM something."* In the interview at the end of the study, one junior developer said *"the colors in the [call] graph are very useful, because JavaScript is a mess and helps to see what returns from where and what gets passed to where."* Given that developers were able to see functional slices of the code through the node labels and colors, the eight developers who did not use facet filter buttons may have found all the information needed to support their sensemaking in the call graph itself.

5.7.4. Result 3: The developers used a variety of different strategies during sensemaking

In addition to understanding how participants used Isopleth's features to improve their conceptual models of interactive web features, we were also interested in studying whether junior and senior developers used different strategies during sensemaking. We found that they do adopt distinct program comprehension strategies. Senior developers were more likely to approach the inspection task with concrete hypotheses about how the feature was implemented. We therefore observed them scanning the call graph to find particular components and using custom facet filters to hone in on particular functionality. In contrast, junior developers were more likely to approach the task with vague ideas about the implementation. We observed them tracing backwards from the most recent invocations to the right of the call graph, and working through nodes systematically as they built up their conceptual models.

However, beyond these general differences between junior and senior developers, we also found that the participants used a wide range of sensemaking strategies. As described in the previous section, not every developer used every feature, and feature usage was often dependent on the developers' incoming knowledge and approach to the exploration task. Some extensively manipulated the call graph and source frame representations to express their current understanding, while others never made changes. Some used facet filters to view slices of the call graph, while others explored the whole graph and used facet colors and labels to help them dig through the complexity. These patterns demonstrate that Isopleth's core features can support a variety of program comprehension strategies.

In this section, we present user stories that reflect the strategies used by three distinct participants in our study: one who rejected their original conceptual model, one who changed their model, and one who accepted their model. All participant names have been changed. These stories demonstrate the variety of program comprehension strategies we observed, and show how Isopleth's core features were used to support different sensemaking approaches. Importantly, Isopleth's features were able to effectively support these diverse approaches, showing that they are flexible and do not require users to follow a particular sensemaking procedure. The central goal of Isopleth is to help novice learners bridge from their own intuitive understanding; as a result we see this flexibility in supporting distinct program comprehension approaches to be a significant strength of the system.

5.7.4.1. Grace explores BBC. A junior participant Grace had pre-test score of 28%, and rejected her original conceptual model during the post-test. She activated Isopleth on the BBC website to explore its header and image loading functionality, and began her

sensemaking process by looking at the call graph. Grace moved the call graph around looking for familiar syntax or potential starting points. Without a strong intuition to start her search, Grace switched back to the website and recreated the effect to see which call counts updated on the graph. Returning to the call graph, Grace was still unsure where to look, but clicked on the DOM facet filter after remembering that was helpful in narrowing down the search space during the pre-task demo. While clicking on a few top level nodes, Grace said *"I'm trying to figure out which one starts the chain of stuff."* Grace found top-level DOM nodes that referenced `nav-browse-shows` and `nav-expanded`, and labeled a node *"once the hamburger is open"* to indicate that the function was a callback for when the drawer view expanded. Interested in seeing how her mouse movement related to the drawer open effect, Grace remembered the Mouse facet filter from the demo and clicked it. With the two filters applied, Grace clicked through three nodes to the left of her first labeled node, as these nodes occurred before the "hamburger is open." Grace worked through each node by updating its label and following its direct calls, and she discovered how mouse events that were bound at setup triggered a draw-open and image-load-in-carousel effect. Grace rejected her prior model, which outlined a click, resize, and insert-pics functionality, and instead created a new model that included lazy-loading of images, changing CSS classes to trigger opens and closes, and even the analytics that were transmitted when opening and closing the header.

5.7.4.2. Ada explores National Geographic's New York Skyline Article. A junior participant Ada had a pre-test score of 44%, and changed her original conceptual model during the post-test. She activated Isopleth on the New York Skyline article and began her sensemaking process on the right side of the call graph (see her pre-post model

Figure 5.13). Seeing light green top-level nodes, Ada searched for one to begin with, and selected the node that was furthest to the right of the graph (the most recent call). Ada was unsure what the source code was doing, and visited neighboring nodes by following async call lines and proximity in the call graph. Ada found a node pre-labeled as "scroll.waypoints handler" and began an iterative manipulation process. She worked through several of its child nodes by clicking to open their source frames, activating the call, argument, and return value buttons, and writing notes in the comments (e.g. *"variable a is the building, loop, now b is the building, move page"*). Ada renamed each of the child nodes with short descriptions such as *"remove click event''*, *"listen for hover"*, *"add to horizontal position''*, and repeated this process up the tree renaming the top level node, *"listen to mouse, move the skyline."* Through bottom-up iteration, Ada learned how the skyline moved and how it showed details on hover. During her post-test, Ada changed her prior model, which featured button callbacks and simple listeners, to include more components such as scroll handlers, relationships such as hover to `translate3d`, and data flow showing which building to animate.

5.7.4.3. Alan explores XKCD. A junior participant Alan had a pre-test score of 63%, and accepted his original conceptual model during the post-test. He activated Isopleth on XKCD to understand its click-and-drag effect. Alan saw the node colors, async relationships, and relationships among top level callers. He read the pre-populated node labels and gained an understanding of the role each node played in the context of the larger picture. Alan selected a node carefully based on its label and relationships, and saw a node that looked like a callback bound at page load which modified the DOM and had a mouse facet label as well. Alan started by reading the source frame of a "mousedown" node; he

followed lines to "drag" and "update" nodes and continued to move through the graph. Given the roles of these nodes, Alan visited neighboring nodes to verify the context of the original nodes. Alan read the code in the source frames carefully, making assumptions about unknown methods based on how they are used in context and mentally bookmarking spots he was confused about. Alan repeated this process until he gained a verified working knowledge of the effect. In his post-test Alan validated his prior model, adding specific details about the scope of mouse bindings and the clever image-index naming convention for lazy-loading images.

5.8. Discussion

Our work advances the creation of Readily Available Learning Experiences (RALE) that transform professional web applications into opportunities for authentic learning. As a first step towards realizing RALE, this paper contributes *Isopleth*, a web-based platform that helps learners interactively explore the complex front-end code of professional web applications. Isopleth embeds sensemaking scaffolds informed by the learning sciences and the program comprehension literature to help learners build conceptual models of how components coordinate to produce complex interactions in professional code. Through a case study, we found that Isopleth provided helpful scaffolds for making sense of event bindings, web application design, and complex interactive features across a wide range of professional websites, and can be used to surface the various ways that the same features can be implemented across professional web applications. Through a user study with junior and senior developers, we found that Isopleth helped junior developers significantly improve their conceptual models between a pre- and post-test (by 31%); they improved

their understanding of individual code components, functional relationships among components, and data flow. Moreover, Isopleth helped senior developers reach a near-perfect 97% accuracy on the post-test. These results provide promising early evidence of how tools like Isopleth can enable learners to use professional examples to build deeper conceptual understanding.

Beyond demonstrating conceptual learning gains, our study results confirmed our design arguments by validating how Isopleth's core characteristics can be used to support the process of making sense of complex code artifacts. First, Isopleth exposes hidden functional and event-driven relationships between code components with its condensed call graph and source frames. We argued that these affordances make disciplinary knowledge and practice accessible to learners by surfacing all connections among code components in the call graph (including asynchronous ones) and encourage the expert practice of thinking about functions in terms of inputs and outputs through source frames. Through our studies, we found both ourselves and other developers following asynchronous links in the call graph to discover how separate functions in different areas of the source code worked together; using source frames to examine the arguments and return values of method calls; and switching between exploring the structure of the call graph and investigating relationships in detail through source frames throughout the sensemaking process. These usage patterns demonstrate how making the semantics of the discipline apparent can help learners compose a deeper and more complete understanding of how components coordinate to implement a feature.

Second, Isopleth provides visual organizers that allow learners to view slices of code that are functionally related through its facet filters and facet labels. We argued that

these affordances help learners use their intuitive understanding of how interactions cause visual effects (e.g., mouse and keyboard events) and organize code into functional slices that reflect how experts think about functionality (e.g., where events are bound and where AJAX calls are made). In our studies, facet filters helped us and three developers with high pre-test scores to explore concrete hypotheses by significantly filtering down on the call graph to hone in functional slices of interest. Facet filters also helped three junior developers who did not have a concrete hypothesis gain entry points into portions of the code to start their exploration based on their intuitions. The remaining developers who did not use the facet filters nevertheless referred to facet labels and coloring, e.g., to locate nodes of interest based on their functional roles during their sensemaking process. These observations illustrate how visual organizers can help learners connect their prior knowledge (however deep or shallow) to performing the sensemaking task at hand.

