Ply: A Visual Web Inspector for Learning from Professional Webpages

Sarah Lim, Joshua Hibschman, Haoqi Zhang, Eleanor O'Rourke
Northwestern University
Evanston, IL
{slim, jh}@u.northwestern.edu, {hq, eorourke}@northwestern.edu

ABSTRACT

While many online resources teach basic web development, few are designed to help novices learn the CSS concepts and design patterns experts use to implement complex visual features. Professional webpages embed these design patterns and could serve as rich learning materials, but their stylesheets are complex and difficult for novices to understand. This paper presents Ply, a CSS inspection tool that helps novices use their visual intuition to make sense of professional webpages. We introduce a new *visual relevance testing* technique to identify properties that have visual effects on the page, which Ply uses to hide visually irrelevant code and surface unintuitive relationships between properties. In user studies, Ply helped novice developers replicate complex web features 50% faster than those using Chrome Developer Tools, and allowed novices to recognize and explain unfamiliar concepts. These results show that visual inspection tools can support learning from complex professional webpages, even for novice developers.

ACM Classification Keywords

H.5.0. Information Interfaces and Presentation: General

Author Keywords

Developer tools; web inspection; CSS; authentic learning.

INTRODUCTION

Novice programmers often rely on online resources while learning to code [3], particularly in the domain of web development [10]. When learning to style webpages using Cascading Style Sheets (CSS), novices look to tutorials on platforms like Codeacademy and CSS-Tricks to learn syntax and explore simple examples. However, few resources are designed to help novices create more complex visual effects [36]. For example, a novice who wants to overlap two elements may struggle if she does not know that the `z-index` property also requires the `position` property to be set. Moving beyond the basics requires substantial additional knowledge: developers must understand overloading in the cascade, interdependencies between properties, and modern layout models such as CSS Flexbox and Grid. These approaches are often opaque to novices, and there is an overall lack of materials designed to bridge this gap.

When tutorial examples do not meet developers' needs, they often turn to professionally-authored webpages for design inspiration [23, 25, 16, 11]. Professional webpages embed rapidly-evolving best practices and conventions not covered by tutorials, and are continually updated as new solutions arise. Most importantly, all webpages are freely inspectable using browser developer tools, which expose the Document Object Model (DOM) and CSS responsible for a page's appearance. As a result, professional websites represent an appealing class of resources to help novices learn implementation approaches.

In practice, however, professional webpages are too complex for novice developers to understand through inspection. In a needfinding study, we observed that novices approach inspection from a visual perspective, asking questions about how the features on a webpage are implemented in code (e.g. *"How does this webpage create overlapping boxes?"*). This visually-driven approach leads to two challenges. First, even a state-of-the-art web inspector might display over a hundred CSS properties at a time, many of which have no observable effect on the page. As a result, novices struggle to locate the lines of code responsible for an effect using visual intuition alone. Second, even after locating relevant code, novices with only superficial CSS knowledge have trouble understanding properties without intuitive visual effects. For example, the property `color: red;` is familiar to most novices and has a clear visual effect, whereas `position: relative;` is more abstract.

To overcome these challenges, we introduce *Ply*, a CSS and DOM inspection tool designed to help novices replicate visual features and understand CSS concepts on professional webpages. Ply identifies *visually relevant* CSS properties by analyzing their observable effects on the page. Using this information, Ply helps novices (1) locate relevant properties by pruning code with no observable effect on the page; and (2) understand cascading relationships and dependencies between CSS properties through contextual tooltips. By directly connecting properties and their relationships to meaningful visual output, Ply helps novices use their visual intuition to learn abstract concepts from complex professional webpages.

Our conceptual contribution is the idea that *web inspectors that incorporate visual relevance can help novices apply visual*

UIST 2018, October 14–17, 2018, Berlin, Germany.

ACM ISBN 978-1-4503-5948-1/18/10 ...$15.00.
https://doi.org/10.1145/3242587.3242660

intuition to learn from professional webpages. Our approach is informed by research in the learning sciences that provides guidelines for designing software to support novice *sensemaking* during scientific inquiry [32]. We adapt these guidelines for our domain, arguing that web inspection tools should support novices' visual approach to inspection by hiding visually irrelevant properties and explaining visually unintuitive concepts with in-situ hints. To quantify visual relevance, we introduce a novel *visual relevance testing* technique that measures the observable effects of CSS properties by systematically deleting them from the page source and testing for visual changes in the resulting page. This technique enables Ply's core features, including the first approach we know of that can automatically detect dependencies between CSS properties.

We evaluate Ply through two user studies that measure novices' ability to replicate visual features and understand CSS concepts, respectively. In our first study ($n = 12$), novices using Ply successfully replicated a complex grid feature from a professional webpage 50% faster than those using Chrome Developer Tools. In our second study, novices ($n = 6$) successfully recognized unfamiliar CSS design patterns and implicit dependencies using Ply, and generalized these approaches beyond the given examples. These results provide exciting evidence of the potential for visual inspection tools to support learning from professional webpages, even for novice developers.

RELATED WORK

Prior systems for example-driven development focus on helping programmers adapt curated examples or locate features in complex codebases. Ply builds on this work to target novices interested in learning expert approaches to static web design.

Programming with examples

While many existing systems support developers in foraging [25, 18, 6] and adapting [3, 23, 22] curated examples, comparatively few systems attempt to surface the programming concepts contained within those examples to support learning. For example, Lee et al. help users design webpages by modifying examples from a structured corpus [25], while Blueprint [3] helps developers incorporate code snippets from forums, blogs, and tutorials. These systems focus on identifying similar examples to provide inspiration, but cannot explain the language features and design principles embedded within those examples. Our work targets novice developers with professional aspirations who are actively invested in learning new CSS concepts, not just adapting existing designs.

Feature location in professional interfaces

Another class of systems aims to help developers perform *feature location* tasks, or identifying code responsible for functionality of interest within a program. Specifically, a number of systems provide visual affordances to help developers reverse-engineer dynamic behaviors within complex interactive applications. Rehearse [4] highlights the active lines of code during program execution, and locates related lines based on API definitions. In the web domain, Telescope [17] and FireCrystal [29] link DOM and CSS modifications to responsible lines of JavaScript code, and provide affordances for visualizing the resulting interaction timelines. Scry [5] represents the application state using a graphical timeline, and compares snapshots of DOM nodes at different points in time to help users identify visually-meaningful state changes. These systems focus on inspecting dynamic behavior, such as interactions driven by JavaScript. Such interactions may include small CSS modifications, but tools like Telescope, FireCrystal, and Scry are not designed to explain the entire webpage's appearance at one moment in time. In contrast, we explore static webpage inspection to help users learn about CSS behavior.

