Visual Web Inspection for Complex Professional Examples

Sarah Lim
Northwestern University
slim@u.northwestern.edu

INTRODUCTION

Novice programmers rely on online resources for learning to code [1], particularly in the domain of web development [5]. While platforms such as Codeacademy and TutsPlus teach syntax and constrained tutorial examples, they do not teach the authentic design and problem-solving practices necessary to produce professional-grade output [16, 18]. Professional webpages rely on complex Document Object Model (DOM) structures and Cascading Style Sheets (CSS) to create rich user interfaces. These webpages can serve as design inspiration [12] and demonstrate modern implementation best practices. Moreover, since existing webpages are freely inspectable, they form a compelling potential body of examples for aspiring professional developers.

Despite their availability and authenticity, however, most professional examples are unusable as learning materials. Inspectors such as the Chrome DevTools (CDT) display hundreds of unfamiliar HTML elements and CSS properties at once, obscuring relevant code and overwhelming developers. Information overload is particularly detrimental to novice programmers, who reason about examples visually but struggle to locate and understand the lines of code responsible for a particular visual outcome [7, 2]. Our needfinding observations confirm this result, while highlighting two domain-specific obstacles to web inspection: (1) *ineffective properties*, which appear relevant in the inspector but have no effect on the webpage when disabled (Figure 1), and (2) learners' *missing conceptual knowledge* of core language semantics, which impedes understanding even after relevant code has been located.

To support authentic learning, inspection tools must reconcile two incongruous paradigms: the browser engine, which adheres to a precise technical specification but has no concept of relevance beyond runtime coverage; and the developer's visual intuition, which often supplants their understanding of language semantics. We bridge this gap with Ply, a web inspector which *embeds visual awareness* to surface the precise code and concepts responsible for a feature of interest to a learner. Ply implements a novel technique called *visual regression pruning* (VRP), which hides ineffective CSS properties by checking for visible *regressions* to a webpage following each property's removal. Our key technical insight is that VRP can be extended to not only remove irrelevant code, but also to compute dependencies between lines of code. Surfacing these dependencies through contextual hints can help even highly inexperienced developers make sense of unfamiliar professional examples, since all inferences made by the system correspond to observable visual effects.

In addition to multiple iterations of prototype testing, we conducted two summative evaluations of Ply's core features. In a between-subjects study ($n = 12$), pruning ineffective properties helped developers replicate a complex professional example feature more quickly. Novices in a second qualitative study ($n = 6$) used Ply to recognize unfamiliar design patterns and dependencies from professional examples, and successfully generalized

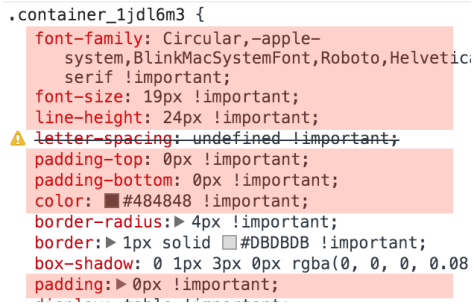

Figure 1. Ineffective properties (highlighted in red) appear active in CDT, but have no visible effect on the webpage when disabled. This example is taken from the highest-precedence CSS rule in the cascade.

these approaches beyond the original context. These results provide preliminary evidence for the viability of professional examples as authentic learning materials.

RELATED WORK AND UNIQUENESS OF APPROACH

Interactive systems can support designers and developers in understanding, adapting, and combining [1, 12, 11] example programs. In particular, Gross and Kelleher provide design principles to help novices leverage examples effectively during feature location tasks [7]. Most of these systems target *end-user programmers* (EUPs), who are uninterested in learning more programming than necessary to achieve their goals. While source reduction may help EUPs replicate designs more quickly, Ply targets developers actively invested in learning new concepts, particularly those with professional aspirations.

