Isopleth: Supporting Sensemaking of Professional Web Applications to Create Readily Available Learning Experiences

JOSHUA HIBSCHMAN, DARREN GERGLE, ELEANOR O'ROURKE, and HAOQI ZHANG, Northwestern University

Online resources can help novice developers learn basic programming skills, but few resources support progressing from writing working code to learning professional web development practices. We address this gap by advancing Readily Available Learning Experiences, a conceptual approach for transforming all professional web applications into opportunities for authentic learning. This article presents Isopleth, a web-based platform that helps learners make sense of complex code constructs and hidden asynchronous relationships in professional web code. Isopleth embeds sensemaking scaffolds informed by the learning sciences to (1) expose hidden functional and event-driven relationships, (2) surface functionally related slices of code, and (3) support learners manipulating the provided code representations. To expose event-driven relationships, Isopleth implements a novel technique called Serialized Deanonymization to determine and visualize asynchronous functional relationships. To evaluate Isopleth, we conducted a case study across 12 professional websites and a user study with 14 junior and senior developers. Results show that Isopleth's sensemaking scaffolds helped to surface implementation approaches in event binding, web application design, and complex interactive features across a range of complex professional web applications. Moreover, Isopleth helped junior developers improve the accuracy of their conceptual models of how features are implemented by 31% on average.

CCS Concepts: • **Human-centered computing → Human computer interaction (HCI)**;

Additional Key Words and Phrases: Reverse engineering, developer tools, web inspection, JavaScript, authentic learning

ACM Reference format:
Joshua Hibschman, Darren Gergle, Eleanor O'Rourke, and Haoqi Zhang. 2019. Isopleth: Supporting Sensemaking of Professional Web Applications to Create Readily Available Learning Experiences. *ACM Trans. Comput.-Hum. Interact.* 26, 3, Article 16 (April 2019), 42 pages.
https://doi.org/10.1145/3310274

1 INTRODUCTION

Aspiring web developers are turning to online resources to teach themselves to code. Online learning platforms such as Codecademy, Khan Academy, and CodeSchool attract millions of learners and significantly increase the number of advanced beginners. However, these platforms primarily teach syntax or provide practice on constrained tutorial examples; they lack the

Funding for this research was provided in part by the National Science Foundation under Grant #1735977.
Authors' address: J. Hibschman, D. Gergle, E. O'Rourke, and H. Zhang, Northwestern University, Evanston, IL 60208; emails: jh@u.northwestern.edu, {dgergle, eorourke, hq}@northwestern.edu.

1073-0516/2019/04-ART16 $15.00
https://doi.org/10.1145/3310274

authenticity required to support the progression from writing functional code to writing professional-quality software. Aspiring web developers can learn to set up, read, and write basic JavaScript web applications, but they still lack conceptual knowledge of design patterns used in professional web solutions. As a result, significant gaps in knowledge and experience remain between advanced beginners and professional developers.

With few resources to support authentic learning of professional web development practices, we consider the use of professional web applications as a potential resource to support authentic learning. All professional web applications can be inspected using modern browser developer tools to reveal their underlying implementation. They offer rich details missing from training examples. They embed the programming concepts and implementation techniques that are used by professionals, and are continually updated as new solutions arise. But despite the abundant availability of front-end code, professional examples are complex and difficult for learners to understand.

We are interested in developing tools and approaches that can transform such examples into a learning resource that empowers aspiring web developers to learn professional practices and fill gaps in their knowledge. We envision the creation of *Readily Available Learning Experiences* (RALE) that 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. To accomplish these goals, our work on RALE seeks to bridge the knowledge gap between novice and expert web developers by creating software supports for self-directed learning from professional examples so that learners can leverage the richness and diversity of the web to support continual advancement of knowledge and skills. RALE has the potential to vastly increase the number of professional examples for learning; improve the breadth, depth, and quality of learners' conceptual understanding; and train more novice developers to tackle programming challenges in professional work.

As a first step toward RALE, this article focuses on addressing the challenge of learning programming concepts in professional web code. Specifically, we are interested in helping learners make sense of complex code constructs and hidden asynchronous relationships to understand how code components work together to implement features in professional web applications. For instance, a learner may be interested in understanding how Zillow's homepage search supports populating the user's previous home searches into its autocomplete search bar, or what code constructs were responsible for the hover effect upon mousing over buildings in National Geographic's New York Skyline article. Features in professional web applications such as these often contain many small components working together asynchronously, and learners lacking the conceptual knowledge required to make sense of the undocumented code may fail to see these relationships and connections. One challenge is the abundance of dynamic callbacks typical of front-end JavaScript implementations. Existing tools are limited in linking function invocations to their declaration context; learners cannot easily expose where functions are bound, passed, or set as callbacks. Another challenge is that connections among code components responsible for core aspects of functionality are often hidden deep in the code, e.g., in lower level functions and in library code. Hiding these details provides a limited view of how code components coordinate to achieve a feature, but revealing all relevant source code to include such details can easily overwhelm the learner given the large codebases typical of professional web applications.

To address these challenges, we introduce *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 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