Moreover, while feature location tools can help developers inspect professional examples, these tools frequently abstract away the embedded development practices. A simplifying interface can sacrifice authenticity: for instance, WebCrystal [7] generates CSS snippets based on questions about an element's appearance, but reduces the cascade of authored styles to the *used CSS values* calculated by the browser engine. In other words, WebCrystal might display `width: 45.66667px;` rather than the authored property `width: 33%;`. Developers are unlikely to set an element's `width` to `45.66667px` in practice, limiting WebCrystal's utility for learning authentic styling practices. In contrast, Ply is designed to help developers inspect and understand CSS at the source level, and targets novices interested in learning real-world design patterns.

CHALLENGES OF INSPECTING CSS

Professionally-authored webpages provide an attractive corpus of learning materials that embed expert design patterns and are richer than available web design tutorials. Yet inspecting professional webpages with state-of-the-art browser tools is often intractable for novices. Reasoning about CSS is a nontrivial task for both humans and machines [19, 31], due to the complexity of the language and a lack of automated tooling beyond syntax-checkers [12, 30]. To our knowledge, no prior study has examined the difficulties associated with inspecting CSS on professional webpages. Before designing Ply, we therefore conducted a needfinding study with novice developers to understand these challenges.

Needfinding methods

First, we surveyed 20 undergraduate student web developers about their experience inspecting webpages and learning from HTML and CSS tutorials. We then followed up with an in-person study with ten of these developers. Eight users were novice CSS developers with experience on one or two course projects, and two users were more advanced and had completed internships focused on front-end web development.

During the in-person study, we asked users to talk aloud while replicating a single web feature (a responsive full-screen background image, login form, or responsive grid layout). Each user was given three professional example implementations of the target feature to inspect using Chrome Developer Tools (CDT).[1] Users could freely search the web for documentation or additional resources. We observed users and analyzed their progress using informal milestones for each feature (e.g.,

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

adding a background image; making the image cover the viewport). Participants were compensated with Amazon gift cards ranging from $15 to $25, depending upon the task length.

Needfinding results

None of the ten users successfully completed the replication task in full, and only the two most experienced users achieved any milestone beyond the most basic (adding the key HTML elements with margins and padding, but no other meaningful styles). The two more experienced participants made progress by relying on their prior knowledge of CSS to recreate the feature. Information overload was a recurring theme: all users repeatedly described feeling overwhelmed by CDT's dense interface, which displays the entire DOM at once, along with a lengthy cascade of matched CSS styles for the selected element. Through our observations, we identified two recurring obstacles: (1) there was a mismatch between novices' visual approach to inspection and the information displayed by CDT, and (2) novices lacked the conceptual knowledge needed to form hypotheses and reason about example code.

Visually ineffective properties

To our surprise, users of all skill levels followed the same reverse-engineering process, with varying degrees of success. First, users identified a visual entity of interest on the page (e.g. a row of grid cells), and formulated a hypothesis about the entity's implementation (e.g. "I'm looking for code that keeps these boxes in a row, maybe a `float: left;`"). Within CDT, users searched for a DOM element with the hypothesized styles. When an element is selected in CDT, the browser engine computes the set of CSS rules matched to that element, and displays these rules in descending order of precedence based on static factors such as declaration order and the specificity of each rule's selector. If users noticed a promising set of styles within this cascade, they transferred those styles into their editors. In the best case, these additions brought their output closer to the example, and they moved on to a new objective. Far more often, the changes had no visual effect, and users either revised their hypotheses or gave up.

These visually ineffective properties were the primary source of frustration and wasted time we observed. We say that a fragment of CSS is *ineffective* if its presence or removal has no effect on the page's appearance. Any property that is overridden by a higher-precedence declaration is ineffective, and CDT denotes these properties with a strikethrough. However, we were surprised to find that *many properties that appeared relevant in CDT were nonetheless visually ineffective* (Figure 1). Complex interfaces often require a large number of styles to ensure consistency in rendering across all possible conditions, and the global cascading nature of CSS means that it is often easier for developers to declare redundant properties "just in case," rather than risk failure in an edge case. When these properties appear at or near the top of the cascade, they are indistinguishable from relevant code in the CDT interface, and frequently misled users in our study.

Beyond slowing down feature replication, ineffective properties defied novices' visual intuition and prevented comprehension of example code. Throughout the inspection process,

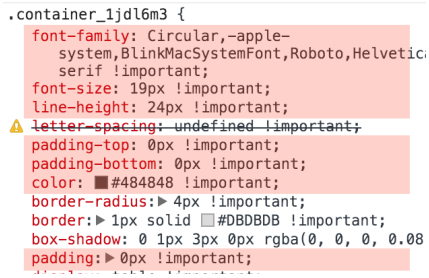


Figure 1. Ineffective properties (highlighted in red) appear active in CDT, but have no visible effect on the webpage when disabled. This example is taken from the highest-precedence CSS rule in the cascade.

novices approached inspection from a visual perspective, relying on visual terminology and deictic references to describe their goals and hypotheses ("How do I make this button look *like this*?"). Since ineffective properties have no visual effect, novices could not understand why these properties appeared active in the inspector. Without a coherent mental model, novices either gave up or resorted to copying and pasting code without attempting to understand the example.

Missing conceptual knowledge

Even after users located a set of relevant styles, they frequently struggled to understand how the different properties worked together to produce the element's appearance. This was particularly true in cases where related properties were distributed across multiple rules within the cascade, often overriding one another. Here, we discuss two classes of approaches frequently used in professional code that were opaque to our users.

Visual subtypes. Professional developers often modularize styles by declaring a set of base rules to ensure uniformity across components (akin to a base class in object orientation), and then declaring additional rules to override specific properties (akin to a subclass):

```
1  <button class="button">Default</button>
2  <button class="button button-inverted">Inverted</button>

3  .button {
4    font-family: sans-serif;
5    padding: 11px 18px;
6    background-color: pink;
7    color: white;
8  }
9  .button-inverted {
10   background-color: white;  /* overrides line 6 */
11   color: pink;  /* overrides line 7 */
12 }
```

This design pattern, which we call *visual subtyping*, was particularly challenging for the novices in our study who did not understand the behavior of the cascade. Given two rules with conflicting property declarations, browsers assign precedence based on static features such as selector specificity and source location.[2] We found that most users in our study were unfamiliar with these rules, which have no intuitive visual basis, and could not interpret examples like the one above.

Implicit dependencies. While the most straightforward visual changes can be achieved with a single CSS property (e.g. `color: red;`), more sophisticated effects require coordination

[2] https://www.w3.org/TR/CSS2/cascade.html

between multiple properties. Consider the following example based on a scenario we repeatedly witnessed during needfinding: a novice developer, Cuthbert aims to vertically center text within a , and writes the following:

```
1  <div class="container">
2    Cascading Style Sheets are so expressive!
3  </div>

4  .container {
5    height: 100px;
6    vertical-align: middle;
7  }
```

This code does nothing to vertically center the text, because the vertical-align property only applies to elements with display: inline or display: table-cell;, and elements are assigned display: block; by default. Frustrated, Cuthbert searches "vertical align middle" and notices that the top suggested completion is "vertical align middle not working."[3] Based on a StackOverflow suggestion, Cuthbert adds display: table-cell; below line 6, a common hack[4] used to center text in elements. Now, vertical-align: middle; behaves as expected – it has an *implicit dependency* upon display: table-cell;.