Examples have been shown to help novice programmers form richer conceptual schemas of underlying programming principles [17], but prior work has largely neglected *professional* examples because they impose additional cognitive load that distracts from learning outcomes. In particular, the complexity of professional source code presents additional design challenges. A number of systems provide visual affordances to help developers identify a subset of code responsible for interactive behaviors of interest. Rehearse [2] highlights relevant lines of code during program execution, then locates related lines based on API definitions. Scry [3] and FireCrystal [13] link DOM and CSS modifications to responsible lines of JavaScript code, and provide affordances for visualizing the resulting interaction timelines. Telescope [8] computes similar linkages across HTML and JavaScript source, but augments the source code view directly. Ply extends feature location to static web design, focusing on the unique challenges of reasoning about CSS — a language notorious for its dearth of automated tooling and complex semantics [6, 14]. Systems such as WebCrystal [4] sidestep this complexity by reducing the cascade of authored styles to the exact *used values* calculated by the browser engine. For instance, the browser engine may resolve `width: 33%;` to a used value of `width: 45.66667px;`. Unfortunately, this approach sacrifices authenticity — a developer is unlikely to ever

set an element's `width` to `45.66667px` in practice. To make authentic learning a first-class goal, Ply must reduce complexity while preserving the richness unique to professional examples.

DESIGN PROCESS

We conducted needfinding interviews with 20 student web developers and observed ten of these developers attempt to locate and replicate features from complex webpages using Chrome Developer Tools (CDT).[1] Only two developers made any meaningful progress within the first 30 minutes. All developers struggled to navigate hundreds of DOM nodes and a staggering cascade of cross-browser resets, vendor prefixes, global style guides, and mobile-first `@media` queries. Two recurring obstacles compounded this information overload: *ineffective properties* and *missing conceptual knowledge*.

Ineffective properties

Developers wasted the overwhelming majority of their time and energy on *ineffective properties*, which had no visible effect on the webpage despite appearing active in CDT (Figure 1). These properties occurred in every example we explored. Complex interfaces require a large number of styles to ensure consistency across all possible conditions, and it is often easier for authors to declare redundant properties "just in case" rather than risk failure. To help developers locate relevant styles, CDT provides a modicum of visual guidance: CSS rules are listed in descending order of precedence, computed by the browser engine based on static properties such as selector *specificity* and source location. If a property is overloaded by a higher-precedence declaration, CDT denotes the property's inactivity with a strikethrough. Unfortunately, not all active properties are visually effective with respect to the current webpage. These false leads were a major source of frustration for developers and impeded feature replication and understanding, since even "relevant" code included extraneous properties with no intuitive explanation.

Missing conceptual knowledge

Even after locating a relevant set of styles, developers struggled to understand how multiple properties spread across different rules coordinated to produce a single stylistic outcome. Drawing relationships between program fragments is a well-documented learning barrier for novice programmers [7, 2, 10]. While many of the examples exhibited best practices in CSS organization and composition, developers failed to notice these patterns due to their own misunderstandings of CSS overloading behavior and property side effects.

Corroborating findings in [7], developers relied on visual descriptors and deictic references ("I want it to look more *like this*") as a substitute for understanding the formal semantics of HTML and CSS. They articulated goals and explained code in terms of visual effects rather than language constructs, and used the word "thing" to describe selectors, rules, and properties, often within the same sentence. While visual intuition provided an accessible starting point, developers froze when confronted with unexpected behaviors that did not readily admit a visual explanation. Sparse domain vocabularies prevented developers from formulating effective search queries, perpetuating the cycle of misunderstanding.

[1]For the rest of this paper, we use "CDT" as a metonym for state-of-the-art inspection tools available to practitioners.

Final design guidelines

To support novice developers during authentic inspection tasks, we build on Quintana et al. [15]'s design guidelines for scientific *sense-making* tools. In the learning sciences literature, sense-making is defined as an iterative process in which learners reason about a phenomenon, test their conjecture empirically, and refine their understanding based on the results [9, 15]. Feature replication is similar: a developer applies their existing knowledge to conjecture how a feature might be implemented, searches the DOM and style cascade for potentially relevant code, and validates their guess by transferring the code to their own editor. Unexpected results at this stage often reveal the developer's misconceptions about CSS.