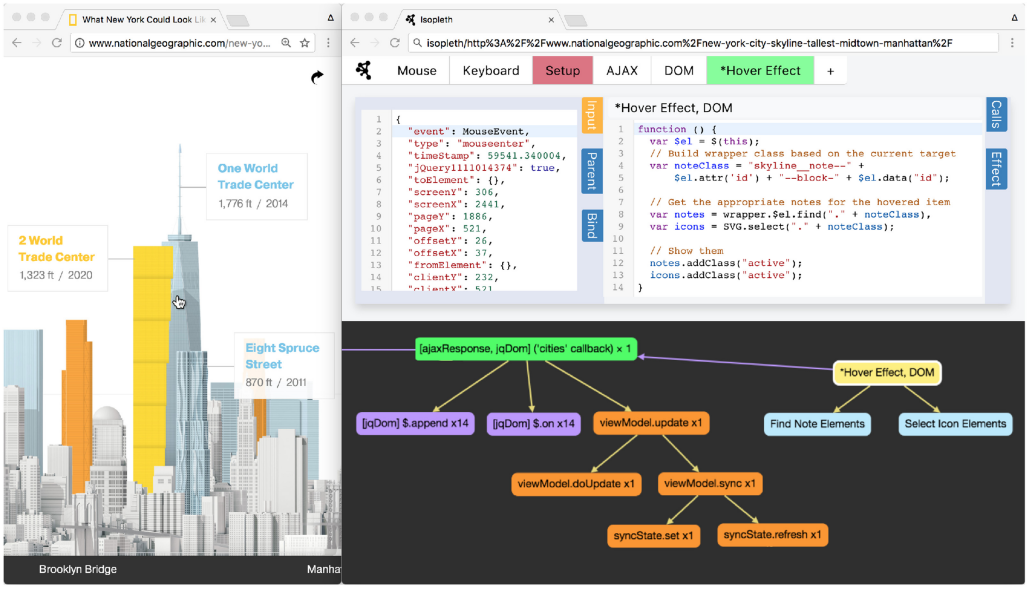

Fig. 1. Isopleth is a sensemaking platform that supports learners studying the call graphs of complex interactive websites. It provides automated and interactive supports for exposing hidden functional and asynchronous relationships, surfacing functionally related slices of code, and manipulating the provided code representation as learners make sense of complex JavaScript in professional websites. In this example, a learner is using Isopleth to understand the source code of the New York Skyline feature from National Geographic, which interactively displays information about buildings in the skyline as a user mouses over them (left). Using *facets* to hide setup code and to isolate code related to mouse hover events (top right), the learner examines the *condensed call graph* (bottom right) and finds an asynchronous binding (purple line) between the yellow selected node and a green AJAX callback node. By re-labeling nodes, examining *source frames* (middle right), and adding comments to the source code, Isopleth allows the learner to make sense of the code and uncover the elegant approach this web application uses to load data dynamically and bind hover events only once the data is loaded. The AJAX callback loads JSON data containing coordinate locations and metadata for each building in the skyline, and binds all the hover events to add "active" classes to display the relevant information.

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 this work 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 [10, 28, 29, 43] help to surface the most relevant code (e.g., Telescope [29], Unravel [28]), and provide methods for walking through code in execution order (e.g., Scry [10] and fireCrystal [43]). 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 article 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 [31]). While related toolkits can already fully instrument JavaScript code in professional web applications [24, 29, 35], 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 how web features are implemented. In addition to SD, the article 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, we introduce a facet detection approach that uses inputs and outputs to test for arguments or return values in function invocations. We further introduce techniques for bubbling up detected facets so that they can be properly visualized even when library internals are hidden from learners to reduce complexity.

To evaluate Isopleth, we conducted a case study across 12 popular professional websites and a user study with 14 developers (10 junior, 4 senior). Case study results illustrate how Isopleth's sensemaking scaffolds can help to surface implementation approaches in event bindings, web application design, and complex interactive features across a diverse set of complex professional web applications. Case study results further revealed how Isopleth can provide a broad understanding of the various ways in which the same features can be implemented across professional web applications. User study results show that Isopleth helped junior developers improve the accuracy of their conceptual models of how features on professional websites are implemented by 31% on average. Further, we found that both junior and senior developers leveraged Isopleth's sensemaking scaffolds to support a variety of program comprehension strategies, demonstrating Isopleth's flexibility in allowing users to approach sensemaking in intuitive ways. These study results provide exciting evidence for the effectiveness and potential of RALE for supporting learning from professional code.

2 RELATED WORK

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.

2.1 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 [6]. 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 [5]. Fast and Bernstein designed Meta, a Python language extension that allows programmers to share and compare their implementations of utility functions [19]. 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 [37] or reason about the structure of code [46]. 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 RALE 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 [27], Gidget [34], WebCrystal [11], Whyline [31], and Dinah [25] provide question–answer workflows to help learners resolve questions about a program's state or effects. Source visualization approaches such as Gliimpse [18], PyTutor [26], and Bret Victor's learnable programming [61] 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.*

2.2 Existing Tools for Learning from Complex, Professional Examples

Closest to our work, tools such as fireCrystal [43], Scry [10], Unravel [28], and Telescope [29] 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 lies beyond DOM-touching code in events, timing, data-retrieval, and data management code. 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 [29] 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 [29] 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 such as these 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 [10] 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. Last, 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 [10, 28, 29, 43] provide affordances to follow JavaScript and see its links to HTML modification, it remains difficult to see how dataflows through the functions and how functions relate to one another amid large numbers of function invocations.

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.

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 [63]. Early work on sensemaking from information science [17, 42, 47, 51] 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* [9, 45, 56, 58, 60, 62] 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, 12, 13, 16, 38, 40, 64]. 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 [32]. 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 [65, 66].

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 [48]. 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 [37, 48]. Finally, tools should provide opportunities for learners to inspect the underlying representation

in different ways [48]. Providing multiple ways to visualize the underlying data helps learners build dense, interconnected conceptual representations [3, 7, 53]. Through the design of Isopleth, we considered how guidelines for designing sensemaking scaffolds, originally designed to support sensemaking during scientific inquiry [48], 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 [9], (2) bottom-up from statements to abstractions [56], (3) beacons from familiar code with plan decomposition in unfamiliar code [58], and (4) bottom-up through control-flow abstraction from microstructures to macrostructures to form a situational model [45]. Isopleth supports learners adopting 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.

3.2 Design Arguments

In this work, we present a tool called Isopleth 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. [48] 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 [45, 58].

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.

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 [48]. Since expert practices rely on domain knowledge that learners lack, they need support in understanding, recognizing, and applying these practices during sensemaking [48, 50]. As a result, effective scaffolds organize information in disciplinary ways to help learners approach problems the way experts 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 [29], Unravel [28], fireCrystal [43], and Scry [10], 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 backend 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.

Characteristic 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 modifications 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 of 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.

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 [48]. 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 [28] and Telescope [29] 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 [43] and Scry [10] 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:

Characteristic 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 on 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 concepts like setup code and AJAX calls, and 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.

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 [48]. 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 [60]. Echoing Quintana et al., program comprehension researchers have also recommended that program comprehension tools should provide multiple views of the code [60].

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 build up in progressive layers and changes over time [56]. 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 [28], Telescope [29], fireCrystal [43], and Scry [10] 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:

Characteristic 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 [58] 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.

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 2, supports the following 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.

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 2. 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, 35]. 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.