Implicit dependencies were particularly baffling to novices in our study, as most had only used straightforward CSS properties such as color and did not realize that properties could depend upon one another. While such dependencies are well-defined in the specification, they are not centrally documented and impossible to statically infer. Our conversations with professional front-end developers confirm that implicit dependencies are a common source of frustration, often tediously memorized through years of practice.

Without an understanding of these core concepts, novices in our study struggled to apply their visual intuition to the examples. This is consistent with the general finding that novice programmers have trouble identifying relationships between example code constructs [13, 4, 21]. On multiple occasions, our users recognized that they were missing important knowledge, but lacked the domain vocabulary to formulate an effective search query. This inhibited the less experienced participants from searching for resources, as the more advanced ones did. When asked to rate the usefulness of professional examples for replication on a 1 (not helpful) to 10 (very helpful) scale, one user said, *"Either a 7 or a 1, if there's some concept I don't understand. If there ended up being something that required some background knowledge...you just get lost."*

DESIGN RATIONALE

Our needfinding study confirms that novice developers are unable to replicate visual features from professional webpages using current tools. We identified two obstacles that made inspection difficult for novices: (1) a mismatch between their visual approach to inspection and the information prioritized by CDT, and (2) missing conceptual knowledge needed to form hypotheses and reason about example code. In this section, we describe our approach for overcoming these obstacles.

[3]This is true at the time of submission.

[4]Before the Flexbox module, CSS 2.1 did not provide an idiomatic way to vertically center text within a block-level element.

We draw on literature from the learning sciences on supporting novices during sense-making tasks. Broadly speaking, *sense-making* refers to the process of building an understanding of an artifact or example by constructing mental representations of what is known [34]. In our work, we build upon a set of guidelines for designing software to support sense-making in the context of scientific inquiry, a domain with notable similarities to learning from code examples on the web. In scientific inquiry, sense-making is an iterative process that involves reasoning about a phenomenon, testing conjectures empirically, and deriving new understanding from the results [20, 32]. During web inspection, programmers follow a similar iterative process that involves forming hypotheses about how a feature might be implemented, searching the DOM and style cascade for evidence, and refining hypotheses accordingly. Quintana et al. [32] propose a set of guidelines for designing software to help learners overcome obstacles during scientific sense-making. Given the similar obstacles associated with sense-making during scientific inquiry and web inspection, we argue that these guidelines can inform the design of web inspection tools for novices.

Obstacle: mismatch between novice intuition and tools

In scientific disciplines, learners must use formal scientific representations to express their understanding and empirically test hypotheses [24, 32]. However, learners are often overwhelmed by the formalisms used by experts [8] and struggle to use their own intuition to reason about unfamiliar phenomena [33]. Likewise, experienced programmers rely on formal representations of programming concepts to make sense of complex code and documentation. However, these patterns are often opaque to novices, preventing them from building a deep understanding of the domain [1, 35, 27].

To overcome this obstacle, Quintana et al. argue that software should *use representations and language that bridge learners' understanding* by explaining complex concepts in ways that build on learners' intuition [32]. Consistent with prior studies [13] of novice programmers more generally, our needfinding study showed that novices approach inspection from a visual perspective, and struggle to reason about the many visually ineffective properties displayed by CDT. Building on Quintana et al.'s guideline, we argue that web inspection tools should bridge from novices' intuitive visual approach by highlighting properties of visual interest:

Characteristic 1: Hide visually-irrelevant code from inspector output to minimize information overload and support novices' visual approach to sense-making.

Current CSS inspection tools display a large number of properties with no observable effect, complicating novices' sense-making efforts. Ply implements Characteristic 1 by giving users the option to remove visually ineffective properties from the inspector output. For example, when a learner inspects an element styled with @media query rules for multiple screen sizes, Ply will hide irrelevant properties designed for smaller screens. To compute relevance, we introduce a novel *visual relevance testing* technique that allows Ply to identify and hide visually-irrelevant code.

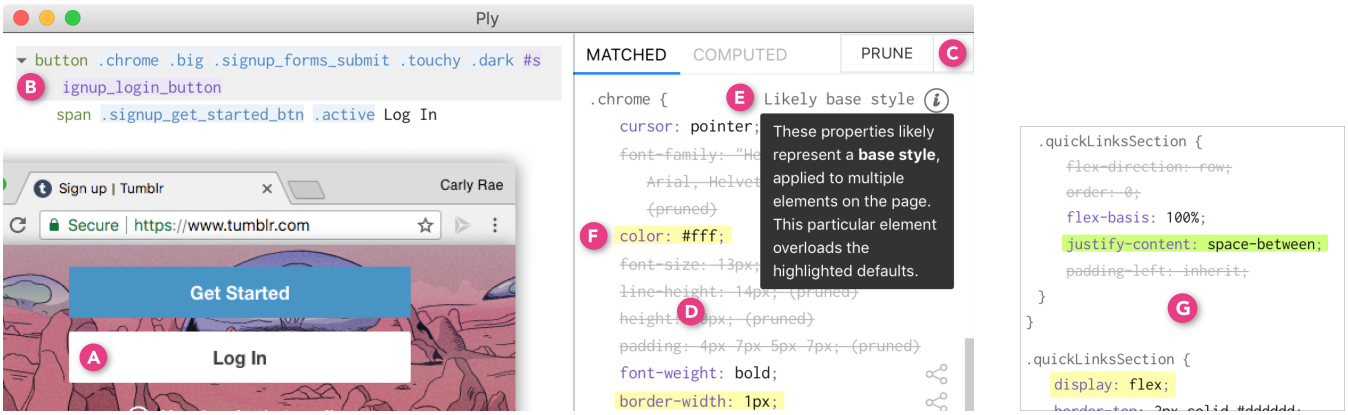


Figure 2. Left: Ply's DOM and CSS inspection interface is designed to minimize visually irrelevant information during inspection tasks. Right: Ply's implicit dependency overlay reveals dependencies between CSS properties across different rules. Here, the developer has requested dependencies for display: flex; (highlighted in yellow). Ply identifies that justify-content as a dependent of display: flex;, whereas flex-basis is not.

Obstacle: missing conceptual knowledge

During scientific inquiry, learners struggle when they lack the domain knowledge needed to form hypotheses about scientific phenomena and draw inferences from data [32]. Similarly, programmers rely on heuristics and conceptual knowledge of programming constructs and language semantics as they make sense of complex examples [1, 35, 27], and novices struggle to reason about unfamiliar constructs without this understanding.

To overcome this obstacle, Quintana et al. argue that software should *embed expert guidance* into the sense-making process to provide missing domain knowledge. In our needfinding study, we discovered that novices lack conceptual understanding required to reason about professional CSS design patterns, particularly when multiple properties work together to produce a visual effect. We identified two constructs that were particularly challenging: *visual subtypes* and *implicit dependencies*. Building on Quintana et al.'s guideline, we argue that web inspection tools should fill in missing conceptual knowledge about these relationships by providing in-situ guidance:

Characteristic 2: Embed contextual guidance into inspector output to explain how CSS properties coordinate to produce visual effects.