Finally, Isopleth supports iterative manipulation of code representations to reflect a learner's understanding as it develops through affordances for moving nodes, editing node labels, adding code comments, and creating custom facets. We argued that these affordances can help learners to externalize their mental models of code structures and functionality in ways that further support their sensemaking process. In our studies, we found ourselves and many other developers constantly dragging nodes to group them according to their functional roles, adding code comments to capture our understanding, and relabeling nodes with more intuitive names to serve as beacons for further investigation. Seven of the ten junior developers not only added node labels and code comments, but also referenced these annotations later in their sensemaking processes. These results confirm our hypothesis that learners can benefit from sensemaking scaffolds that not only provide

multiple representations for viewing code but that also allow learners to manipulate those representations directly in flexible ways.

While we found that many Isopleth features were used extensively by our study participants, some features were underutilized. For example, custom facets were rarely used successfully by junior developers. The custom facet filters may have been too advanced for junior developers, since many did not have concrete hypotheses they could have explored through a custom filter. To reduce the complexity of authoring custom filters, future work may explore allowing users to define filters with forms and drop-downs rather than by writing predicates in code. We expect that some of the other features were underutilized because they were not needed to support some users' sensemaking processes. For example, while nine of the 14 developers modified node labels to keep track of their progress, the other developers never modified labels, with some instead highlighting code as they explored and taking mental notes to keep track of their understanding. While these other developers may still benefit from the manipulation affordances when exploring even more complex examples, their sensemaking approaches were successful without these affordances for the examples included in the study.

More generally, by providing multiple sensemaking affordances that learners could use as needed, we found that Isopleth supported a variety of different sensemaking strategies. This enabled learners to follow a flexible, self-directed sensemaking processes during which they used features as they saw fit, without being constrained to a predefined procedure. For example, we observed developers use top-down and bottom-up approaches while inspecting the call graph. We also found that junior and senior developers used different sensemaking strategies based on their background knowledge and the specificity of their

hypotheses about the feature implementation. We observed senior developers scan the call graph to look for particular functionality, while junior developers worked through nodes more systematically as they built up their conceptual models. Despite these differences in approach, we found that Isopleth helped both junior and senior developers improve the accuracy of their conceptual models, whether they rejected prior models, changed large portions, or accepted their (correct) models and added additional details.

5.8.1. Limitations

Our user study focused on evaluating how learners of varying experience levels used the Isopleth features to make sense of complex professional websites, and how the experience changed their conceptual models of the websites' implementation. As a result, we chose to design a single condition study that compared understanding before and after interacting with Isopleth. While we think this design was appropriate for addressing our immediate research questions, it does have a number of limitations. In particular, we did not compare Isopleth to a baseline to uncover how much value Isopleth adds beyond current tools for web inspection. It is likely that learners would be able to build some level of understanding by spending the same amount of time exploring website source code using currently available tools. Therefore, in the future, we would like to conduct a comparative study in which users in one condition would use Chrome Developer Tools to inspect websites, while those in another condition would use Isopleth. This comparative study would help us better understand the relative strengths and weaknesses of both tools, and help us further refine the design of Isopleth. In addition, our study was conducted in a lab, so learners were not participating due to their own motivation and interest. Learners may

apply different strategies to make sense of websites that interest them personally. As a result, we would like to conduct a in-the-wild study of Isopleth to better understand how learners explore websites in more naturalistic settings in the future. Despite the limitations of our current user study, we believe it provides an important initial understanding of Isopleth's features and the ways in which they support novices in sensemaking.

5.8.2. Future work on RALE

Our vision for Readily Available Learning Experiences (RALE) aims to empower learners to leverage the entire web of professional examples as a resource for learning programming concepts, practicing concept implementations, and applying concepts across problems. But while professional web examples provide an invaluable resource for authentic learning, they pose an enormous cognitive burden for novice developers who lack the expert models needed for making sense of complex examples. To use professional examples as a learning resource, novices must drive an inquiry-like process of making sense of complex examples, managing the investigative learning process, and reflecting on their progress to monitor and plan how best to continue learning. These skills are required for learning from complex examples, and more generally, for preparing students for CS jobs where they will be expected to learn independently and keep up with the latest technologies and techniques.

Our work on RALE seeks to overcome such challenges by advancing technologies that scaffold self-directed learning from authentic professional examples so that learners can better manage their learning process, assess what they have learned, and drive further investigation. Rather than taking a case-based approach and manually curating example cases for learners, our approach is to leverage existing professional examples that embed

the rich set of concepts and skills that learners want to practice, and to scaffold learning directly from those examples. We have chosen to use professional examples for multiple reasons. Given the complexity required for examples to embed authentic professional design patterns and practices, curating a large number of cases would be cost-prohibitive. Providing only a limited number of authentic example cases would limit their utility for learning. Moreover, given the rapid rate of technological change in the domain of web development, case libraries would quickly become out-of-date.

While Isopleth is designed to help learners understand JavaScript code, we expect our general approach of applying principles from the learning sciences to form sensemaking scaffolds that address the specific learning challenges of a context and domain to be likewise effective for learning from professional examples in other programming disciplines. As an illustrative example, our recent work led to Ply [**55**], a visual web inspector that supports novices learning to replicate the appearance of professional webpage features in CSS. To help novices build conceptual knowledge, Ply implements a novel technique called visual regression testing to hide visually-irrelevant CSS properties and surface common professional design patterns in which multiple CSS properties coordinate to produce visual effects. This helps learners build on their intuitive understanding of visual effects, and fill gaps in their conceptual knowledge about how different CSS properties worked together. Beyond learning from professional websites, our future work will take a similar approach to explore the broader applications for RALE across many languages such as Java, Python, and C++. For example, while for a Java web REST API it may not make sense to have facets for mouse, keyboard, and DOM-modifying code as Isopleth does, surfacing functionally-related slices of code is likely to still be a useful sensemaking scaffold, and

may involve defining facets around API endpoints and for discovering components such as Java controllers, services, or repository accessors.

Beyond sensemaking, our continuing work on RALE will pursue the design of learning environments and associated technologies that provide self-directed learning scaffolds for (1) *process management* to help novices implement concepts from professional examples; and (2) *reflection and articulation* to support learning to apply concepts across diverse problems. For (1), while Isopleth helps novices understand conceptually how a feature is implemented, novices can still struggle to replicate a feature or use a design pattern to implement a similar feature on their own. Practicing implementing a feature reveals additional self-directed learning challenges for novice developers who lack strategic knowledge of how to select practice activities and coordinate the process of learning [**84**]. Novices can easily become overwhelmed by the complexity of options, be distracted by less important tasks, or approach tasks that are larger than what they can productively handle. To overcome these challenges, our future work will design effective *process management scaffolds* that (a) create scaffolded exercises that decompose features into practice activities; (b) teach programming concepts and libraries in-context; and (c) provide workspaces to help learners manage their practice and test their code. These scaffolds can increase novices' ability to learn to implement complex web features by supporting their practicing component skills and then progressively combining them to implement features [**3**]; draw novices' attention to professional practice [**20, 10**]; and handle routine tasks to reduce cognitive load and disruption [**63**].

For (2), while novices can use Isopleth to study multiple examples, they may not attend to the similarities and differences in implementation approaches across examples, which is

helpful for learning how to apply programming concepts across diverse problems. Learning to apply concepts across problems presents additional self-directed learning challenges for novice developers who may explore examples haphazardly without reflecting on how concepts can be applied across examples to support future knowledge transfer [**29**]. Novices may focus on achieving quick outcomes rather than deep understanding; for example, a novice may quickly identify two examples as similar based on surface features, and fail to recognize core differences in implementation approach [**21**]. To overcome these challenges, our future work will design effective *articulation and reflection scaffolds* that (a) provide 'knowledge maps' to help learners curate and discover similar or contrasting professional examples; (b) prompt learners to label similar and contrasting implementation approaches and techniques; and (c) use guided reflection to encourage learners to reason about alternate implementation choices. These scaffolds support continual sensemaking across examples [**56**] to guide learners through analogical reasoning [**27, 28, 61, 50**]; help learners use contrasting and boundary cases to refine conceptual understanding [**11, 79, 80**]; and highlight deep features to overcome transfer issues [**68**]. Together, they can improve learners' ability to create knowledge maps that relate web features and implementation approaches, and their ability to apply concepts to construct solutions to diverse problems.