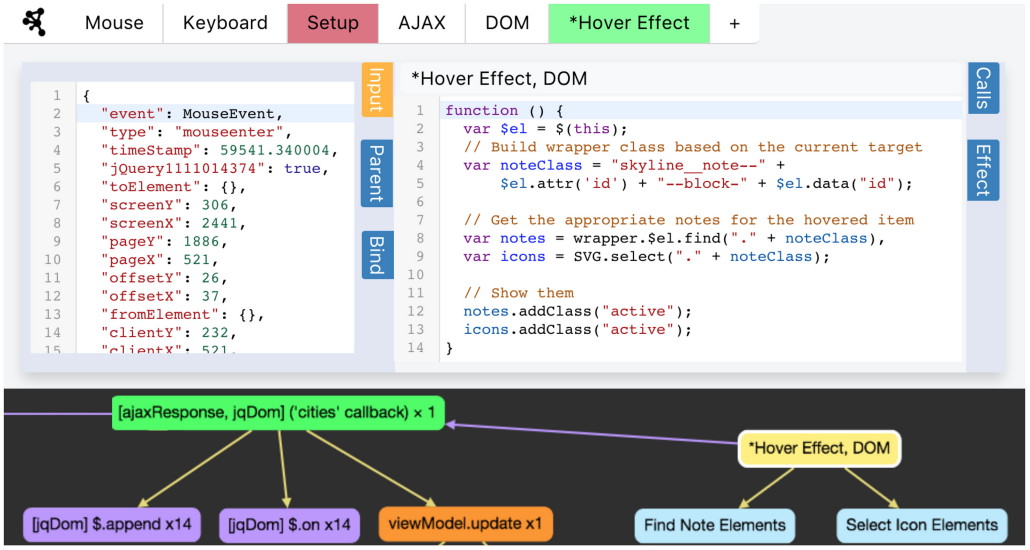

Fig. 2. A learner is using Isopleth to understand JavaScript code constructs related to moving and scrolling their mouse on National Geographic's New York Skyline article. Once activated, Isopleth opens in a new window and continuously updates with JavaScript activity. The condensed call graph (bottom) 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.

When the learner clicks on a node in the condensed call graph, Isopleth displays the function body in the source frame view (Figure 2, 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 the 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

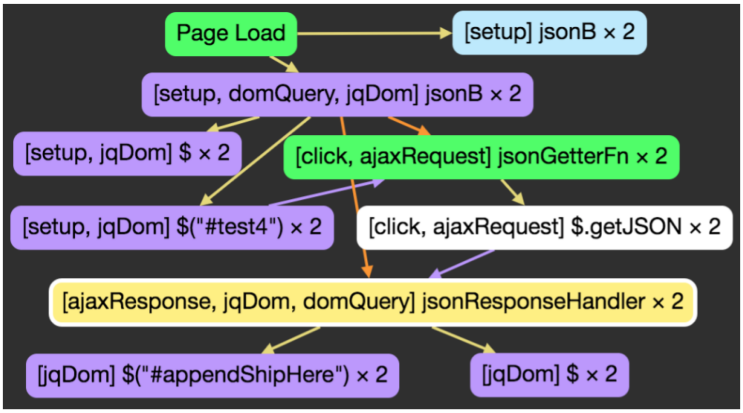

Fig. 3. A cluster of related collated function invocations (with their invoke-counts) in a condensed call graph, manually organized here for display. Nodes are colored green for top-level calls and yellow for currently selected; other nodes are colored based on the facets they match: purple for DOM, 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.

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.

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 2, 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 3 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 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.

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.

Isopleth also allows learners to create custom facet filters that help them explore functional slices beyond the set of default facets (see Figure 4). 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.

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

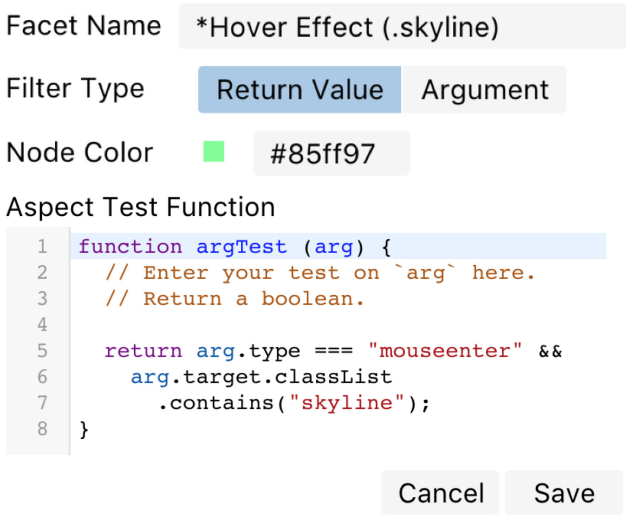

Fig. 4. A learner is creating a custom facet filter through the facet creator view. Facets are functional input–output schemas; creating a custom facet thus involves writing a test for arguments and return values to identify function invocation nodes that match such conditions on the argument or return 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 2).

the call graph for the learner. In this section, we present the technical methods that support these two core functionalities.

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 [24, 29, 35], 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 [24, 35], 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 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):

(1) Initiate public website instrumentation using the Sleight of Hand (SoH) technique [29] to instrument a website's source code.
(2) Extract the source for instrumentation via website-instrument-swap-and-trace (Wisat) architecture [29].