Current CSS inspection tools have no awareness of the relationships between individual properties, and therefore cannot provide contextual guidance. Ply implements Characteristic 2 by providing tooltip hints beside CSS properties that relate to other properties via subtyping or implicit dependencies. For example, when a learner inspects a button that overrides the default button style, Ply displays a hint about visual subtypes.

PLY: A VISUAL WEB INSPECTOR

We use these guidelines to design Ply (Figure 2), a visual web inspector modeled after CDT and similar tools. Ply augments the standard inspection interface with the ability to hide ineffective properties, identify instances of visual subtyping, and trace implicit dependencies. As a representative example, consider a novice developer Stella 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). Unlike CDT, which displays the entire document at once, Ply treats the selected element as the root of the inspection tree and hides any non-descendants. This allows Stella to scope her inspection task to the region of interest. As in CDT, hovering over an element in Ply highlights the corresponding region on the page, reinforcing the link between code and visual output.

Relevance pruning

When Stella inspects an element, Ply displays the corresponding cascade of matched CSS rules. Clicking a property toggles it on and off, allowing her to observe the result. To hide ineffective properties, Stella clicks the "Prune" button in the toolbar (Figure 2c). After a brief delay, visually-ineffective properties are crossed out and displayed in greyscale (Figure 2d), with the option to hide them completely. By hiding visually ineffective properties, Ply helps novices locate the CSS responsible for a feature of interest.

Understanding the cascade through visual subtypes

To help Stella understand the behavior of the cascade, Ply detects examples of visual subtyping and annotates their corresponding base styles with explanatory tooltips. The Tumblr homepage in Figure 2 uses visual subtypes to define a default button style ("Get Started") and an alternate color scheme ("Log In"). In this case, the "Log In" button is a visual subtype, because it overrides a subset of its base styles.

When Stella inspects the "Log In" button and prunes the cascade, Ply displays a *Likely base style* hint next to the base style rule (Figure 2e). This rule defines padding and typography, along with the default grey-on-white color scheme. Mousing over the hint reveals a tooltip, which explains the concept of a base style in intuitive language. Ply highlights the specific properties overridden by the subtype, background-color and color (Figure 2f). Hovering over either highlight displays a second tooltip explaining the concept of an overridden property. These in-situ hints can be used to fill in missing syntax knowledge [15] and provide expert guidance during problem-solving

[2]. By linking terms such as "specificity" and "overriding" to concrete visual examples, Ply helps novices restate their intuition in terms of expert practice.

Implicit dependencies

Relevance pruning reveals *which* properties are relevant, but does not explain *why* they are relevant. For properties corresponding one-to-one with a visual effect (e.g. color, margin, width), toggling them on and off suffices to illustrate their role. However, many properties (e.g. display, position) do not produce an effect by themselves; rather, they serve as implicit dependencies for other properties in the cascade. Without advance knowledge of these dependencies, users may not understand why such properties are visually relevant.

Stella turns her attention to a footer element, which uses the display: flex; property (Figure 2g). She is only vaguely familiar with Flexbox layout, but guesses that this property turns its element into a flex container, and that some of the other properties behave as flexbox modifiers. Based on naming similarity, Stella guesses that flex-basis is related to display: flex;. While her guess is reasonable, it is incorrect: flex-basis is not related to display: flex; because display: flex; defines a flex *container*, and flex-basis modifies the behavior of a flex *child*. Conversely, while justify-content does not contain a flex- prefix, it defines the behavior of a flex container and therefore depends upon display: flex;.

To help novices make sense of these relationships, Ply reveals the dependents of unfamiliar properties. Stella clicks on the "Show dependents" icon next to the display: flex; property, which highlights the selected property in yellow, and all of its dependents in green (Figure 2g). Hovering over justify-content: space-between; displays a tooltip explaining that the property implicitly depends on the selected property display: flex;. Toggling the parent property now toggles its dependents as well, reinforcing the relationship visually. By surfacing these non-obvious relationships, Ply helps novices develop a more precise understanding of CSS behaviors.

ANALYZING CSS BASED ON OBSERVABLE EFFECTS

In contrast to existing inspection tools, which do not consider rendered output when computing CSS property relevance, Ply uses image comparison to compute visual relevance *from the user's perspective*. We consider a CSS property to be *visually relevant* if the user can observe its effect on the page. This definition of visual relevance is key to detecting ineffective properties. While CDT can reliably determine whether a property is *active* (i.e. evaluated and not overridden) within the cascade, an active property does not necessarily have an observable effect. For instance, a property might re-define the browser's default styles, have an unsatisfied implicit dependency, or only apply to an occluded element.

In this section, we introduce our core approach for determining whether CSS properties are visually relevant, and describe how this technique can be applied to (1) prune ineffective properties, (2) identify visual subtypes, and (3) compute implicit dependencies between properties. Given the prevalence of implicit dependencies in CSS, this last contribution is particularly significant—to the best of our knowledge, no existing technique can automatically detect these dependencies.

Visual relevance testing

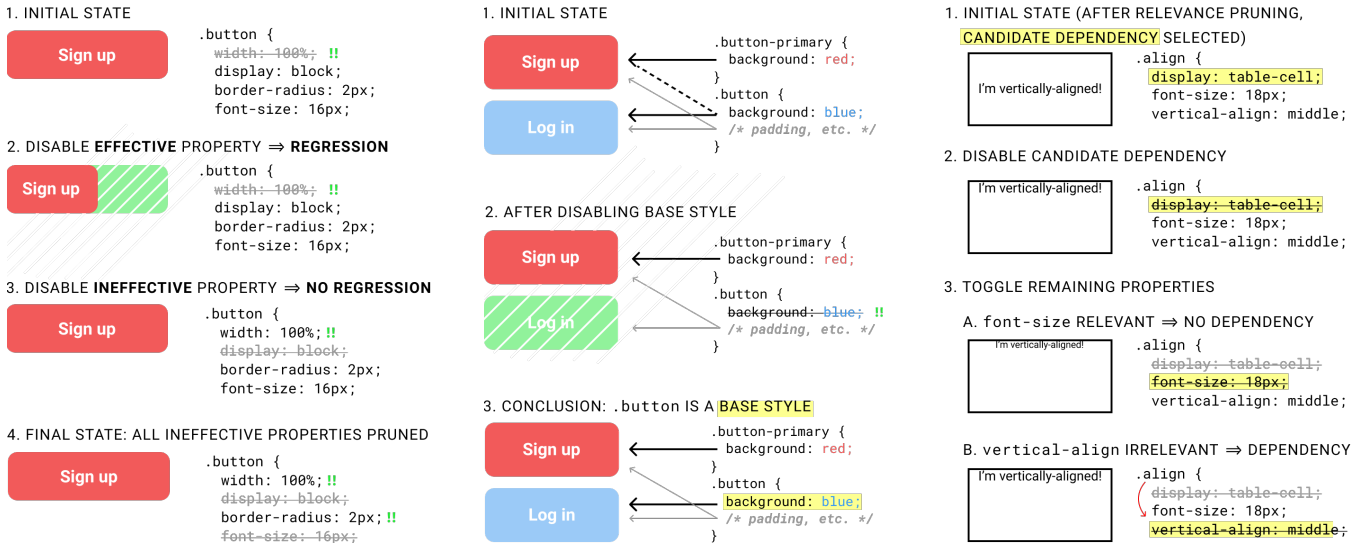
In order to compute visual relevance from the user's perspective, we draw inspiration from visual regression testing, a commercial approach to testing programs with graphical outputs (such as web applications). To confirm that program modifications do not break the application in unexpected ways, the developer defines a test suite with ground truth screenshots. The testing framework applies the program modifications, re-renders the application, and compares the updated views to the ground truth screenshots using a black-box image comparison algorithm. Any change in the application's appearance constitutes a *visual regression*, corresponding to a potential breakage. Visual regression testing has been implemented in commercial continuous integration services (**https://percy.io/**), and used to provide in-editor warnings about breaking changes [26].