Based on our needfinding observations, we propose the following design guidelines: (1) *Hide visually-irrelevant code* to set boundaries for feature location tasks and minimize unnecessary cognitive burden, and (2) *Embed contextual guidance* to explain relationships between lines and blocks of code using observable effects. These guidelines reflect Quintana et al.'s suggestion to use representations and language that bridge learners' understanding [15]. Since novice developers rely on visual intuition in lieu of robust semantic understanding, automated tools should rely on visual criteria when possible.

SYSTEM DESCRIPTION

We illustrate these guidelines with Ply (Figure 2), a semantic web inspector designed to support novice developers in replicating and making sense of professional examples. Ply loosely models the inspection interface found in CDT and similar tools, but uses *visual regression pruning* to hide ineffective properties and surface dependencies.

Inspecting an element

Consider a developer interested in replicating the login buttons (Figure 2a) on the Tumblr homepage. After activating Ply's browser extension and hovering over the button of interest, Ply loads the element and its subtree, isolated from the rest of the DOM (Figure 2b). Hovering over a node highlights the corresponding element on the page, and clicking an node displays the corresponding cascade of matched CSS rules. After clicking "Prune" in the toolbar (Figure 2c), visually-ineffective properties are crossed out and displayed in greyscale (Figure 2d).Unlike CDT, Ply guarantees that each remaining property has some effect on the visible portion of the webpage; clicking a property toggles it on and off, allowing the developer to observe the result.

Surfacing overloaded base styles

To highlight the design patterns used in professional webpages, Ply annotates CSS with highlights and tooltip hints. As one example, production webpages often include style guides for visual consistency. A common organizational approach involves declaring a CSS rule corresponding to a component's *base style*, then defining a visual variant by declaring a second rule which selectively overloads properties from the base style (cf. object-oriented *method overriding*). Ply identifies base styles (Figure 2e) automatically during pruning, displaying a "Likely base style" hint next to the relevant CSS rule and providing additional explanation when the user mouses over the hint. Overloaded properties appear highlighted in yellow (Figure 2f), and mousing over these highlights displays a second tooltip explaining that the property is a default value within the base style. These annotations restate intuitive visual relationships in terms of cascading

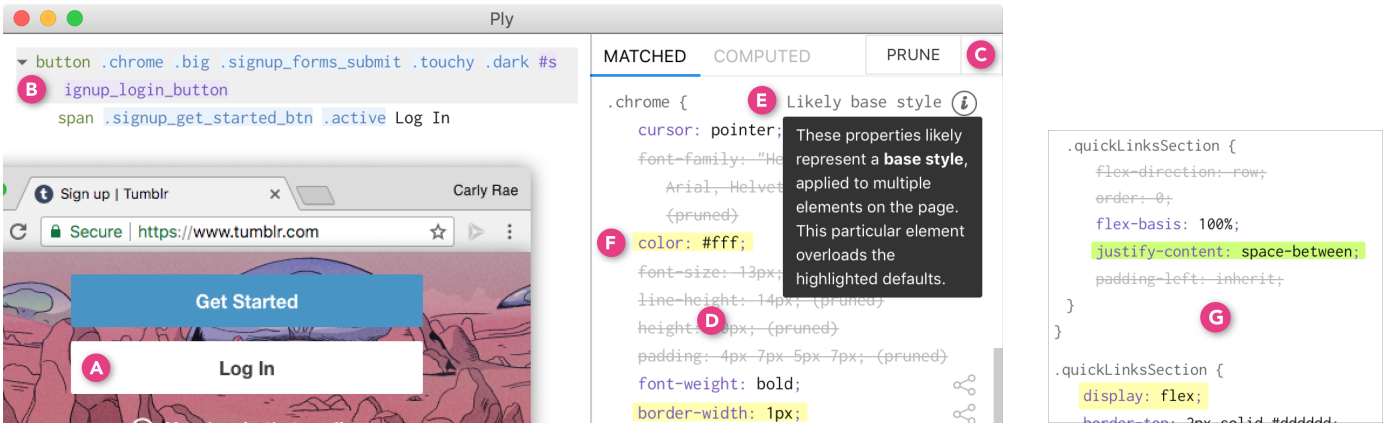

Figure 2. Left: Ply's DOM and CSS inspection interface is designed to minimize visually irrelevant information during inspection tasks. Right: Ply's implicit dependency overlay reveals that `justify-content` as a dependent of `display: flex;`.

and overloading terminology, bridging the gap from intuition to domain formalisms.