Beyond the above-mentioned self-directed learning scaffolds, novices may need support for identifying and selecting examples that are good for learning (e.g., those that embed good practices, that are at an appropriate level of complexity, that reinforce learned concepts, etc.). Moreover, learners may need help understanding why a design pattern is chosen over some other approach, and whether the programmer actually chose the right design pattern. To address these needs, we are interested in ways to embed expert

guidance into RALE, so that self-directed learning tools are used in conjunction with established curricula, expert teachers, and learning communities that help to curate content and explain difficult concepts.

Through advancing these research directions, we envision a future that provides vast opportunities for learning from the entire corpora of websites on the Internet and from publicly available professional code everywhere. This can help train increased numbers of developers who are capable of pursuing professional careers in Computer Science, and produce broadly applicable and available computer science education content that targets conceptual understanding through authentic worked examples. Beyond learning to code, RALE provides a compelling direction for generally enabling many learners to pursue mastery in complex domains by transforming real-world artifacts into authentic learning resources. Progress in this direction will require creating new interactive technologies that continue to be informed by the learning sciences, and that contribute new ways of integrating automated methods, interface affordances, and learner-created artifacts to realize the desired scaffolds for learning complex skills.

5.8.3. Enabling RALE

Isopleth is the first system to prove the techniques of RALE as a concept, and its user study shows corrective and constructive effects in conceptual model formation, distinct user sensemaking patterns compared to other systems, and deeper user understanding of the subject matter (i.e. design patterns in professional websites) than was observed in Telescope or Unravel. Isopleth scaffolds sensemaking for users by providing them low-effort entry points into exploring how a web application works while still allowing them to

explore code constructs in a fully comprehensive way. Isopleth facilitates mixed-initiative sensemaking by allowing users to create their own custom facets and code refactoring, while incorporating their additions into the recalculation of views and entry points into exploration.

CHAPTER 6

Discussion

6.1. Applications of Unravel, Telescope, and Isopleth

The three RALE tools detailed in this dissertation were created sequentially (Unravel, Telescope, then Isopleth) and extend off of each previous tool, yet they have distinct differences in use and applications for their target users class: those who can setup, read, and write basic JavaScript web applications but lack the conceptual knowledge of design patterns used in professional web solutions. Unravel works as a lightweight browser extension, and its user study showed the benefits of applying it as a tool to discover entry points into complex code. Telescope is built on a more complex architecture, but showed promise in its exploratory study for generating small examples (e.g. snippets or fiddles) of dynamic web features and the interplay between JavaScript and HTML. Isopleth extends Telescope's architecture, but shows relationships in the code and gives users scaffolds to make sense of complexities they encounter. Different from prior work and existing inspection tools, Unravel, Telescope, and Isopleth are specifically designed to help users understand unfamiliar professional web application code by overcoming learning barriers and scaffolding sensemaking.

6.1.1. Unravel Applications

When curious about an interesting feature on a website, Unravel helps experts navigate quickly to code likely supporting the feature, and it helps beginners overcome the information barrier of where to begin inspection. Tools created prior to this dissertation [**5, 45, 43, 53, 31, 32, 14, 36**] are certainly capable of exposing feature functionality but were designed to debug applications under development rather than delve into completely unfamiliar code. For web features with tens of thousands of function invocations, it quickly becomes unclear — even to experts — how the application works due to overwhelming amounts of debugging information. Prior tools for exploring unfamiliar code make contributions to record and diff program state but still are subject to overwhelming users with too much runtime information when recording thousands of function invocations or DOM changes [**67, 15, 14, 2**]. Unravel uses excessive runtime counts to its advantage by bubbling relevance based on counts and reducing output by unique DOM element and unique call stack. Users look to Unravel to provide sortable filtered lists of DOM changes and function invocations; they can use the lists as an index linking into specific locations in the well-featured Chrome Developer Toolkit.

An example scenario is for a developer who has a vague idea of how to achieve a card-swipe transition effect but wants to check comparable websites to quickly consider and compare other professional strategies. The developer launches Unravel and looks for specific function invocations calling API's like animation, iterating over CSS properties, or invoking frameworks. The developer sees which techniques are being supported by other companies in production. After finding and inspecting a website with a cleaner and

more maintainable pattern, the developer decides to follow their pattern of adding and removing CSS-animate classes via JavaScript.

But beyond finding quick entry points into the application's code, further exploration is inhibited by information barriers in having to navigate large amounts of code. Telescope can be applied to produce low-barrier learning materials from large amounts of code.

6.1.2. Telescope Applications

When interested in discovering *all* of the JavaScript and HTML necessary to recreate a web feature in an efficient way, Telescope is able to display relevant source code at variable levels of detail in its composite view. Telescope overcomes Unravel's main limitations of JavaScript observation scope (DOM only) and shallow exploration (simple pointers into the code) by extracting all the code from a website and displaying it in a composite view for the user, condensing JavaScript based on a configurable detail level. Unravel displays HTML and JavaScript function invocation counts in different pains with links to Chrome Developer Tools, while Telescope can display all of the JavaScript and HTML for a website with visual links between the two languages. Telescope enables users to vary the level of detail of JavaScript displayed between: DOM-querying, invoked, library, and all. This helps users see different patterns in JavaScript such as in-memory data management in an MVC architecture. Similar to a fiddle or code snippet, Telescope's output can be used as a starting point to recreate or feature or learn new design patterns in a professional website. While Unravel provides pointers into the browser's web inspector, Telescope pulls out relevant views of JavaScript for the user.

An example scenario involves a developer who wants to recreate a draggable map interface they found on their website, but they aren't aware of any techniques to create this implementation. Briefly searching the web, all of the techniques they find are either overly cumbersome (such as deploying a map-creator SDK) or too dependent on a third-party (such as Google Maps). Knowing they simply want to create a similar solution to what they see, they activate Telescope in the website and identify a simple 60-line starting set of how to construct a draggable-map component in JavaScript. After looking through the code toward the beginning of the timeline, they want to see how code later in the timeline modifies the initial draggable-map component. They move the timeline and increase the detail level to see a set of active event-bindings and listeners. With a set of starting code, users wishing to gain more insight into professional web application development can discover techniques while using parts of the code to craft their own custom solution.

6.1.3. Isopleth Applications

When interested in learning new professional web development techniques more comprehensively, Isopleth provides opportunistic views to learners and scaffolds basic sensemaking through mixed initiative affordances. Somewhat like a worksheet that adapts to its student, users can work through findings in Isopleth and add their own custom facets and names; Isopleth reshapes its views in response. Isopleth captures JavaScript invocations by extending Telescope's architecture, but instead of displaying variable detail levels of relevant JavaScript, Isopleth displays reduced and filtered views of a JavaScript call graph based on facets (i.e. code constructs defined by their inputs and outputs). Users can see

how functions were called, how functions relate to other functions, which hidden asynchronous links exist between functions, and how calls are classified via their facet labels. With these unique views fully decomposing an unfamiliar JavaScript web application, users can study small pieces and relationships in an effort to build an accurate mental model of how a web feature was created. Isopleth is the first of the three tools in this thesis to offer learning scaffolds, which helps inexperienced developers make sense of complex code.

An example scenario is for a frustrated self-starting web developer who is eager to understand how click-and-drag bindings work in JavaScript but is overwhelmed even by Telescope's simplified output, as it lacks program flow and asynchronous relationships. Telescope condenses website code into learning materials, but in doing so loses structures and dependencies in the code necessary to form a deeper understanding. The developer doesn't understand how a function could be run before it is declared; that should not be possible. The developer launches Isopleth and starts examining nodes in the call graph, broken down into small pieces with relational asynchronous links. The developer notices that the function is actually declared in the first tree in the call graph, as it is labeled with a *setup* facet. The developer follows an asynchronous relationship line to the right side of the call graph where a drag function invocation has occurred. It is labeled with a mouse facet. After studying the functions and following relationships in their trees, the developer sees how the function was bound to click-drag events at setup and invoked later on. By using Isopleth, the developer has corrected an inaccurate mental model of the application.