Our key insight is that visual regression testing can be adapted to determine the precise visual effect of any CSS fragment. We introduce a new *visual relevance testing* technique that identifies relevant CSS properties by systematically deleting them from the page source and testing for visual regressions in the resulting page. The main idea behind this approach is that *a CSS property is relevant if and only if its deletion causes a regression*. Visual relevance testing thus enables a class of techniques for analyzing CSS fragments in terms of their observable effects.

Formally, we define a predicate IsRelevant(p, R) to determine whether disabling a property p (e.g. margin: 0 auto;) causes a regression within a region of interest R (e.g. the viewport). First, a base screenshot of R is captured. Next, the property p is commented out in the stylesheet source, temporarily disabling it for the page. After disabling p, a new screenshot of R is captured and compared with the base screenshot using a black-box image comparison algorithm. Finally, the algorithm returns a boolean denoting whether the two screenshots differ according to the comparison, and therefore whether the property is visually relevant.

Application 1: Relevance pruning

Visual relevance testing enables the automatic identification of active yet visually ineffective properties; this allows Ply to prune these properties from its inspector output. Figure 3(a) illustrates how we perform relevance pruning for a given DOM element, in this case a "Sign up" button. Ply first requests the cascade of matched CSS properties from the browser engine. Iterating over properties in descending order of precedence, Ply computes IsRelevant(p, viewport) for each property p (Figure 3(a)-1) to check for regressions anywhere within the browser viewport. If IsRelevant returns True, the property's removal causes a regression, and immediately restored (Figure 3(a)-2). Otherwise, the property is deemed ineffective and left disabled (Figure 3(a)-3). The resulting pruned cascade contains only properties with a visual effect on the webpage (Figure 3(a)-4).



(a) Relevance pruning (b) Visual subtypes (c) Implicit dependencies

Figure 3. Ply uses three applications of visual relevance testing to (a) hide irrelevant code, (b) surface visual subtyping relationships, and (c) identify implicit dependencies between properties.

Application 2: Visual subtypes

Our second application of visual relevance testing identifies visual subtype relationships, a visually-salient example of overloading behavior in the cascade. Here, the main idea is to check *where* on the page a regression occurs. Consider the example in Figure 3(b)-1: both buttons share common styles (padding, font, etc.), but the "Sign up" button is a visual subtype because it overrides the default blue background (akin to method overloading in object orientation). Disabling the default background color will cause a regression on other buttons in the DOM, but *not* on the inspected element (Figure 3(b)-2). This allows Ply to determine that background: blue; is a *base style*, and that the "Sign up" button is a visual subtype of the default button style.

Subtype detection amounts to calling IsRelevant twice on different regions, and comparing the results. First, IsRelevant(p, viewport) tests for a global regression anywhere in the viewport ("Are any visible buttons blue?"). Next, IsRelevant(p, element) tests for a local regression to the inspected element's bounding box ("Is the current button blue?"). If there is a global regression but not a local one, then p must be visually effective but overridden in the current element's cascade, which indicates that it is a base style.

Application 3: Implicit dependencies

Our final application of visual relevance testing identifies implicit dependencies between properties in the cascade. Figure 3(c) illustrates the intuition: if a property p depends on another property q, then IsRelevant(p, viewport) will return True if and only if property q is enabled. This is because disabling q leaves the dependency unsatisfied, rendering p ineffective.

Before computing dependencies, the algorithm first performs relevance pruning (Figure 3(a)) to filter out visually ineffective properties. If one of the remaining properties does not have an intuitive visual effect, the user can select the property to query for its dependents (Figure 3(c)-1).[5] We call this property a *candidate dependency* because it might serve as a dependency for other properties in the cascade. Given this user-selected candidate dependency q, the algorithm temporarily disables q (Figure 3(c)-2) and re-prunes all other effective properties. If a property p now becomes ineffective without q (Figure 3(c)-3b), the algorithm concludes that p depends on q, since p is effective if and only if q is active.

Existing approaches to CSS analysis

Our approach to visual relevance testing is closely related to two areas of research in programming languages and software engineering. *Program slicing* aims to approximate the minimal subset of a program necessary to preserve some feature of interest, and has been recently been applied to picture description languages with graphical outputs [37]. *Redundancy analysis* broadly aims to identify and optimize redundant program statements. For CSS, such analyses generally focus on computing the coverage of individual selectors to determine which style rules apply to which DOM elements. This relation can be tested dynamically at runtime [28] or verified statically by formally modeling selector logic and DOM manipulation [12, 14]. Like CDT, however, these approaches define redundancy in terms of what code is (or will be) *evaluated as active*, rather than what code has an *observable effect*.

We build upon this body of work in two ways. First, visual relevance testing imposes a weaker definition of redundancy, in which a property is deemed redundant if it is visually ineffective. This is the criterion of interest during web inspection, and allows our relevance pruning technique to eliminate not

[5] In practice, many properties with unintuitive effects are included only as dependencies necessary for other properties.

just inactive rules, but also active rules with no visible effect. Second, we demonstrate how repeated application of program slicing and redundancy analysis can be used to infer relationships between properties, such as visual subtyping and implicit dependencies. No prior work has applied redundancy analysis to perform inference beyond stylesheet maintenance and dead code elimination.

Implementation details

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 inspected webpage through the Chrome Remote Debugging Protocol.[6] To perform image comparison, our prototype uses the pixelmatch algorithm by Mapbox;[7] our custom fork early-returns after finding the first differing pixel. Since visual relevance testing treats image comparison as a black box, the difference threshold could be adjusted depending on the inspection context, or replaced by more sophisticated techniques capable of identifying visual features within interfaces (e.g. those introduced in [9]). Source code for the front-end, proxy server, and Chrome extension are available on GitHub.[8]

STUDY 1: FEATURE REPLICATION

We conducted two user studies to evaluate how Ply supports novice developers in inspecting professional examples. Our first study asked whether and how pruning ineffective properties can help developers replicate features more quickly. We recruited 12 undergraduate and graduate students with varying levels of web development experience, and randomly divided them into control and experimental groups. The control group used CDT, and 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. 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 provided 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 for block-level elements, and patterns for overloading styles with higher-precedence rules.