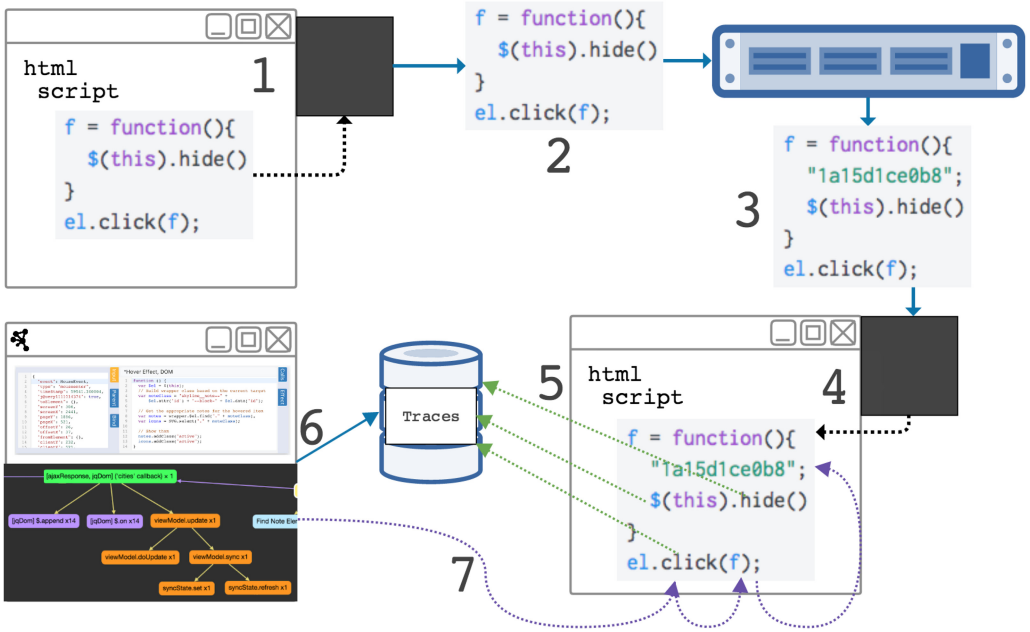

Fig. 5. The Serialized Deanonymization technique pictured above is a seven-step process for tracing an anonymous JavaScript function's path from creation to invocation. (1) Website JavaScript is extracted and (2) sent to an instrumentation server. (3) UUID's are injected into all function bodies. (4) 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.

(3) While applying Fondue tracer code [35] 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 3) by backtracing function invocations through the logs of arguments and return values from other function traces.

5.2 Identifying Facets with Facet Tree Decoration and Node Collation

With current tools [10, 35, 43], 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 [23] and minifiers swap variable names with short system-generated names that hold no semantic meaning [4, 52, 59].

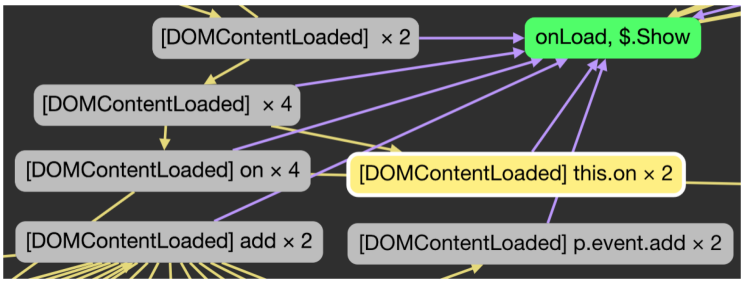

Fig. 6. This figure shows how Serialized Deanonymization allows for a DOM-modifying facets to be bubbled up out of a library call. After removing library code filtering from the condensed call graph, we can see DOM-modifying functions existing inside the library (grey nodes; and the currently selected 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).

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 [30] 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 facet. Figures 6 and 7 provide two examples that show how facets are bubbled out of library code through asynchronous links (using SD) and through function invocations, respectively.

5.3 Implementation

Activating Isopleth follows the same workflow as Telescope [29]. 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 [29] to instrument websites and extends Fondue [35] by adding unique identifiers through the SD technique.[1]

[1]Github: Isopleth's Fondue API https://github.com/NUDelta/Isopleth/tree/master/fondue-api.

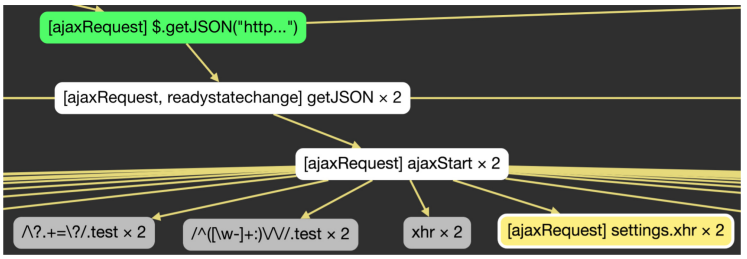

Fig. 7. This figure demonstrates how facets are bubbled out of library code through function invocations. During call graph calculation, if a facet is detected in a library, we bubble the facet up to the first occurrence of non-library code to help learners identify the facet roles of library API calls. 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).

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.[2] 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. 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.[3]

5.3.1 Technical Limitations. By extending Fondue [35] and the Wisat architecture [29] for instrumenting source code on public websites, Isopleth and its SD 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, and Flash are not captured. However, JavaScript calls to these APIs are captured and are surfaced to support sensemaking.

Isopleth's SD 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.,

[2]Github: Isopleth's CSP Modifier https://github.com/NUDelta/Isopleth/blob/master/chrome-extension/background.js.
[3]Github: UglifyJS beautifier https://github.com/mishoo/UglifyJS2.

`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 [49], 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 [36] addresses this limitation by providing a visual web inspector that supports novices learning professional web page features in CSS.

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 8. 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