6.1.4. Future Toolkit

In the future, it is conceivable to imagine a single tool that encompasses all of the functionality of Unravel, Telescope, and Isopleth without reliance on third party technology. By integrating Isopleth's instrumentation in a browser's JavaScript precompile step, all of the information needed to bubble relevance, vary detail, filter libraries, or deanonymize anonymous asynchronous calls would be available to a live inspector. No third party server architecture would be needed. Note: the intended goal of the existing architecture is to be agnostic of browser implementation, but the proposed implementation here would require a customized integration per browser.

6.2. RALE: Design Claims and Evidence

This thesis has contributed three systems toward the goal of creating Readily Available Learning Experiences for professional websites. This section details the primary design claims of Readily Available Learning Experiences and reviews evidence of their necessity. In brief, a RALE should:

- Surface hidden design patterns, code constructs, and relationships (both direct and indirect) from professional websites.
- Minimize learning barriers while supporting personalized exploration of unfamiliar website code.
- Scaffold mixed-initiative sensemaking to help users walk through unfamiliar complexities in the surfaced resources.
- Scale the conversion of examples into learning resources without additional authorship or maintenance.

6.2.1. Surfacing Hidden Patterns, Constructs, and Relationships

RALE addresses a class of users who are frustrated by their knowledge gaps in web development and thus limited in their ability to interpret complex professional code. After completing web tutorials, a beginner might attempt a new project but realize they lack sufficient knowledge to complete their project goals. Inspired by existing professional examples, a beginner might inspect the professional website, but in doing so find that it is unclear where to find and how to interpret design patterns, code constructs, and programmatic relationships. Surfacing patterns, constructs, and relationships from professional websites helps users overcome gaps in knowledge, shortcuts inefficient forms of web foraging such as searching the web for relatable tutorials and Q&A posts, and provides them with opportunities to learn authentically — in a personally meaningful way using multiple modes of the discipline (i.e. web application programming).

Design patterns in this context are defined as reusable techniques or sets of techniques which can be applied in multiple situations to solve similar problems. While many design patterns are well known and have names (e.g. Model-View-Controller, Bootstrapping, Lazy-Loading), naming a pattern is less a concern in RALE than helping a learner build an accurate reusable mental model of a design pattern. A popular unstandardized design pattern in web development involves toggling a CSS class on a DOM element via JavaScript to achieve a show/hide or animation. Test users found this simple yet effective pattern on Tumblr, BBC, Amazon, and Kickstarter using Unravel, Telescope, and Isopleth. While none of the tools actually observe CSS, they were successful in identifying the patterns that operate on CSS by surfacing constructs that operate on DOM elements.

Surfacing relevant code constructs and relationships between them informs users about the coordination among components to achieve an effect. Identifying relevance in the constructs provides users with entry cues as to which constructs or relationships should be examined first. Code constructs in this case are functions, sets of procedures, or DOM structures. Relationships are direct or indirect dependencies or operations between code constructs. Together, code constructs and their relationships are the subjects of study in a RALE, which inform higher level design patterns. While Unravel lacks relational information, its construct ordering of function invocations and DOM element changes helps users quickly identify entry points into learning from complex applications. Telescope collects active code constructs together in a composite view, shows invocation counts, and relationally links HTML to JavaScript constructs. Telescope's exploratory study highlighted the advantages of surfacing a composite view of constructs and their relationships to help the user determine the scope of a feature's implementation and its interplay between constructs. Isopleth fully decomposes constructs of a web application feature in its call graph and visualizes direct and indirect relational links in JavaScript. With Isopleth users can easily determine the hierarchy and dependencies of constructs as well as links between setup, runtime, and specific facets.

6.2.2. Minimizing Learning Barriers

There are many tools that enable developers to inspect and step through every line of code responsible for a feature [**67, 15, 2, 31, 5**], yet developers still are overwhelmed in learning from professional websites due to the size and complexity of modern web applications [**6, 77, 86**]. While these tools are fully featured and support deep inspection into the runtime

of a web application, they were designed to serve purposes of inspection and debugging and thus streamline goals toward program maintenance and implementation. But the comprehensive nature of these tools can introduce additional barriers to a learner when trying to understand complex concepts. In Ko et al's work on *six learning barriers*, they identified barriers such as the Information Barrier where users had helpful information available to them but didn't know to look for it or read it, or the coordination barrier where users were confused with how multiple components worked together to achieve an outcome [**49**]. Presenting users with detailed stack traces, variable states, and nested program flows could easily introduce additional barriers to a learner. Thus it is a primary goal of RALE to minimize the effects of additional learning barriers created by surfacing hidden details from software.

Most notable in Telescope's user studies and Unravel's controlled study, users greatly benefitted from being shown informational views tuned specific to their goals to help overcome learning barriers. In the early prototypes of Telescope, users were given detailed call stacks and verbose amounts of active JavaScript and function hit counts. While some of the test users appreciated being able to visualize the inner workings of a complex web application, they described confusion in trying to understand the overly verbose output. In three user-centered design iterations, the output of Telescope was reshaped to show JavaScript activity most relevant to visual changes in a web application first, linked to active HTML (i.e. the composition of the rendered website view), with the option to expand detail. These changes diminished the effects of information and coordination barriers, because users were not required to sift through large sets of function invocations or to make

manual connections between HTML and JavaScript. Telescope's exploratory study revealed that most users appreciated the default views with more limited information to gain an introductory understanding of unfamiliar website code. Unravel's user study findings highlight the importance of filtering and aggregating large volumes of information to streamline the search for entry points into website runtime activity.

In addition to shaping default informational views to minimize learning barriers, users with different gaps in knowledge are subject to different learning barriers and need flexibility in the display of data. Unravel's user study showed a strong decrease in efficiency beyond surfacing the first relevant code construct; Unravel only allows for the filtering and sorting of aggregated information rather than expanding upon levels of detail like in Telescope or shaping information display with rich labels, or facets, in Isopleth. Telescope's final design incorporated user feedback from Unravel and earlier Telescope prototypes that users need to see different levels of detail at different times. In using Telescope with expanded detail beyond just DOM-querying JavaScript, users found richer design patterns such as lazy-loading or view-routing. Isopleth provides users with an interactive call graph and facets to simplify the display of information; when users add new facets to Isopleth, its views are further enriched with their customization. During Isopleth's user study, participants commonly began their searches at the rightmost portion of its graph for simplicity, but expanded their search backwards in time in Isopleth's call graph to construct mental models about the program's constructs and relationships. Therefore, it is important to not only surface information in RALE but to allow for personalized exploration of the data surfaced.

6.2.3. Scaffolding Mixed-Initiative Sensemaking

RALE builds upon a rich body of work in automated tutoring [**17, 38, 51, 47, 36**], program visualization [**89, 24, 37, 12**], and web application exploration [**67, 14, 15**] by setting sensemaking and learning scaffold goals based on strongly grounded literature in the learning sciences [**56, 70, 72, 91, 1, 92, 57, 62, 18, 19, 22**]. Prior tools such as FireCrystal, Scry, and WhyLine expose hidden aspects of program behavior and even allow the user to query parts of their interface as scaffolds for sensemaking, but they are not designed to support a learner's progression from writing functional code to writing professional-quality software. Specifically, they surface constructs but not design patterns from professional code; they do not decompose complexity into explorable pieces, and they do not allow users to customize or extend the interface. As a result, they lack authenticity and do not provide opportunities for learners to think in the modes of the discipline [**72**]. RALE furthers the goal of these tools by calling for techniques and affordances to support learners during their sensemaking process and provide them with cues to engage in multiple modes of the web programming discipline such as architecture, implementation, and refactoring.

While many of the prior tools lack evaluations, results from evaluating Unravel and Telescope emphasized the necessity of mixed-initiative sensemaking in designing RALE. Most tools in this body of work, with the exception of Isopleth and Bret Victor's Learnable Programming [**89**], fall into either categories of read-only interfaces which tell users about hidden activities [**2, 67, 15, 53**] in the program runtime or read-and-query interfaces which allow users to query for certain informational views revealing details about hidden program activity [**17, 47, 52, 48**]. In Bret Victor's programming environment,

affordances are provided to directly visualize the effects of a user's design decisions in code while they are attempting to modify or build a project. Ideally a RALE should scaffold sensemaking by allowing users to not only observe information about a program's runtime, but to modify and personalize that information as a mixed-initiative with the system — each contribution between the user and system provides gains for the user [**42**] towards the goal of learning new professional programming concepts.