Users were given a walkthrough of each milestone, including the expected visual criteria for success, and told they could complete the milestones in any order. As a secondary priority after the milestones, users were told they could style the overall appearance (colors, typography, etc.) of the page. Tasks such as setting the font and colors of a webpage were more straightforward for users, because they involved independent properties with straightforward meanings rather than coordinating styles on multiple related nodes.

[6] https://chromedevtools.github.io/devtools-protocol/

[7] https://github.com/mapbox/pixelmatch

[8] At https://github.com/sarahlim/ply and https://github.com/sarahlim/chrome-remote-css

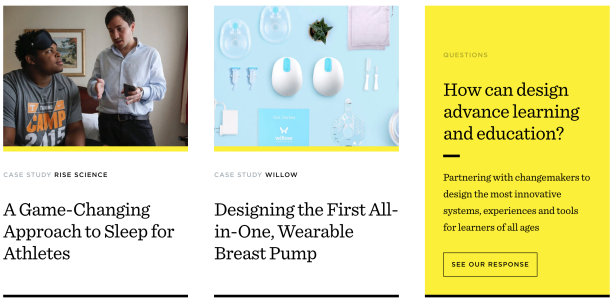


Figure 4. In our first user study, developers replicated this grid feature from the IDEO homepage.

After hearing the task description, both groups reported similar confidence levels on a 1 to 5 scale (Ply: $\mu = 3, \sigma = 1.14$; CDT: $\mu = 3.17, \sigma = 0.52$). To guard against self-perception bias, we asked users to describe their previous experience with HTML and CSS. The authors independently assigned scores based on reported experiences such as creating a personal webpage, completing a front-end web internship, building webpages for side projects, and serving as a teaching assistant for the Human-Computer Interaction course, which teaches basic web design. These assigned experience scores did not differ meaningfully from self-reported confidence levels. Each user received a $20 Amazon gift card for their participation.

During the task users had access to three windows: (1) the example itself, (2) an inspection tool (either Ply or CDT), and (3) a JSBin live editing environment. The editor was pre-populated with the feature's outerHTML markup, but contained no styling. CDT was initialized by selecting the DOM node corresponding to the feature root. Ply was initialized by setting the inspection context to the feature root, and pruning all nodes in advance. Since the aim of the study was to isolate the impact of relevance pruning on replication speed, users did not have access to Ply's visual subtyping or implicit dependency annotations.

Results

Overall, Ply users completed their three milestones about 50% faster (Ply: $\mu = 16.67$ minutes, $\sigma = 1.63$; CDT: $\mu = 24.83$ minutes, $\sigma = 8.08$) (Figure 5). This difference was most pronounced on the first milestone, where Ply users were 3.5 times faster than CDT users (Ply: $\mu = 2.5$ minutes, $\sigma = 1.64$; CDT: $\mu = 8.83$ minutes, $\sigma = 4.167$). For the milestone completion time dependent variable, Ply users finished the first two significantly faster than control users ($M1 : t(10) = -3.5, p = .01$; $M2 : t(10) = -3.4, p = .01$). The difference was not significant for the third milestone ($t(10) = -2.4, p = .06$), likely due to our small sample. One possible explanation for the front-loading is that 8 out of 12 users completed the grid milestone first, which only required the user to locate one property on the starting node. Since Ply's interface showed only a single collapsed node and a handful of relevant properties, users quickly identified the display: flex; property in the inspector, toggled it on and off to verify its effect, and added it to their solution. Conversely, CDT displayed multiple ineffective properties and a confusing array of styles denoting various states of inactivity, obscuring the relevant lines of code.

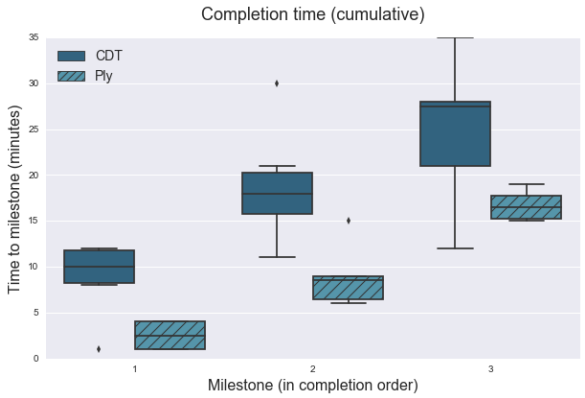


Figure 5. Ply users had lower cumulative completion times for all three milestones. Ply users also had less variation in their completion times, despite higher variation in confidence and experience.

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 slightly greater variation in experience levels and reported 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 webpages 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 final milestone in 16 versus 17 minutes, respectively. For the two CDT users with the widest gap in experience and confidence, this difference was 12 versus 35 minutes. While it is inappropriate to generalize with such a small sample, this result is consistent with our needfinding observations that highlighting visually-salient properties can help narrow the performance gap between novice and intermediate developers by giving novices access to the information filtering heuristics used by more experienced developers.

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 six student developers with very minimal web design experience; two had never used HTML and CSS outside of an undergraduate HCI course, and another had previously styled desktop applications using Qt stylesheets[9] but had never used CSS itself. During the study, developers described their prior experience with CSS and example-based learning, completed two 20-minute tasks, and provided feedback on their user experience. Each user was compensated with a $20 Amazon gift card.

Task 1: Visual subtyping

Our first task evaluated whether Ply's visual subtype detection and guidance could help users understand the behavior of the cascade. We used a pair of buttons on the Indiegogo homepage as an example (Figure 6). First, we conducted a pre-task to elicit developers' prior knowledge of style organization approaches. Developers were given a sheet of paper with a screenshot of the Indiegogo buttons and three sets of paper code snippets corresponding to the common button styles, the white-on-pink color scheme, and the inverted color scheme, respectively. We asked developers to construct CSS rules using these paper code snippets and draw lines connecting each rule to the corresponding button element(s). Developers were instructed to generate as many distinct approaches as they could think of, then contrast the approaches and explain their rationale for each. There were no constraints on the number of snippets that could be grouped in each rule, or the cardinality of the matchings between rules and elements.

[9] http://doc.qt.io/archives/qt-4.8/stylesheet.html

Figure 6. In study 2, users inspected this Indiegogo example which uses visual subtypes to style buttons with overlapping visual characteristics.

After the pre-task, we asked developers to inspect the original buttons from the Indiegogo webpage using Ply, then reconstruct the approach used in the example. Finally, we asked developers to contrast the approach used by the Indiegogo example with the approaches they generated during the pre-task.