Surfacing implicit dependencies

Ply is the first web inspector capable of surfacing *implicit dependencies* between CSS properties. An implicit dependency occurs when a property declaration depends on another declaration to take effect. For instance, the declaration `vertical-align: middle;` depends on `display: table-cell;` being set on the same element, a common source of grief for novice CSS developers. These dependencies are well-defined in the specification but not centrally documented, and impossible to statically infer. As a result, most developers spend years tediously learning implicit dependencies through trial and error.

Figure 2g shows Ply's interface for surfacing implicit dependencies. A developer who is vaguely familiar with CSS Flexbox might guess that `display: flex;` turns its element into a flex container, and that `flex-basis` is somehow related. When the developer hovers over a property and clicks the "Show dependents" icon, Ply traverses the cascade and identifies other properties which depend on the selected property to be visually effective. Here, Ply highlights `display: flex;` in yellow, and its dependent `justify-content` in green. Hovering over `justify-content` displays a tooltip explaining that the property *implicitly depends* on `display: flex;` to be effective. Notably, Ply reveals that `flex-basis` is *not* a dependent of `display: flex;`, despite their similar names; this is because `flex-basis` modifies the behavior of a flex *child*, not a flex container. `justify-content`, on the other hand, defines how a container should distribute its children. Toggling `display: flex;` toggles its dependents as well, reinforcing the relationship between properties using visual effects that are meaningful to the learner.

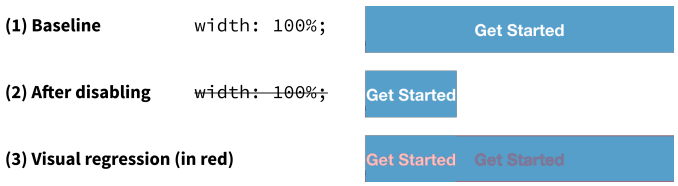

Figure 3. Visual regression pruning determines whether the CSS property `width: 100%;` is effective with respect to the page.

Visual regression pruning in Ply

To identify ineffective properties, we draw inspiration from commercial services for *visual regression testing*, which check for breaking changes to a UI codebase by comparing rendered webpages against a stored groundtruth snapshot. Ply extends this approach using a novel technique called *visual regression pruning* (VRP), which determines the visual significance of each line of code in a cascade of CSS styles. First, VRP captures a *baseline snapshot* of the visible portion of the webpage (Figure 3-1). Iterating over the cascade in descending order of precedence, the algorithm *disables each property* by removing it from the source stylesheet, captures a *second snapshot* of the resulting webpage (Figure 3-2), and invokes a black-box *image comparison* algorithm to compute the perceptual difference between the two snapshots (Figure 3-3).[2] If the difference exceeds a predetermined threshold, then the removed property is relevant to the visible portion of the webpage. Otherwise, the property is deemed *ineffective*, and hidden from inspection.

To identify base styles, Ply applies VRP twice: first, by checking for regressions to the entire visible portion of the page; second, by checking for regressions to the currently-selected element only. If a property does *not* affect the current element but *does* affect the rest of the page, the property must be overloaded by the current element. By comparing the outputs of each VRP pass, Ply identifies these overloaded properties as part of a *default base style*.

To prune implicit dependencies, Ply again applies VRP twice and compares the two sets. Rather than conditioning on area of effect, however, Ply runs VRP with and without a *candidate dependency* (such as `display: flex;` in the example from Figure 2g). If a property p is marginally effective, but becomes ineffective in the absence of the candidate dependency q, Ply concludes p is a dependent of q. To our knowledge, this is the first automated approach capable of identifying a subset of implicit dependencies between CSS properties, and represents a significant contribution of this work.