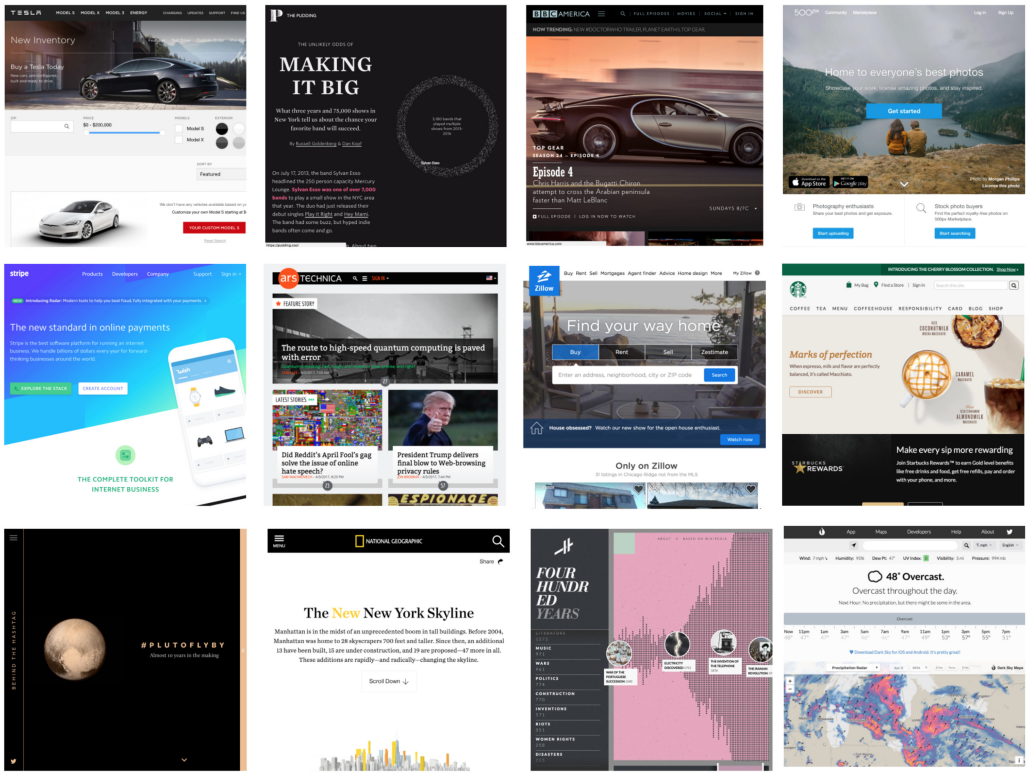

Fig. 8. We studied Isopleth's ability to support sensemaking and elicit design patterns across 12 websites selected from a diversity of industries based on Alexa popularity rankings, the Webby awards, and personal interest. From top left to bottom right: Tesla, The Pudding's "Making it Big," BBC America, 500px, Stripe, ArsTechnica, Zillow, Starbucks, HashTagsUnplugged's "#PlutoFlyBy" article, National Geographic's "*New* New York Skyline" article, Histography.io, and DarkSky.net.

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 as follows:

- *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).
- *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?

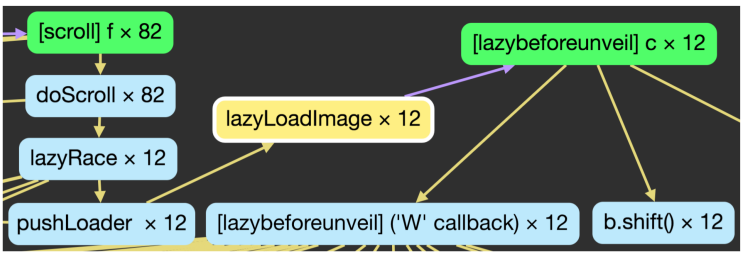

Fig. 9. Isopleth's condensed call graph representation of BBC America's lazy-image-loading strategy. By reading predefined node labels and following functional (yellow lines) and asynchronous links (purple lines), we see that the scroll event (top-left node) was passed to an event handler responsible 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.

—**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 front-end 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.

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.

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.

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 9. 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

Fig. 10. 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.

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`.

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.

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 the search box queried the user's `localStorage` for recent user search queries, thereby revealing an unexpected lightweight per-user caching strategy (see Figure 10).

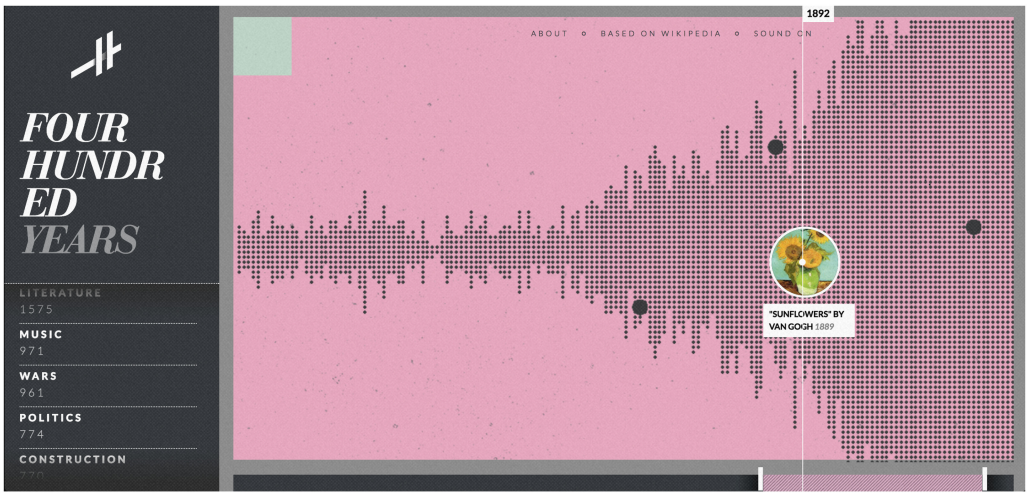

Fig. 11. The most complex UI we tested was histography.io, which triggers thousands of function invocations in response to mouse movements. On hover, historical events on a timeline bubble up with randomly decaying dots.

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.

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 11). 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 labeling 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.

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.

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.

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.

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.

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. Three 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 four 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.

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?

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 10 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 3 years of professional experience. The 10 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 5 minutes interacting with the feature on average. After interacting with the website, we asked each participant to spend 5 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 10 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 5 years of professional web development experience and produced the ground truth diagrams through a deep review of

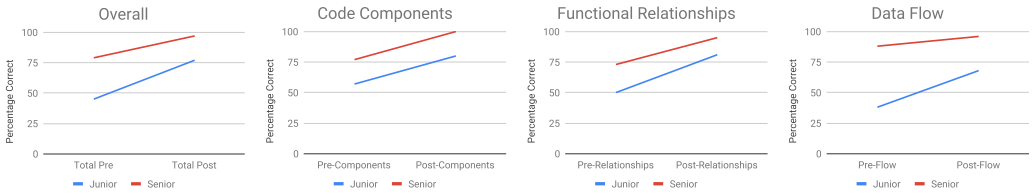

Fig. 12. Graphs show the change in the accuracy of junior and senior developers' conceptual models before and after using Isopleth. The *Overall* graph shows the average change in accuracy, while the *Code Components*, *Functional Relationships*, and *Dataflow* graphs show the changes in accuracy for 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.

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 dataflow 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.

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 12, participants more accurately described code components, functional relationships, and dataflow 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 10 junior developers, 4 rejected their pre-test model, 5 changed their model, and 1 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 dataflow attributes each participant successfully identified. As shown in Figure 12, 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 dataflow 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. Again, the improvements were distributed across components, relationships, and dataflow attributes, as shown in Figure 12. 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.

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 one junior developer drew for the XKCD click-and-drag map, shown in Figure 13. 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.