Result 1: Developers learned new organizational approaches

The Indiegogo website style guide organizes their button styles using visual subtyping. A complete set of base styles is declared for the default "Sign up now" button, and a second rule inverts the color properties of the "Learn more" button. Only P3 was able to produce this arrangement of styles during the pre-task. Most users expressed discomfort with overloading behavior and preferred to style each element by applying two rules: one containing only the common base styles (akin to an abstract base class) with a separate color rule (akin to a virtual function implementation).

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

Our second task evaluated how Ply's implicit dependency detection and guidance helps users identify dependencies between CSS properties on a sticky header on the Oscar home-

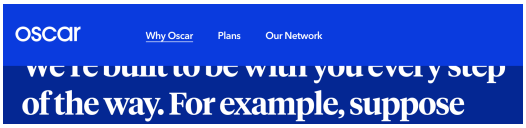


Figure 7. In study 2, users inspect this sticky header from the Oscar homepage, which is styled using properties with implicit dependencies.

page (Figure 7). The header is implemented using the top and z-index properties, which only apply to elements with a position value other than the default of static.[10] To elicit users' prior understanding of these dependencies, we presented each user with a toy example depicting a blue rectangle overlapping a red rectangle. The blue rectangle had the following properties applied:

```
position: fixed ;
top: 0;
width: 100%;
z-index: 3000;
```

In this example, the property position: fixed; is a dependency for top: 0; width: 100%;, and z-index: 300;. The top and z-index properties are used to adjust an element's position, so they always require the element to have a position other than the default value of static. (Incidentally, the top Google search completion for "z-index" is "z-index not working.") width does not always depend upon position in the CSS specification, but applying position: fixed; removes an element from the page flow, so a percentage-based width will be computed relative to the entire viewport, rather than the parent element. We constructed this example to parallel the Oscar header implementation, in order to evaluate users' prior knowledge of implicit dependencies.

We asked users to draw a diagram representing the dependencies between properties in this example. Each property-value declaration was represented by a node. If the user believed that a property p depended upon a property q, they drew a directed edge from p to q. If they were unsure of the direction of the relationship, they drew an undirected edge between p and q. If they were unsure, but tentatively believed that a relationship existed, they drew a dotted undirected edge.

After drawing and explaining their diagram, each developer inspected the Oscar example using Ply's implicit dependency tooltips, then drew a diagram for the dependencies within the Oscar example. Finally, developers revisited the original toy example and drew a revised diagram, based on the knowledge they had gained from the Oscar task.

Result 2: Developers identified new relationships

In the pre-task, none of the users could identify a relationship between position and z-index or characterize the effects of position: fixed; when asked: *"I don't remember all the things you can specify with position"* (P4). After using Ply to inspect Oscar, all 6 users drew correct diagrams, showing that top, width, and z-index depended upon position in Oscar. Moreover, users accurately characterized the nature of the dependency when asked: *"z-index's effects depend on whether position is fixed or not: if you turn off position: fixed, z-index doesn't have an effect anymore"* (P4).

Developers used Ply to confirm their prior intuitions (*"I am now confident that z-index depends on position,"* P3) and revise misconceptions (*"I see now that z-index requires a position to be set. I wouldn't have said [that] before...it seems like z-index makes sense without having position: fixed;,"* P2). The least-experienced developer, who had only used the Qt stylesheet language, enriched their mental model 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). These findings show that Ply's tooltip guidance successfully helped novices inspect, make sense of, and generalize this design pattern.

DISCUSSION

This paper presents Ply, a web inspector designed to help novice developers learn CSS by exploring complex professional webpages. In a needfinding study, we found that novice developers approach web inspection from a visual perspective. Therefore, rather than overwhelming users with properties that have no visual effect on the page, Ply uses *visual relevance testing* to hide irrelevant code. Ply also provides embedded guidance to explain visually unintuitive relationships between properties. Our conceptual approach builds on design principles from the learning sciences for supporting novice sense-making, which we adapt to the domain of making sense of professional webpages. In particular, we argue that web inspectors that incorporate visual relevance can help novices apply visual intuition to learn from professional webpages. Through two user studies, we found that Ply allows novice developers to replicate complex web features more quickly than with Chrome Developer Tools, and that novices understand unfamiliar design patterns and concepts after inspecting professional examples with Ply.

Limitations

Since visual relevance testing relies on snapshot comparison, the technique in its current form cannot be used to inspect JavaScript-driven animations and interactive behaviors. It would be straightforward to extend our approach to support CSS keyframe animations, as well as interactions implemented using :hover pseudo-classes. Non-deterministic interface elements, such as live-updating feeds and dynamic widgets, can also produce false positives during relevance testing. One potential solution is to allow the user to exclude certain regions of the page from the image comparison procedure. Finally, our prototype of Ply cannot teach advanced features of CSS preprocessors such as Sass and Less,[11] which support mixins and custom function definitions. While visual relevance testing can still help users reason about the compiled CSS, and even link to the corresponding preprocessor code when sourcemaps are available, full support for these language extensions would require adding a compilation pipeline.

[10] https://www.w3.org/TR/css-position-3/#property-index

[11] https://sass-lang.com/ and http://lesscss.org/

Future work

"*Style sheet languages are terribly under-researched.* This statement dates back from 1999, but it is still true" [12]. This statement dates back from 2012, but it is also still true. As CSS remains the only major option for styling webpages, there is an urgent need for new tools capable of reasoning about the language's increasingly complex semantics. Despite its maturity, CSS lacks a formal grammar and machine-readable specification (aside from a partial encoding of the CSS 2.1 layout model introduced in [30]). Moreover, while many groups have studied the usability shortcomings of CSS, research has overwhelmingly focused on designing replacement languages rather than improving available tooling [12]. Implicit dependencies in particular are under-researched and pose a significant usability barrier; Ply takes a first step towards improving the usability of CSS by surfacing implicit dependencies between properties affecting the same element. Future work should address more complex dependencies, such as those between properties on different elements, and explore possibilities for inferring language-wide dependencies from a diverse corpus of webpages.

Another potential avenue for future work involves incorporating visual relevance testing into human-in-the-loop refactoring and maintenance systems. By design, visual relevance testing only checks whether a CSS property is effective on the currently-visible webpage. For instance, a modal element might be styled to support both "open" and "closed" states; if the modal is currently open, Ply will prune all styles pertaining only to the "closed" state. While this allows Ply to prune more aggressively than CDT, it also means that the results cannot be used to refactor stylesheets without human input.[12] However, incorporating visual relevance as a coverage metric within UI testing frameworks could highlight potentially redundant or obsolete CSS for programmer review.

Finally, our work has important implications for web developer training more broadly. Prior work on example-based learning has largely neglected professional examples because they present too much information to be meaningful to learners. However, tutorial examples do not teach the design patterns and problem-solving practices needed to produce professional-quality work. Our results provide preliminary evidence that tools designed to support novice intuition by reducing information and embedding guidance can make learning from professional examples tractable, even for inexperienced developers. Future exploration into the design of learner-centered inspection tools could further bridge the knowledge gap between novice and expert developers.