Implementation

Ply consists of a web application front-end and a Google Chrome extension, which communicate via WebSocket connections to a lightweight proxy server. The extension instruments the

[2] Our prototype implementation of VRP compares browser engine screenshots using pixel-level differentiation. The technique could easily be extended to use more sophisticated approaches based on computer vision.

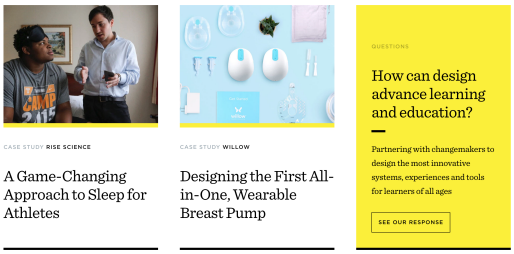

Figure 4. A grid feature on the IDEO homepage.

inspected webpage through the Chrome Remote Debugging Protocol and compares images using a modified fork of the `pixelmatch` algorithm by Mapbox.

STUDY 1: FEATURE REPLICATION

We conducted a user study to explore how pruning ineffective properties helps developers replicate features more quickly. We recruited 12 undergraduate and graduate students with web development experience, and evenly divided users into control and experimental groups through random assignment. Those in the control group used CDT, and those in the experimental group used Ply. Users were given HTML markup for a section of the IDEO homepage containing recent blog posts (Figure 4), and were asked to spend 40 minutes replicating the feature's appearance. After hearing the task description, both groups reported similar confidence levels on a 1 to 5 scale (Ply: $\mu = 3, \sigma = 1.14$; CDT: $\mu = 3.17, \sigma = 0.52$). Each user received a $20 Amazon gift card.

To structure the task and measure user progress, we defined three milestones: (1) styling the three tiles into a horizontal grid, (2) rendering each tile's image, included with the original HTML markup, and (3) visually differentiating the third tile by giving it a yellow background and hiding its image. These milestones covered a diverse set of CSS concepts, including flexbox behavior, how `height` is calculated, and patterns for overloading styles with higher-precedence rules. As a second priority, users could style the overall appearance of the page.

Results

Overall, Ply users completed their three milestones about 50% faster (Ply: $\mu = 16.67, \sigma = 1.63$; CDT: $\mu = 24.83, \sigma = 8.08$) (Figure 5). This difference was most pronounced on the first milestone, where Ply users were 3.5 times faster than CDT users (Ply: $\mu = 2.5, \sigma = 1.64$; CDT: $\mu = 8.83, \sigma = 4.167$). As hypothesized, all CDT users wasted time attempting to copy and debug ineffective properties, whereas Ply users did not copy any ineffective properties. Moreover, visual clutter in the CDT interface meant users struggled to identify relevant properties even when they were in full view.

For all milestones, the variation in completion time was markedly lower in the Ply group compared to CDT (Figure 5), even though the Ply group had greater variation in reported experience and confidence. For instance, both the least (P6) and most (P10) experienced users in the study used Ply. P6's only experience with CSS consisted of "small tweaks to other people's templates," and they reported a 1 out of 5 in confidence. By comparison, P10 reported a 4 out of 5, had several years of experience building websites for paying clients, and had recently completed an internship in front-end development at a major software company. Despite this difference in experience levels, P10 and P6 completed their third milestone in 16 versus 17 minutes, respectively. For the two CDT users with the widest gap in experience, this difference was 19 versus 35 minutes excluding outliers.

STUDY 2: LEARNING NEW DESIGN PATTERNS

Having shown that pruning supports developers during replication, we conducted a second evaluation to understand how Ply's embedded guidance could help novice developers learn new language concepts. We recruited 6 student developers with minimal web design experience: two had never used HTML and CSS outside of an undergraduate HCI course, and the last user (P5) had previously styled desktop applications using Qt stylesheets but had never used CSS itself. Each user was compensated with a $20 Amazon gift card.