7.2.2 Changed Model. Five junior developers changed their original conceptual models after using Isopleth. In general, these developers described their hypotheses about 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 14, 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),

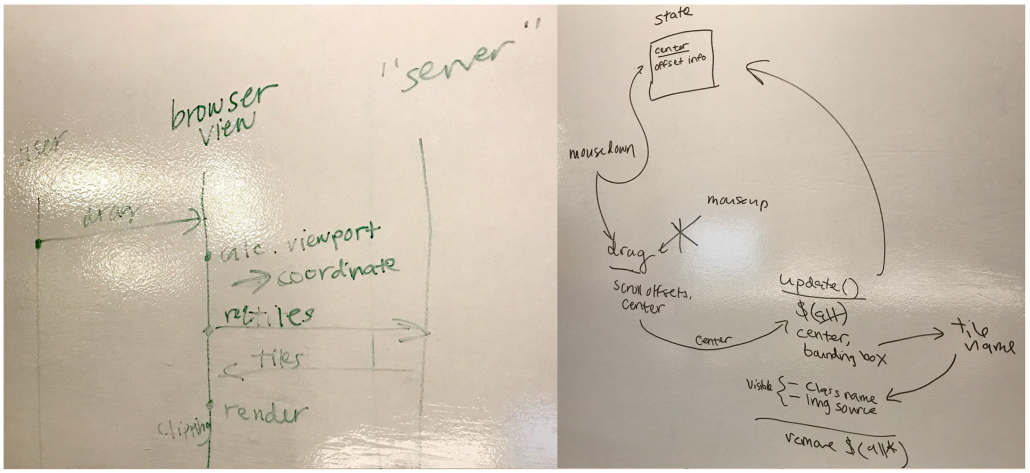

Fig. 13. A junior developer diagrams their conceptual model before (left) and after (right) using Isopleth. The developer rejected their conceptual model in favor of a new model formed using Isopleth. Before using Isopleth, the developer thought that XKCD tracks a drag, calculates the viewport change, 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.

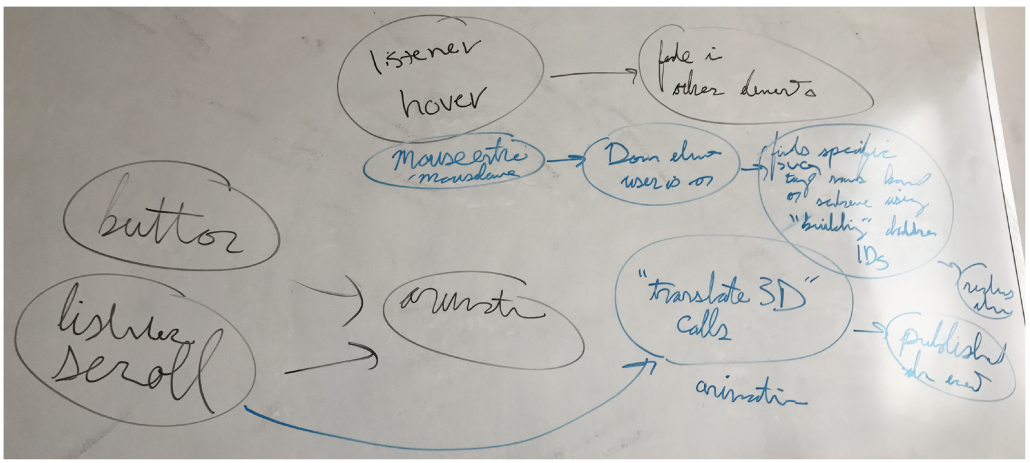

Fig. 14. A junior developer diagrams their conceptual models before (black) and after (blue) using Isopleth. The developer changed their conceptual model by exchanging some components and relational attributes with more accurate representations. Before Isopleth, the developer thought that the New York 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.

but was able to improve their conceptual model by exchanging some components and relational attributes with more accurate ones after using Isopleth.

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

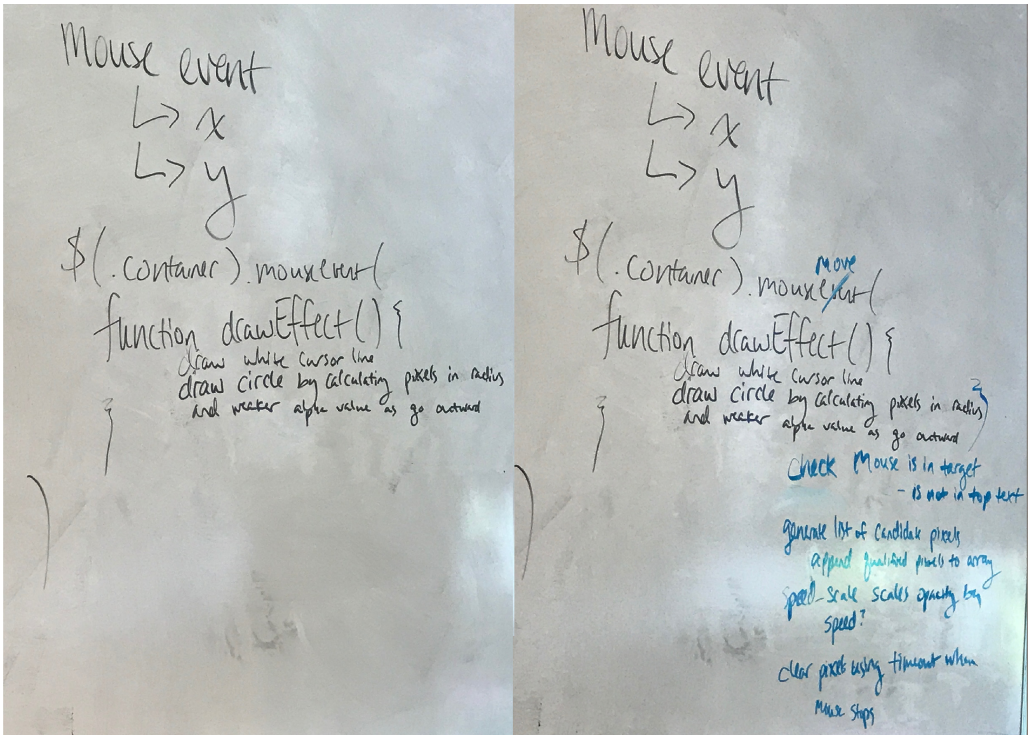

Fig. 15. A senior developer diagrams their conceptual models before (left) and after (right) using Isopleth. The developer validated their jQuery-style pseudocode model and added specific details about Histography.io's cursor movement found in Isopleth's source frame views. Prior to Isopleth, they described 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 dataflow and added clever implementation details to their prior model.

of the logical components, how they coordinated, and how the dataflow 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 15. 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 dataflow, and added implementation details that were missing from their original model.

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.

Feature	Junior	Senior	Total
Call Graph	10	4	14
Click Node	10	4	14
Drag Node	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	10	4	14
Read Source Code	10	4	14
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

Fig. 16. The number of junior, senior, and total developers that used different Isopleth features to make sense of the provided professional code examples.