Isopleth is the first system to prove the techniques of RALE as a concept, and its study shows strong effects in conceptual model formation, distinct user sensemaking patterns compared to other systems, and deeper user understanding of the subject matter (i.e. design patterns in professional websites) than was observed in Telescope or Unravel. Isopleth scaffolds sensemaking for users by providing them low-effort entry points into exploring how a web application works while still allowing them to explore code constructs in a fully comprehensive way. Isopleth facilitates mixed-initiative sensemaking by allowing users to create their own custom facets and code refactoring, while incorporating their additions into the recalculation of views and entry points into exploration. In using Isopleth effectively, users were encouraged to modify Isopleth's source code views, facets, and graph nodes for their personalized goals. This allowed them to work slowly through understanding parts of the program in a bottom-up code comprehension strategy. While some of Isopleth's study participants described new mental model formation of observed professional website behaviors, others enriched their existing mental models with implementation patterns important to production environments (e.g. caching queries to speed up query results or abstracting authentication to support proprietary and social logins). Though Isopleth was designed for beginning developers struggling to fill their

gaps in knowledge, experts appreciated the efficiency of Isopleth's scaffolds, entry cues, and filtering in streamlining their searches for functionality. Isopleth's study participants described the decompositional graphing of function invocations as a streamlined way to work through code constructs, where each node is like a simple "to do" in working toward an understanding how the application works.

Much future work remains beyond Isopleth in advancing mixed-initiative sensemaking scaffolds in RALE. Quintonna et al provide rich guidelines for scaffolding sensemaking in software inquiry [**72**], and the current body of work provides a platform to apply these strategies. The following itemization characterizes how new strategies can be applied given the current platform:

- Embed Expert Guidance: Users in Isopleth's evaluation desired to know more about the design patterns they discovered, such as why one pattern was chosen over another. Future tools can incorporate expert commentary on professional design decisions surfaced by existing tools.
- Provide Structure for Complex Tasks and Functionality: Existing tools provide a limited set of techniques to enable source exploration and scaffold sensemaking. For more complex learning tasks, future tools could incorporate additional structures and scaffolds to guide users on a particular learning path, such as sandboxing part of an application and having users incrementally recreate a feature in order to gain authentic practice.
- Facilitate Navigation Among Tools: Many tools exist in source code exploration and tutorial-generation, yet without guidance, users are generally unaware of the pros and cons of using different tools. Future work could incorporate a

survey of this body of work, conducting studies on many of the tools lacking evaluation to determine their effectiveness in scaffolding sensemaking, enabling code comprehension, and overcoming learning barriers.

- Facilitate Ongoing Articulation and Reflection: The current suite of tools elicits concepts from professional code and provides basic learning scaffolds but lacks ongoing reflection (other than repeat use of the tools). Quintonna states that learners often do not know to articulate their ideas, or they need help to do so productively [**72**]. Future tools could provide mechanisms for learners to record their findings for review, reuse, and practice in other domains.

6.2.4. Scale Continuously without Burdens of Authorship or Maintenance

With millions of active learners looking to the web for online learning along with the fast pace of innovation in web programming, teachers and content authors meet only a small portion of the ever-expanding demand for learning materials. The primary goal of RALE is to transform inspiring professional websites into opportunities for learning with no dependencies on authoring. Developers often turn to the web to forage for designs, web features, interactive techniques, and performance optimizations they wish to learn how to create [**9**]. With limited articles and tutorials keeping up with the pace of innovation, users turn to Q&A networks, chat rooms, web tutorials, and MOOCs. Social platforms like Q&A and chat rooms have social norms; they can seem abrasive to newcomers who aren't familiar with the correct terms in which to ask their question [**58**]. Web tutorials are limited and can grow outdated or even contradict one another in terms of best practices or design patterns to use (e.g. do or don't lazy-load content). After completing MOOCs,

significant gaps in knowledge can remain when trying to transition from the pre-authored material to new project scenarios. Therefore, a central claim of RALE is that it must scale continuously across its medium without burdens of authorship (i.e. manual creation of learning materials) or maintenance (i.e. manual support for unwinding technological features into learning content).

Unravel, Telescope, and Isopleth were designed with the goal to scale to the domain of professional websites on the open web, which enables both authentic learning and the continuous creation of learning materials. Prior tools either required users to perform technical workarounds to gain full access to web code [**53, 2, 67**] or only provided views of DOM-querying JavaScript [**14, 15, 17**]. To support authentic learning, systems should (1) surface the rich details from professional websites that are missing from training examples while (2) providing users with opportunities to think in multiple modes of the discipline (i.e. software engineering), and (3) embed programming concepts and implementation techniques that are used by professionals when creating scaffolds for users to explore. To avoid additional barriers to learning, these systems should not require users to have knowledge of advanced inspection techniques to surface learning materials or scaffolds. Unravel, Telescope, and Isopleth are each deployed via one-click workflows when users encounter professional websites of interest. Learning materials are surfaced for them, and scaffolds are created automatically. Each of these tools currently supports discovery on the web and will continue to support future discovery.

6.3. Broader Applications of RALE

6.3.1. RALE in Software

Beyond learning from source code in professional websites, broader applications exist for RALE in other programming disciplines. The central design claims of RALE described in this thesis generalize to other venues of professional source code for future development of RALE tools. By extending the instrumentation and graph analysis techniques in Theseus [**53**], Telescope, and Isopleth, mixed-initiative sensemaking scaffolds could be created across many languages such as Java, Python, and C++. In each new application of RALE it is important to understand common barriers users face when learning from professional source code and surface which techniques or patterns they seek to understand. For example, in a Java web REST API, it would not make sense to surface facets in a call graph about mouse, keyboard, and DOM-modifying code. It would be more appropriate to surface API endpoints and the relevant operations on web API request (e.g. scaffold the user through discovering Java controllers, services, and repository accessors). By extending RALE to new programming disciplines, new opportunities are introduced to draw parallels and distinctions among patterns in multiple technologies.

Design Techniques from Unravel, Telescope, and Isopleth have broader potential in other programming disciplines. Unravel demonstrated techniques to surface relevance in code by bubbling invocation counts and UI changes, with links into code inspection. Telescope provided a composite of a minimal set of code comprised of invocation counts, runtime timestamps, and library filtering with interactive visual lines to UI code. Isopleth provided mixed-initiative sensemaking scaffolds on function invocations linked by

direct and indirect calls. Similar programming techniques are present in a wide array of programming languages, and with more languages adopting functional design and lambda expressions, the need for visualizing and learning from complex construct relationships is increasing. Ko et al's learning barriers exist in many languages [**49**], calling for new solutions to guide incoming learners. Potential drawbacks to applying RALE techniques in other disciplines stem from the availability of source code. Proprietary packaging or compilers could make RALE generation difficult. While the source code of many professional software products may not be available, large public open source repositories such as Github can serve as a rich repository of open source code. It is common to find many of the same professionals who have developed proprietary technology developing open source and publishing on a company-sponsored public repository (e.g. Adobe, Google, Microsoft, Amazon).

6.3.2. Learning Technique Trade-offs

Broader applications for RALE have learning tradeoffs when compared to existing learning techniques in each domain. 1-on-1 tutoring, mentoring, or formal courses are excellent ways to prepare for professional work, however the supply of tutors and mentors is not scalable and formal courses are not accessible to many. Further, the learning resources provided are finite and thus not fully comprehensive across a domain of potential professional applications. RALE can be helpful in scaling authentic learning beyond the current learning system's limitations. However, while RALE relieves limitations in authoring, there is currently no supported technique to inform users that materials generated from professional products might be either outdated, invalid, or contain bad practices.

Even though code may be developed by professionals, it may not always meet high standards in software engineering. Authored tutorials, guides, books, and lectures often have indicators as to their timely relevance, such as publication date, community comments, software versions, and compatibility. These indicators may be unavailable through the RALE technique of automating learning materials. A potential solution for this is to incorporate communities of learners to discuss and reflect upon design patterns or techniques they have discovered using RALE. Communities for discussing techniques and patterns already exist in different domains [1] [2], thus there exists a potential to leverage communal knowledge to overcome risks in RALE quality.