Task 1: Methodology

Our first task evaluated whether users could use Ply's base style annotations to understand organizational approaches for visually-similar buttons on the Indiegogo website (Figure 6). We conducted a pre-task to elicit each user's familiarity with approaches for organizing CSS styles. During this task, the user arranged code snippets to replicate the appearance of the buttons. The provided snippets corresponded to the common styles (such as `font-size` and `padding`), the base white-on-pink colors, and the inverted colors. Each user generated and explained as many distinct arrangements as they could think of. Users then inspected the original example using Ply, constructed the example approach if they hadn't already, and contrasted this approach with their pre-task output.

Result 1: Developers learned new organizational approaches

Only P3 initially produced an arrangement of styles isomorphic to the Indiegogo example. This approach, analogous to method overriding in object-oriented programming, declares two rules: a complete set of base styles for the default button (cf. *parent class*), and a second rule containing just the overloaded colors (cf. *subclass*). Both rules are applied to the variant, but only the first rule is applied to the default element. In the pre-task, users expressed discomfort with overloading semantics and preferred to style each element by applying two rules: one containing only the common base styles (cf. *abstract base class*) with a separate color rule.

After inspecting Indiegogo with Ply, all users correctly constructed the overloaded approach using the provided code snippets. They characterized the approach using terminology from other programming domains: "to achieve the secondary style, they are composing classes which have overrides" (P5). Users

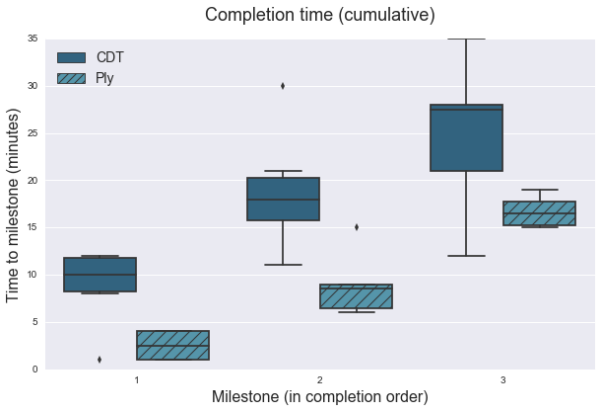

Figure 5. Ply users had lower cumulative completion times for all three milestones. Ply users also had less variation in their completion times, despite higher variation in confidence and experience.

Figure 6. Two buttons with overlapping visual characteristics.

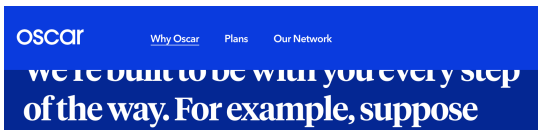

Figure 7. A sticky header with implicit dependencies.

identified scenarios in which the overloaded approach would be preferable to the disjoint approach they had previously preferred: "for theming and consistency on a large site" (P0), "if I had a lot of pink buttons, and only a few white buttons...you wouldn't want to type [the `.pink` class] every single time" (P4). Inspection exposed users to new approaches, as reflected in their unprompted remarks: "I actually haven't thought about doing it this way" (P2), "I might have thought of this before, but I wouldn't have done that — but if that's what professionals are doing, it seems better" (P0), "I re-learned something I forgot about overriding properties" (P4).

Task 2: Methodology

Our second task evaluated whether Ply's dependency overlay allows users to identify dependencies between CSS properties on a sticky header on the Oscar homepage (Figure 7). In the CSS specification, properties such as `top` and `z-index` only apply to elements with a `position` other than the default value of `static`. To elicit their prior understanding of these dependencies, we gave each user a toy example with these CSS properties, and asked the user to diagram the dependencies between properties. Next, each user inspected the Oscar example using Ply's dependency overlay, then drew a diagram for the dependencies within the Oscar example. Finally, users revisited the original example and drew a revised diagram, based on the knowledge they had gained from inspecting Oscar.