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 16 summarizes feature usage by junior and senior developers.

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 16, all 14 participants interacted with both the call graph and the source frames, and 9 changed node labels and edited source code during their sensemaking process.

All 14 participants began their sensemaking process by exploring the condensed call graph and looking for nodes of interest, as 1 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 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 Isopleth's 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.

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 16, 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 juniors 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 had the background knowledge required to take advantage of the predefined and custom facet filters to explore hypotheses.

The three other developers (all juniors) 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 juniors, two seniors) 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.

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.

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.

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 14). 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 dataflow showing which building to animate.

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.

8 DISCUSSION

Our work advances the creation of RALE that transform professional web applications into opportunities for authentic learning. As a first step toward realizing RALE, this article 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 dataflow. 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 9 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 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.

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.

8.2 Future Work on RALE

Our vision for 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 [36], 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 [57]. 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 [7, 14]; and handle routine tasks to reduce cognitive load and disruption [41].

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 [22]. 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 [15]. 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 [37] to guide learners through analogical reasoning [20, 21, 33, 39]; help learners use contrasting and boundary cases to refine conceptual understanding [8, 54, 55]; and highlight deep features to overcome transfer issues [44]. 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.

ACKNOWLEDGMENTS

We thank anonymous reviewers, Rob Miller, Sarah Lim, and members of the Design, Technology, and Research program and the Delta Lab for their insightful feedback and helpful discussions.

REFERENCES

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

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

[3] Susan A. Ambrose, Michael W. Bridges, Michele DiPietro, Marsha C. Lovett, and Marie K. Norman. 2010. *How Learning Works: Seven Research-Based Principles for Smart Teaching*. John Wiley & Sons.

[4] Michael Bolin. 2010. *Closure: The Definitive Guide*. O'Reilly Media, Inc., 333–357.

[5] Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R. Klemmer. 2010. Example-centric programming: Integrating web search into the development environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 513–522.
[6] Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. 2009. Two studies of opportunistic programming: Interleaving web foraging, learning, and writing code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 1589–1598.
[7] John D. Bransford, Ann L. Brown, and Rodney R. Cocking. 2000. *How People Learn*, Vol. 11. National Academy Press, Washington, DC.
[8] John D. Bransford, Jeffery J. Franks, Nancy J. Vye, and Robert D. Sherwood. 1989. New approaches to instruction: Because wisdom can't be told. *Similarity and Analogical Reasoning* 470 (1989), 497.
[9] Ruven Brooks. 1983. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies* 18, 6 (1983), 543–554.
[10] Brian Burg, Andrew J. Ko, and Michael D. Ernst. 2015. Explaining visual changes in web interfaces. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*. ACM, 259–268.
[11] Kerry Shih-Ping Chang and Brad A. Myers. 2012. WebCrystal: Understanding and reusing examples in web authoring. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 3205–3214.
[12] William G. Chase and Herbert A. Simon. 1973. Perception in chess. *Cognitive Psychology* 4, 1 (1973), 55–81.
[13] Michelene T. H. Chi, Paul J. Feltovich, and Robert Glaser. 1981. Categorization and representation of physics problems by experts and novices. *Cognitive Science* 5, 2 (1981), 121–152.
[14] Allan Collins. 1996. Design issues for learning environments. *International Perspectives on the Design of Technology-Supported Learning Environments* (1996), 347–361.
[15] Elizabeth A. Davis and Marcia C. Linn. 2000. Scaffolding students' knowledge integration: Prompts for reflection in KIE. *International Journal of Science Education* 22, 8 (2000), 819–837.
[16] Adriaan D. De Groot. 1978. *Thought and Choice in Chess*, Vol. 4. Walter de Gruyter GmbH & Co KG.
[17] Brenda Dervin and Patricia Dewdney. 1986. Neutral questioning: A new approach to the reference interview. *RQ* (1986), 506–513.
[18] Pierre Dragicevic, Stéphane Huot, and Fanny Chevalier. 2011. 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*. ACM, 257–262.
[19] Ethan Fast and Michael S. Bernstein. 2016. Meta: Enabling programming languages to learn from the crowd. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*. ACM, 259–270.
[20] Dedre Gentner. 1983. Structure-mapping: A theoretical framework for analogy. *Cognitive Science* 7, 2 (1983), 155–170.
[21] Dedre Gentner, Jeffrey Loewenstein, and Leigh Thompson. 2003. Learning and transfer: A general role for analogical encoding. *Journal of Educational Psychology* 95, 2 (2003), 393.
[22] Mary L. Gick and Keith J. Holyoak. 1983. Schema induction and analogical transfer. *Cognitive Psychology* 15, 1 (1983), 1–38.
[23] Elena L. Glassman, Lyla Fischer, Jeremy Scott, and Robert C. Miller. 2015. Foobaz: Variable name feedback for student code at scale. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*. ACM, 609–617.
[24] Google. 2017. Enabling the Async Callstack. Retrieved from https://developers.google.com/web/tools/chrome-devtools/javascript/step-code#enable_the_async_call_stack.
[25] Paul Gross, Jennifer Yang, and Caitlin Kelleher. 2011. 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*. ACM, 3397–3400.
[26] Philip J. Guo. 2013. Online python tutor: Embeddable web-based program visualization for cs education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*. ACM, 579–584.
[27] Andrew Head, Codanda Appachu, Marti A. Hearst, and Bjorn Hartmann. 2015. Tutorons: Generating context-relevant, on-demand explanations and demonstrations of online code. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*. IEEE, 3–12.
[28] Joshua Hibschman and Haoqi Zhang. 2015. 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*. ACM, 270–279.
[29] Joshua Hibschman and Haoqi Zhang. 2016. Telescope: Fine-tuned discovery of interactive web UI feature implementation. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*. ACM, 233–245.
[30] jQuery. 2017. Current Active Browser Support. Retrieved from https://jquery.com/browser-support/.
[31] Andrew J. Ko and Brad A. Myers. 2004. Designing the whyline: A debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 151–158.