ACKNOWLEDGEMENTS

We thank Lea Verou, Bert Bos, Michail Yasonik, Jordan Scales, members of the Khan Academy development team, and students in the Design, Technology, and Research program for valuable discussions. We also thank Andrea Cardaci for early technical advice. This work was supported by the National Science Foundation under Grant IIS-1735977, and an Undergraduate Research Grant from Northwestern University.

[12]The general problem of determining whether two stylesheets are observably equivalent under all circumstances is undecidable.

REFERENCES

1. Beth Adelson and Elliot Soloway. 1985. The role of domain experience in software design. *IEEE Transactions on Software Engineering* 11 (1985), 1351–1360.
2. John R. Anderson, Albert T. Corbett, Kenneth R. Koedinger, and Ray Pelletier. 1995. Cognitive tutors: Lessons learned. *The Journal of the Learning Sciences* 4, 2 (1995), 167–207.
3. Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R Klemmer. 2010a. Example-centric programming - integrating web search into the development environment. *CHI* (2010).
4. Joel Brandt, Vignan Pattamatta, William Choi, Ben Hsieh, and Scott R Klemmer. 2010b. *Rehearse: Helping Programmers Adapt Examples by Visualizing Execution and Highlighting Related Code*. Technical Report.
5. 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 and Technology*. ACM, New York, NY, USA, 259–268.
6. Jill Cao, Scott D Fleming, and Margaret Burnett. 2011. An exploration of design opportunities for "gardening" end-user programmers' ideas. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 35–42.
7. 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, New York, NY, USA, 3205–3214.
8. Michelene TH 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.
9. Morgan Dixon and James Fogarty. 2010. Prefab: Implementing Advanced Behaviors Using Pixel-based Reverse Engineering of Interface Structure. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10)*. ACM, New York, NY, USA, 1525–1534. DOI: http://dx.doi.org/10.1145/1753326.1753554
10. Brian Dorn and Mark Guzdial. 2010. Learning on the Job: Characterizing the Programming Knowledge and Learning Strategies of Web Designers. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, New York, NY, USA, 703–712.
11. Claudia Eckert and Martin Stacey. 2000. Sources of inspiration: a language of design. *Design Studies* 21, 5 (Sept. 2000), 523–538.
12. Pierre Genevès, Nabil Layaïda, and Vincent Quint. 2012. On the analysis of cascading style sheets. *WWW* (2012).

13. Paul Gross and Caitlin Kelleher. 2010. Toward Transforming Freely Available Source Code into Usable Learning Materials for End-users. In *Evaluation and Usability of Programming Languages and Tools*. ACM, New York, NY, USA, 6:1–6:6.

14. Matthew Hague, Anthony W. Lin, and C.-H. Luke Ong. 2015. Detecting Redundant CSS Rules in HTML5 Applications: A Tree Rewriting Approach. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. ACM, New York, NY, USA, 1–19. DOI: http://dx.doi.org/10.1145/2814270.2814288

15. Andrew Head, Codanda Appachu, Marti A Hearst, and Bjorn Hartmann. 2015. Tutorons - Generating context-relevant, on-demand explanations and demonstrations of online code. *VL/HCC* (2015).

16. Scarlett R Herring, Chia-Chen Chang, Jesse Krantzler, and Brian P Bailey. 2009. Getting inspired! - understanding how and why examples are used in creative design practice. *CHI* (2009).

17. Joshua Hibschman and Haoqi Zhang. 2016. Telescope. In *the 29th Annual Symposium*. ACM Press, New York, New York, USA, 233–245.

18. Will Jernigan, Amber Horvath, Michael Lee, Margaret Burnett, Taylor Cuilty, Sandeep Kuttal, Anicia Peters, Irwin Kwan, Faezeh Bahmani, and Andrew Ko. 2015. A principled evaluation for a principled idea garden. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 235–243.

19. Matthias Keller and Martin Nussbaumer. 2010. CSS Code Quality: A Metric for Abstractness; Or Why Humans Beat Machines in CSS Coding. In *2010 Seventh International Conference on the Quality of Information and Communications Technology*. 116–121.

20. David Klahr and Kevin Dunbar. 1988. Dual Space Search During Scientific Reasoning. *Cognitive Science* (1988).

21. Andrew J Ko, Brad A Myers, and Htet Htet Aung. 2004. Six Learning Barriers in End-User Programming Systems. In *VLHCC '04: Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing (VLHCC'04)*. Carnegie Mellon University, IEEE Computer Society.

22. Ranjitha Kumar, Arvind Satyanarayan, Cesar Torres, Maxine Lim, Salman Ahmad, Scott R. Klemmer, and Jerry O. Talton. 2013. Webzeitgeist: Design Mining the Web. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '13)*. ACM, New York, NY, USA, 3083–3092.

23. Ranjitha Kumar, Jerry O Talton, Salman Ahmad, and Scott R Klemmer. 2011. Bricolage - example-based retargeting for web design. *CHI* (2011).

24. Bruno Latour. 1990. Drawing things together. In *In Representation in Scientific Practice*, M. Lynch and S. Woolgar (Eds.). The MIT Press, 19–68.

25. Brian Lee, Savil Srivastava, Ranjitha Kumar, Ronen I Brafman, and Scott R Klemmer. 2010. Designing with interactive example galleries. *CHI* (2010).

26. Hsiang-Sheng Liang, Kuan-Hung Kuo, Po-Wei Lee, Yu-Chien Chan, Yu-Chin Lin, and Mike Y Chen. 2013. SeeSS. In *the 26th annual ACM symposium*. ACM Press, New York, New York, USA, 353–356.

27. Marcia C Linn and Michael J Clancy. 1992. The case for case studies of programming problems. *Commun. ACM* 35, 3 (1992), 121–132.

28. Ali Mesbah and Shabnam Mirshokraie. 2012. Automated analysis of CSS rules to support style maintenance. *ICSE* (2012).

29. Steve Oney and Brad Myers. 2009. FireCrystal: Understanding interactive behaviors in dynamic web pages. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 105–108.

30. Pavel Panchekha and Emina Torlak. 2016. Automated reasoning for web page layout. *OOPSLA* (2016).

31. Vincent Quint and Irne Vatton. 2007. Editing with Style. In *Proceedings of the 2007 ACM Symposium on Document Engineering (DocEng '07)*. ACM, New York, NY, USA, 151–160. http://doi.acm.org/10.1145/1284420.1284460

32. 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. *Journal of the Learning Sciences* 13, 3 (July 2004), 337–386.

33. Bruce L Sherin. 2001. How students understand physics equations. *Cognition and instruction* 19, 4 (2001), 479–541.

34. Karl E Weick. 1995. *Sensemaking in organizations*. Vol. 3. Sage.

35. 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.

36. Jenna Wortham. 2012. A surge in learning the language of the internet. *New York Times* 27 (2012).

37. Shin Yoo, David Binkley, and Roger Eastman. 2016. Observational slicing based on visual semantics. *Journal of Systems and Software* (2016).