6.3.3. RALE in Other Disciplines

Even more broadly speaking, it is not understood yet which claims in RALE are generalizable to other knowledge domains, such as art or physics. For example, by surfacing techniques used to create a professional painting and scaffolding learning about the purpose and usage pattern of said techniques, an aspiring artist could bridge gaps in their knowledge of the discipline. Professionally developed products or deliverables capture and inspire learner interest and promote new opportunities for a RALE to enable authentic learning. Given new techniques to access the underlying constructs that formulated a product, or access the intentions and purposes for which specific techniques were used (e.g. in art), a RALE could be envisioned that operates on data about the underlying constructs or techniques by lowering learning barriers, scaffolding sensemaking, and continuously scaling to new products in the domain. Beyond professionally developed

[1] Stack Overflow Documentation `https://stackoverflow.com/tour/documentation`
[2] Gitter: Developer Chat per Open Source Repository `https://gitter.im/`

products, objects and interactions in nature raise interesting questions for what RALE could enable. Stemming from the central claims of RALE, a RALE could be envisioned for learning why an object occurs and behaves the way it does in nature — given some techniques to expose physical data (e.g. sensors). For example, as an apple falls from a tree, a RALE in nature could surface information about the underlying physical laws and variable states of mass, gravity, friction, and momentum while minimizing learning barriers, scaffolding sensemaking, and scaling continuously across multiple examples of falling objects and gravity. While many questions remain for extending RALE to other disciplines, its central claims can be revisited to conceptualize basic frameworks for guiding learners in new applications and domains.

CHAPTER 7

Conclusion

This thesis introduces Readily Available Learning Experiences (RALE) for professional web applications. Its goal is to help inexperienced learners who wish to become professional contributors but lack the means necessary to advance beyond their gaps in knowledge. The central claims of RALE are (1) surfacing hidden design patterns, code constructs, and relationships (both direct and indirect) from professional websites, (2) minimizing learning barriers while supporting personalized exploration of unfamiliar website code, (3) scaffolding mixed-initiative sensemaking to help users walk through unfamiliar complexities, and (4) scaling the conversion of examples into learning resources without additional authorship or maintenance. To conclude, this chapter will summarize the main contributions and discuss future directions of research.

7.1. Summary of Contributions

The systems and corresponding studies in this thesis more broadly contribute techniques to support reverse engineering, create low-barrier learning materials, and scaffold users into opportunistic sensemaking processes.

- **Unravel: Reverse Engineering**: Unravel's conceptual contribution is the idea of tracing, identifying, and organizing the most relevant functionality to help users find interaction code quickly in complex professional website code. Unravel's technical contribution is a lightweight *API harness* technique designed

to capture specific source code traces to an API. Unravel's evaluation measures the effectiveness of its reverse engineering techniques on novice and professional programmers.

- **Telescope: Creating Low-Barrier Learning Materials**: Telescope's conceptual contribution is the idea of helping users understand complex website code by generating low-barrier learning materials from features of interest. Telescope's technical contributions are the *Wisat* architecture and *Sleight-of-Hand* technique; they enable the capturing of comprehensive runtime JavaScript traces on public websites. Telescope's evaluation measures its performance, effects, and limitations of generating low-barrier learning materials on professional websites.
- **Isopleth: Scaffolding Sensemaking**: Isopleth's conceptual contribution is the idea of scaffolding sensemaking of complex professional code by surfacing hidden relationships between code constructs and providing a mixed-initiative process to interactively explore, label, and identify system components and how they relate. Isopleth's technical contribution is a Serialized Deanonymization (SD) technique that places unique identifiers in all functions in a web application's JavaScript source to trace how functions are bound, passed, returned, and invoked asynchronously. Isopleth's case study measures its capabilities in making sense of concepts in a website, and its user study measures its effectiveness in sensemaking and mental model formation.

7.2. Future Directions

Research in designing, developing, and evaluating RALE has just begun. With a continuum of curious builders and tinkerers, there are many potential venues for RALE to grow in its application space. This section addresses the next logical direction for RALE to continue in the context of professional web applications.

7.2.1. Learning Communities

Based on the techniques from Telescope and Isopleth (Chapters 4 and 5), professional website UI features now have a potential be transformed into portable and indexable learning-examples for users to share and build upon. Shareable real-world examples could provide the basic blocks to build learning communities, as they are inherently relevant and meaningful to learners eager to gain professional experience.

Creating a UI interaction implementation library could help developers with varied experience levels discover techniques used on the web. Junior web developers struggle with creating UI interactions, and experienced web developers have difficulty keeping up with the latest techniques. For example, a user might search for an autocomplete implementation and have the option to compare source code underlying well designed interfaces from Google, Twitter, and Facebook. Further, indexed UI traces from telescope code could be used in-context within IDE's through technologies like Codeletes and BluePrint [**66, 8**]. With labeling and UI metadata, learning material output could be indexed for mining UI *behaviors*, or the combination of user-prompted interaction and underlying source code traces. Output from this mining could be used to elicit implementation patterns or best practices across websites.

7.2.2. Developer Tooling

The output of the systems in this thesis could be integrated into research tools with similar learning goals. This integration could create new opportunities to receive help while developing software [**26, 8**], provide micro explanations of code constructs [**38**], or train classifiers in context-based variable naming and unminification [**74**]. Enhancing Telescope to support *webstrates* would allow bidirectional modification of a website UI, giving users the opportunity to "sandbox" their UI discovery with a real website [**46**]. To maintain the consistent goals of RALE, subsequent research should continue to construct effective technical solutions that are well-grounded in the learning sciences.

7.2.3. Learning Pathways

While this thesis has addressed the extraction, creation, and design of learning materials for making sense of professional websites, much work remains in guiding the growth of a learner *during* their transition from learner to professional contributor. For example, it is unknown how to create guides for learning in RALE, such as a curriculum-design or learning goals. Providing users with learning pathways could give them clear direction of how to achieve specific learning goals with the materials they have from RALE. Helping users reach learning milestones or achievements could provide them with newfound confidence in their abilities to professionally develop software.

Learning to code should be more accessible to everyone. With platforms and techniques that enable Readily Available Learning Experiences in professional websites, I aim to continue lowering the barriers to learning in Computer Science.

References

[1] Beth Adelson and Elliot Soloway. The role of domain expenence in software design. *IEEE Transactions on Software Engineering*, (11):1351–1360, 1985.

[2] Saba Alimadadi, Sheldon Sequeira, Ali Mesbah, and Karthik Pattabiraman. Understanding javascript event-based interactions. In *Proceedings of the 36th International Conference on Software Engineering*, pages 367–377. ACM, 2014.

[3] Susan A Ambrose, Michael W Bridges, Michele DiPietro, Marsha C Lovett, and Marie K Norman. *How learning works: Seven research-based principles for smart teaching*. John Wiley & Sons, 2010.

[4] Jake Archibald. Deep dive into the murky waters of script loading, 2013. Available at `http://www.html5rocks.com/en/tutorials/speed/script-loading/`.

[5] John J Barton and Jan Odvarko. Dynamic and graphical web page breakpoints. In *Proceedings of the 19th international conference on World wide web*, pages 81–90. ACM, 2010.

[6] Michael Bolin. *Closure: The Definitive Guide*. " O'Reilly Media, Inc.", 2010.

[7] Bower.io. Bower javascript library install statistics., 2015. Available at `http://bower.io/stats`.

[8] Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R Klemmer. Example-centric programming: integrating web search into the development environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 513–522. ACM, 2010.

[9] Joel Brandt, Philip J Guo, Joel Lewenstein, Mira Dontcheva, and Scott R Klemmer. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1589–1598. ACM, 2009.

[10] John D Bransford, Ann L Brown, and Rodney R Cocking. How people learn, 2000.

[11] John D Bransford, Jeffery J Franks, Nancy J Vye, and Robert D Sherwood. New approaches to instruction: Because wisdom can't be told. *Similarity and analogical reasoning*, 470:497, 1989.

[12] Simon Breslav, Azam Khan, and Kasper Hornbæk. Mimic: visual analytics of online micro-interactions. In *Proceedings of the 2014 International Working Conference on Advanced Visual Interfaces*, pages 245–252. ACM, 2014.

[13] Ruven Brooks. Towards a theory of the comprehension of computer programs. *International journal of man-machine studies*, 18(6):543–554, 1983.