[32] Andrew Jensen Ko, Brad A. Myers, and Htet Htet Aung. 2004. Six learning barriers in end-user programming systems. In *Procedings of the IEEE Symposium on Visual Languages and Human Centric Computing*. IEEE, 199–206.
[33] Janet L. Kolodner. 1997. Educational implications of analogy: A view from case-based reasoning. *American Psychologist* 52, 1 (1997), 57.
[34] Michael J. Lee and Andrew J. Ko. 2011. Personifying programming tool feedback improves novice programmers' learning. In *Proceedings of the Seventh International Workshop on Computing Education Research*. ACM, 109–116.
[35] Tom Lieber, Joel R. Brandt, and Rob C. Miller. 2014. Addressing misconceptions about code with always-on programming visualizations. In *Proceedings of the 32nd Annual ACM Conference on Human Factors in Computing Systems*. ACM, 2481–2490.
[36] Sarah Lim, Josh Hibschman, Haoqi Zhang, and Nell O'Rourke. 2018. Ply: A visual web inspector for learning from professional webpages. In *Proceedings of the 31st Annual Symposium on User Interface Software and Technology*. ACM.
[37] Marcia C. Linn. 1995. Designing computer learning environments for engineering and computer science: The scaffolded knowledge integration framework. *Journal of Science Education and Technology* 4, 2 (1995), 103–126.
[38] Marcia C. Linn and Michael J. Clancy. 1992. The case for case studies of programming problems. *Communications of the ACM* 35, 3 (1992), 121–132.
[39] Mark A. McDaniel and Carol M. Donnelly. 1996. Learning with analogy and elaborative interrogation. *Journal of Educational Psychology* 88, 3 (1996), 508.
[40] Tanya J. McGill and Simone E. Volet. 1997. A conceptual framework for analyzing students' knowledge of programming. *Journal of Research on Computing in Education* 29, 3 (1997), 276–297.
[41] Yoshiro Miyata and Donald A. Norman. 1986. Psychological issues in support of multiple activities. *User Centered System Design: New Perspectives on Human-computer Interaction* (1986), 265–284.
[42] C Naumer, K. Fisher, and Brenda Dervin. 2008. Sense-making: A methodological perspective. In *Proceedings of the Sensemaking Workshop*.
[43] Stephen Oney and Brad Myers. 2009. FireCrystal: Understanding interactive behaviors in dynamic web pages. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*. IEEE, 105–108.
[44] Fred G. Paas. 1992. Training strategies for attaining transfer of problem-solving skill in statistics: A cognitive-load approach. *Journal of Educational Psychology* 84, 4 (1992), 429.
[45] Nancy Pennington. 1987. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology* 19, 3 (1987), 295–341.
[46] Peter Pirolli. 1991. Effects of examples and their explanations in a lesson n recursion: A production system analysis. *Cognition and Instruction* 8, 3 (1991), 207–259.
[47] Peter Pirolli and Stuart Card. 2005. The sensemaking process and leverage points for analyst technology as identified through cognitive task analysis. In *Proceedings of International Conference on Intelligence Analysis*, Vol. 5. McLean, VA, USA, 2–4.
[48] Chris Quintana, Brian J. Reiser, Elizabeth A. Davis, Joseph Krajcik, Eric Fretz, Ravit Golan Duncan, Eleni Kyza, Daniel Edelson, and Elliot Soloway. 2004. A scaffolding design framework for software to support science inquiry. *The Journal of the Learning Sciences* 13, 3 (2004), 337–386.
[49] Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting program properties from big code. *ACM SIGPLAN Notices* 50, 1 (2015), 111–124.
[50] Brian J. Reiser, Iris Tabak, William A. Sandoval, Brian K. Smith, Franci Steinmuller, and Anthony J. Leone. 2001. BGuILE: Strategic and conceptual scaffolds for scientific inquiry in biology classrooms. *Cognition and Instruction: Twenty-Five Years of Progress* (2001), 263–305.
[51] Daniel M. Russell, Mark J. Stefik, Peter Pirolli, and Stuart K. Card. 1993. The cost structure of sensemaking. In *Proceedings of the INTERACT'93 and CHI'93 Conference on Human Factors in Computing Systems*. ACM, 269–276.
[52] Yasutaka Sakamoto, Shinsuke Matsumoto, Seiki Tokunaga, Sachio Saiki, and Masahide Nakamura. 2015. Empirical study on effects of script minification and HTTP compression for traffic reduction. In *Proceedings of the 3rd International Conference on Digital Information, Networking, and Wireless Communications*. IEEE, 127–132.
[53] R. Keith Sawyer. 2005. *The Cambridge Handbook of the Learning Sciences*. Cambridge University Press.
[54] Daniel L. Schwartz and John D. Bransford. 1998. A time for telling. *Cognition and Instruction* 16, 4 (1998), 475–5223.
[55] Daniel L. Schwartz, Xiaodong Lin, Sean Brophy, and John D. Bransford. 1999. Toward the development of flexibly adaptive instructional designs. *Instructional-Design Theories and Models: A New Paradigm of Instructional Theory* 2 (1999), 183–213.
[56] Ben Shneiderman and Richard Mayer. 1979. Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Parallel Programming* 8, 3 (1979), 219–238.
[57] Elliot Soloway. 1986. Learning to program= learning to construct mechanisms and explanations. *Communications of the ACM* 29, 9 (1986), 850–858.

[58] Elliot Soloway and Kate Ehrlich. 1984. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering* 5 (1984), 595–609.
[59] Steve Souders. 2008. High-performance web sites. *Communications of the ACM* 51, 12 (2008), 36–41.
[60] Margaret-Anne Storey. 2006. Theories, tools and research methods in program comprehension: Past, present and future. *Software Quality Journal* 14, 3 (2006), 187–208.
[61] Bret Victor. 2012. Learnable Programming. Retrieved from http://worrydream.com/LearnableProgramming/.
[62] Andrew Walenstein. 2003. Observing and measuring cognitive support: Steps toward systematic tool evaluation and engineering. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension*. IEEE, 185–194.
[63] Karl E. Weick. 1995. *Sensemaking in Organizations*, Vol. 3. Sage.
[64] Mark Weiser and Joan Shertz. 1983. Programming problem representation in novice and expert programmers. *International Journal of Man-Machine Studies* 19, 4 (1983), 391–398.
[65] Sam Wineburg. 1998. Reading Abraham Lincoln: An expert/expert study in the interpretation of historical texts. *Cognitive Science* 22, 3 (1998), 319–346.
[66] Samuel S. Wineburg. 1991. Historical problem solving: A study of the cognitive processes used in the evaluation of documentary and pictorial evidence. *Journal of Educational Psychology* 83, 1 (1991), 73.

Received August 2018; accepted December 2018