Result 2: Users identified new relationships

In the pre-task, none of the users identified a relationship between `position` and `z-index` or confidently characterized the effects of `position: fixed;` when asked. After using Ply to inspect Oscar, all 6 users correctly identified that `top`, `width`, and `z-index` depended upon `position` in the example. Users characterized the dependency correctly when prompted: "if you turn off `position: fixed`, `z-index` doesn't have an effect anymore" (P4).

Ply helped users confirm prior intuitions ("I am now confident that `z-index` depends on `position`," P3), revise misconceptions ("I see now that `z-index` requires a `position` to be set. I wouldn't have said [that] before," P2), and enrich their understanding of how properties could relate to one another ("Something about `z-index` would change as a result of position not being fixed. `position: fixed;` is doing something beyond pinning in place while you scroll," P5).

CONTRIBUTIONS

This paper introduces Ply, a web inspector designed to support novices in feature replication and program comprehension on professional webpages. To bridge the gap between browser engine semantics and novices' visual intuition, Ply implements a novel visual regression pruning technique to hide ineffective code and surface relationships between properties. Student developers used Ply to (1) replicate features more quickly and (2) identify and explain unfamiliar design patterns from professional examples. Our results provide preliminary evidence that effective *information reduction* and *embedded guidance* can make authentic learning from professional examples tractable, even for inexperienced developers.

REFERENCES

1. J. Brandt, M. Dontcheva, M. Weskamp, and S. Klemmer. 2010a. Example-centric programming - integrating web search into the development environment. *CHI* (2010).
2. J. Brandt, V. Pattamatta, and W. et al. Choi. 2010b. *Rehearse: Helping Programmers Adapt Examples by Visualizing Execution and Highlighting Related Code*. Technical Report.
3. B. Burg, A.J. Ko, and M.D. Ernst. 2015. Explaining Visual Changes in Web Interfaces. *UIST* (2015).
4. K.S. Chang and B.A. Myers. 2012. WebCrystal: Understanding and Reusing Examples in Web Authoring. *CHI* (2012).
5. Brian Dorn and Mark Guzdial. 2010. Learning on the Job: Characterizing the Programming Knowledge and Learning Strategies of Web Designers. *CHI* (2010).
6. P. Genevès, N. Layaïda, and V. Quint. 2012. On the analysis of cascading style sheets. *WWW* (2012).
7. P. Gross and C. Kelleher. 2010. Toward Transforming Freely Available Source Code into Usable Learning Materials for End-users. *PLATEAU* (2010).
8. J. Hibschman and H. Zhang. 2016. Telescope. (2016).
9. D. Klahr and K. Dunbar. 1988. Dual Space Search During Scientific Reasoning. *Cognitive Science* (1988).
10. A.J. Ko, B.A. Myers, and H.H. Aung. 2004. Six Learning Barriers in End-User Programming Systems. *VL/HCC* (2004).
11. R. Kumar, A. Satyanarayan, and C. et al. Torres. 2013. Webzeitgeist: Design Mining the Web. *CHI* (2013).
12. R. Kumar, J.O. Talton, S. Ahmad, and S.R. Klemmer. 2011. Bricolage - example-based retargeting for web design. *CHI* (2011).
13. S. Oney and B. Myers. 2009. FireCrystal: Understanding interactive behaviors in dynamic web pages. *VL/HCC* (2009).
14. P. Panchekha and E. Torlak. 2016. Automated reasoning for web page layout. *OOPSLA* (2016).
15. C. Quintana, B.J. Reiser, and E.A. et al. Davis. 2004. A Scaffolding Design Framework for Software to Support Science Inquiry. *Journal of the Learning Sciences* 13, 3 (2004).
16. David Williamson Shaffer and Mitchel Resnick. 1999. "Thick" Authenticity: New Media and Authentic Learning. *Journal of Interactive Learning Research* 10, 2 (1999).
17. J. Van Merrienboer and F. Paas. 1990. Automation and schema acquisition in learning elementary computer programming. *Computers in Human Behavior* 6, 3 (1990).
18. J. Wortham. 2012. A surge in learning the language of the internet. *New York Times* 27 (2012).