[14] Brian Burg, Richard Bailey, Andrew J Ko, and Michael D Ernst. Interactive record/replay for web application debugging. In *Proceedings of the 26th annual ACM symposium on User interface software and technology*, pages 473–484. ACM, 2013.

[15] Brian Burg, Andrew J Ko, and Michael D Ernst. Explaining visual changes in web interfaces. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, pages 259–268. ACM, 2015.

[16] Nicholas Carlini, Adrienne Porter Felt, and David Wagner. An evaluation of the google chrome extension security architecture. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 97–111, 2012.

[17] Kerry Shih-Ping Chang and Brad A Myers. Webcrystal: understanding and reusing examples in web authoring. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 3205–3214. ACM, 2012.

[18] William G Chase and Herbert A Simon. Perception in chess. *Cognitive psychology*, 4(1):55–81, 1973.

[19] Michelene TH Chi, Paul J Feltovich, and Robert Glaser. Categorization and representation of physics problems by experts and novices. *Cognitive science*, 5(2):121–152, 1981.

[20] Allan Collins. Design issues for learning environments. *International perspectives on the design of technology-supported learning environments*, pages 347–361, 1996.

[21] Elizabeth A Davis and Marcia C Linn. Scaffolding students' knowledge integration: Prompts for reflection in kie. *International Journal of Science Education*, 22(8):819–837, 2000.

[22] Adriaan D De Groot. *Thought and choice in chess*, volume 4. Walter de Gruyter GmbH & Co KG, 1978.

[23] Brenda Dervin and Patricia Dewdney. Neutral questioning: A new approach to the reference interview. *Rq*, pages 506–513, 1986.

[24] Pierre Dragicevic, Stéphane Huot, and Fanny Chevalier. Gliimpse: Animating from markup code to rendered documents and vice versa. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*, pages 257–262. ACM, 2011.

[25] James R Eagan, Michel Beaudouin-Lafon, and Wendy E Mackay. Cracking the cocoa nut: user interface programming at runtime. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*, pages 225–234. ACM, 2011.

[26] Ethan Fast and Michael S Bernstein. Meta: Enabling programming languages to learn from the crowd. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, pages 259–270. ACM, 2016.

[27] Dedre Gentner. Structure-mapping: A theoretical framework for analogy. *Cognitive science*, 7(2):155–170, 1983.

[28] Dedre Gentner, Jeffrey Loewenstein, and Leigh Thompson. Learning and transfer: A general role for analogical encoding. *Journal of Educational Psychology*, 95(2):393, 2003.

[29] Mary L Gick and Keith J Holyoak. Schema induction and analogical transfer. *Cognitive psychology*, 15(1):1–38, 1983.

[30] Elena L Glassman, Lyla Fischer, Jeremy Scott, and Robert C Miller. Foobaz: Variable name feedback for student code at scale. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, pages 609–617. ACM, 2015.

[31] Google. Dev tools tips and tricks, 2016. Available at `https://developer.chrome.com/devtools/docs/tips-and-tricks`.

[32] Google. Inspect and edit pages and styles, 2016. Available at `https://developers.google.com/web/tools/chrome-devtools/iterate/inspect-styles/`.

[33] Google. Chrome rendering and profiler tools, 2017. Available at `https://developers.google.com/web/tools/chrome-devtools/rendering-tools/`.

[34] Google. Enabling the async callstack, 2017. Available at `https://developers.google.com/web/tools/chrome-devtools/javascript/step-code#enable_the_async_call_stack`.

[35] Paul Gross and Caitlin Kelleher. Non-programmers identifying functionality in unfamiliar code: strategies and barriers. *Journal of Visual Languages & Computing*, 21(5):263–276, 2010.

[36] Paul Gross, Jennifer Yang, and Caitlin Kelleher. Dinah: An interface to assist non-programmers with selecting program code causing graphical output. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 3397–3400. ACM, 2011.

[37] Philip J Guo. Online python tutor: embeddable web-based program visualization for cs education. In *Proceeding of the 44th ACM technical symposium on Computer science education*, pages 579–584. ACM, 2013.

[38] Andrew Head, Codanda Appachu, Marti A Hearst, and Bjorn Hartmann. Tutorons: Generating context-relevant, on-demand explanations and demonstrations of online code. In *Visual Languages and Human-Centric Computing (VL/HCC), 2015 IEEE Symposium on*, pages 3–12. IEEE, 2015.

[39] Joshua Hibschman and Haoqi Zhang. Unravel: Rapid web application reverse engineering via interaction recording, source tracing, and library detection. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, pages 270–279. ACM, 2015.

[40] Joshua Hibschman and Haoqi Zhang. Telescope: Fine-tuned discovery of interactive web ui feature implementation. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, pages 233–245. ACM, 2016.

[41] Ariya Hidayat. Detecting JavaScript Libraries and Versions. don't code today what you can't debug tomorrow., 2015. Available at `http://ariya.ofilabs.com/2013/07/detecting-js-libraries-versions.html`.

[42] Eric Horvitz. Principles of mixed-initiative user interfaces. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, pages 159–166. ACM, 1999.

[43] Mozilla Jan Honza Odvarko. Firebug breakpoints feature, 2016. Available at `https://getfirebug.com/doc/breakpoints/demo.html#dom`.

[44] jQuery. Current active browser support, 2017. Available at `https://jquery.com/browser-support/`.

[45] Google Kayce Basques. Pause your code with breakpoints, 2016. Available at `https://developers.google.com/web/tools/chrome-devtools/javascript/breakpoints`.

[46] Clemens N Klokmose, James R Eagan, Siemen Baader, Wendy Mackay, and Michel Beaudouin-Lafon. Webstrates: Shareable dynamic media. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, pages 280–290. ACM, 2015.

[47] Andrew J Ko and Brad A Myers. Designing the whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 151–158. ACM, 2004.

[48] Andrew J Ko and Brad A Myers. Extracting and answering why and why not questions about java program output. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(2):4, 2010.

[49] Andrew Jensen Ko, Brad A Myers, and Htet Htet Aung. Six learning barriers in end-user programming systems. In *Visual Languages and Human Centric Computing, 2004 IEEE Symposium on*, pages 199–206. IEEE, 2004.

[50] Janet L Kolodner. Educational implications of analogy: A view from case-based reasoning. *American psychologist*, 52(1):57, 1997.

[51] Michael J Lee and Andrew J Ko. Personifying programming tool feedback improves novice programmers' learning. In *Proceedings of the seventh international workshop on Computing education research*, pages 109–116. ACM, 2011.

[52] Raimondas Lencevicius, Urs Hölzle, and Ambuj K Singh. Query-based debugging of object-oriented programs. In *ACM SIGPLAN Notices*, volume 32, pages 304–317. ACM, 1997.

[53] Tom Lieber, Joel R Brandt, and Rob C Miller. Addressing misconceptions about code with always-on programming visualizations. In *Proceedings of the 32nd annual ACM conference on Human factors in computing systems*, pages 2481–2490. ACM, 2014.

[54] Einar Lielmanis. Beautify-web/js-beautify, 2015. Available at `https://github.com/beautify-web/js-beautify`.

[55] Sarah Lim. Ply: Visual regression pruning for web design source inspection. In *Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems*, pages 130–135. ACM, 2017.

[56] Marcia C Linn. Designing computer learning environments for engineering and computer science: The scaffolded knowledge integration framework. *Journal of Science Education and technology*, 4(2):103–126, 1995.

[57] Marcia C Linn and Michael J Clancy. The case for case studies of programming problems. *Communications of the ACM*, 35(3):121–132, 1992.

[58] Lena Mamykina, Bella Manoim, Manas Mittal, George Hripcsak, and Björn Hartmann. Design lessons from the fastest q&a site in the west. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 2857–2866. ACM, 2011.

[59] Josip Maras, Jan Carlson, and Ivica Crnkovi. Extracting client-side web application code. In *Proceedings of the 21st international conference on World Wide Web*, pages 819–828. ACM, 2012.

[60] Josip Maras, Maja Stula, Jan Carlson, and Ivica Crnkovic. Identifying code of individual features in client-side web applications. *Software Engineering, IEEE Transactions on*, 39(12):1680–1697, 2013.

[61] Mark A McDaniel and Carol M Donnelly. Learning with analogy and elaborative interrogation. *Journal of Educational Psychology*, 88(3):508, 1996.

[62] Tanya J McGill and Simone E Volet. A conceptual framework for analyzing students' knowledge of programming. *Journal of research on Computing in Education*, 29(3):276–297, 1997.

[63] Yoshiro Miyata and Donald A Norman. Psychological issues in support of multiple activities. *User centered system design: New perspectives on human-computer interaction*, pages 265–284, 1986.

[64] Mozilla. Mozilla remote debugging protocol, 2016. Available at `https://wiki.mozilla.org/Remote_Debugging_Protocol`.

[65] C Naumer, K Fisher, and Brenda Dervin. Sense-making: a methodological perspective. In *Sensemaking Workshop, CHI'08*, 2008.

[66] Stephen Oney and Joel Brandt. Codelets: linking interactive documentation and example code in the editor. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2697–2706. ACM, 2012.

[67] Stephen Oney and Brad Myers. Firecrystal: Understanding interactive behaviors in dynamic web pages. In *Visual Languages and Human-Centric Computing, 2009. VL/HCC 2009. IEEE Symposium on*, pages 105–108. IEEE, 2009.

[68] Fred G Paas. Training strategies for attaining transfer of problem-solving skill in statistics: A cognitive-load approach. *Journal of educational psychology*, 84(4):429, 1992.

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

[70] Peter Pirolli. Effects of examples and their explanations in a lesson n recursion: A production system analysis. *Cognition and Instruction*, 8(3):207–259, 1991.

[71] Peter Pirolli and Stuart Card. The sensemaking process and leverage points for analyst technology as identified through cognitive task analysis. In *Proceedings of international conference on intelligence analysis*, volume 5, pages 2–4. McLean, VA, USA, 2005.

[72] Chris Quintana, Brian J Reiser, Elizabeth A Davis, Joseph Krajcik, Eric Fretz, Ravit Golan Duncan, Eleni Kyza, Daniel Edelson, and Elliot Soloway. A scaffolding design framework for software to support science inquiry. *The journal of the learning sciences*, 13(3):337–386, 2004.

[73] Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from big code. In *ACM SIGPLAN Notices*, volume 50, pages 111–124. ACM, 2015.

[74] Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from big code. In *ACM SIGPLAN Notices*, volume 50, pages 111–124. ACM, 2015.

[75] Brian J Reiser, Iris Tabak, William A Sandoval, Brian K Smith, Franci Steinmuller, and Anthony J Leone. Bguile: Strategic and conceptual scaffolds for scientific inquiry in biology classrooms. *Cognition and instruction: Twenty-five years of progress*, pages 263–305, 2001.

[76] Daniel M Russell, Mark J Stefik, Peter Pirolli, and Stuart K Card. The cost structure of sensemaking. In *Proceedings of the INTERACT'93 and CHI'93 conference on Human factors in computing systems*, pages 269–276. ACM, 1993.

[77] Yasutaka Sakamoto, Shinsuke Matsumoto, Seiki Tokunaga, Sachio Saiki, and Masahide Nakamura. Empirical study on effects of script minification and http compression for traffic reduction. In *Digital Information, Networking, and Wireless*

Communications (DINWC), 2015 Third International Conference on, pages 127–132. IEEE, 2015.

[78] R Keith Sawyer. *The Cambridge handbook of the learning sciences*. Cambridge University Press, 2005.

[79] Daniel L Schwartz and John D Bransford. A time for telling. *Cognition and instruction*, 16(4):475–5223, 1998.

[80] Daniel L Schwartz, Xiaodong Lin, Sean Brophy, and John D Bransford. Toward the development of flexibly adaptive instructional designs. *Instructional-design theories and models: A new paradigm of instructional theory*, 2:183–213, 1999.

[81] David Williamson Shaffer and Mitchel Resnick. " thick" authenticity: New media and authentic learning. *Journal of interactive learning research*, 10(2):195–215, 1999.

[82] Remy Sharp. Js bin collaborative javascript debugging, 2016. Available at `https://jsbin.com/about`.

[83] Ben Shneiderman and Richard Mayer. Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Parallel Programming*, 8(3):219–238, 1979.

[84] Elliot Soloway. Learning to program= learning to construct mechanisms and explanations. *Communications of the ACM*, 29(9):850–858, 1986.

[85] Elliot Soloway and Kate Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on software engineering*, (5):595–609, 1984.

[86] Steve Souders. High-performance web sites. *Communications of the ACM*, 51(12):36–41, 2008.

[87] Margaret-Anne Storey. Theories, tools and research methods in program comprehension: past, present and future. *Software Quality Journal*, 14(3):187–208, 2006.

[88] Bipin Upadhyaya, Foutse Khomh, and Ying Zou. Extracting restful services from web applications. In *SOCA*, pages 1–4, 2012.

[89] Bret Victor. Learnable programming., 2012. Available at `http://worrydream.com/LearnableProgramming/`.

[90] Andrew Walenstein. Observing and measuring cognitive support: Steps toward systematic tool evaluation and engineering. In *Program Comprehension, 2003. 11th IEEE International Workshop on*, pages 185–194. IEEE, 2003.

[91] Karl E Weick. *Sensemaking in organizations*, volume 3. Sage, 1995.

[92] Mark Weiser and Joan Shertz. Programming problem representation in novice and expert programmers. *International Journal of Man-Machine Studies*, 19(4):391–398, 1983.

[93] Mike West. An introduction to content security policy, 2012. Available at `http://www.html5rocks.com/en/tutorials/security/content-security-policy/`.

[94] Sam Wineburg. Reading abraham lincoln: An expert/expert study in the interpretation of historical texts. *Cognitive Science*, 22(3):319–346, 1998.

[95] Samuel S Wineburg. Historical problem solving: A study of the cognitive processes used in the evaluation of documentary and pictorial evidence. *Journal of Educational Psychology*, 83(1):73, 1991.

APPENDIX A

Supplemental Figures

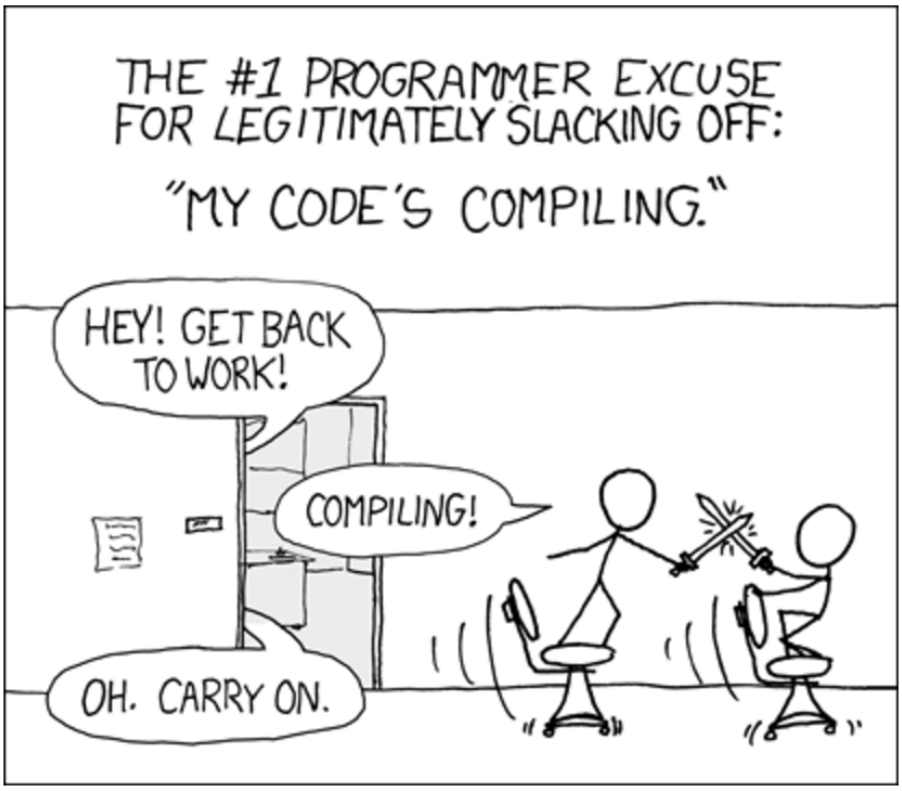

Figure A.1. "Compiling", XKCD, by Randall Munroe. xkcd.com/303