CSSDEV: Refactoring duplication in Cascading Style Sheets

Davood Mazinanian, Nikolaos Tsantalis
Department of Computer Science and Software Engineering
Concordia University, Montréal, Canada
{d_mazina, tsantalis}@cse.concordia.ca

***Abstract*—Cascading Style Sheets (CSS) is a widely-used language for defining the presentation of structured documents and user interfaces. Despite its popularity, CSS still lacks adequate tool support for everyday maintenance tasks, such as debugging and refactoring. In this paper, we present CSSDEV, a tool suite for analyzing CSS code to detect refactoring opportunities. (https://youtu.be/lu3oITi1XrQ)**

***Keywords*-Cascading Style Sheets, Preprocessors, Refactoring**

I. INTRODUCTION

Cascading Style Sheets (abbr. CSS) is a declarative, domain-specific language, used for defining the presentation of structured documents (e.g., HTML, SVG), as well as desktop and mobile application user interfaces, across different presentation media. Indeed, CSS is extensively used by a large number of software developers [1], [2].

In CSS, developers define the presentation of UI elements (e.g., HTML hyperlinks) by defining *style rules*. Each style rule includes a CSS *selector*, which determines what elements are styled (e.g., `a` selects all HTML hyperlinks). The desired presentation for the selected elements is then achieved by defining *style declarations* (e.g., `border-color: red`) in the style rule. Each style declaration assigns some *style values* (i.e., `red`) to a *style property* (i.e., `border-color`).

Despite this apparent syntactical simplicity [3], developing and maintaining CSS can be challenging, due to several reasons, including but not limited to:

- Some more complicated features of CSS, such as cascading, specificity, value propagation through inheritance, and media queries, make CSS code difficult to comprehend,
- The interplay of CSS with HTML, which can be manipulated by JAVASCRIPT or a server-side language at runtime, makes static analysis tools unable to spot problems at development time,
- The lack of a comprehensive and reliable testing framework for CSS makes regression testing difficult,
- The inherent shortcomings in the design of the language (e.g., the lack of constructs enabling code reuse, such as functions), lead to extensive code duplication. In a previous work, we found that around 60% of style declarations are duplicated in real-world CSS files [4],
- The lack of best practices has led to low quality CSS code suffering from various CSS-specific smells [5], and,
- The standardization of CSS is a time-consuming process, causing incompatible implementations in web browsers, which result in inconsistent presentation (the so-called Cross Browser Incompatibility or XBI [6], [7]).

This intricacy, however, can be diminished to a large extent, in the presence of adequate tool and IDE support. Unfortunately, for CSS development and maintenance, tooling is quite immature and far from being satisfactory for the developers' needs. While CSS is extensively used in the industry, the predominant tool for CSS developers is the web browsers' embedded development facilities (e.g., Firebug in Firefox, Developer Tools in Chrome). In other words, the prevalent workflow for a CSS developer includes 1) coding CSS, 2) running the web application in one (or multiple) web browsers and visually inspecting whether the design is acceptable, 3) using the web browser's development tool, which displays the changes live in the browser, to manipulate CSS style rules until the desired presentation is achieved, and propagating the required changes back to the original CSS files.

While this workflow can definitely aid CSS developers, it suffers from various shortcomings. For instance, the CSS code which is used in development might not be the same code processed by the web browser. For instance, the code could be developed using a CSS *preprocessor* (i.e., a language that generates CSS, like LESS [8] and SASS [9]). In that case, propagating CSS changes from the web browser's development tool to the preprocessor code might not be trivial. More importantly, the embedded tools in web browsers do not offer any support for applying complex changes (e.g., safe refactorings). State-of-the-art IDEs (e.g., Eclipse, JetBrain WebStorm) simply offer syntax highlighting, limited coding assistance with auto-completion, and trivial refactoring support, such as renaming CSS class names. Consequently, there is certainly a need for developing new tools and improving IDE support for CSS development and maintenance.

In previous works, we proposed approaches for safely refactoring duplicated code in CSS, by grouping duplicated style declarations into new selectors [4], or by migrating CSS code to a preprocessor language by extracting function-like constructs (i.e., *mixins*) from duplicated style declarations [10]. The proposed approaches have been implemented in CSSDEV[1], which is an IDE-agnostic CSS analysis and refactoring infrastructure. CSSDEV provides a rich set of APIs that, in addition to refactoring duplicated code, can be used for

[1]In addition to the abbreviation for "developer", Dev is the god of war, and a demon with enormous power, in Persian mythology.

resolving many of the aforementioned challenges encountered when developing and maintaining CSS code. As a proof of concept, we have implemented some key features of CSSDEV for refactoring duplicated code in an Eclipse plug-in, which will be demonstrated in the next sections of the paper.

II. TOOL DESIGN

CSSDEV consists of the following main modules:

CSS Model generator module: It generates a lightweight, hierarchical model of CSS, as described in [4]. This model captures information about CSS code elements, which are crucial for enabling CSS analysis. For instance, for each CSS style declaration, the model captures the type and role of each of the style values (e.g., in style declaration `border: dotted 1em #F0F`, the value `dotted` is a CSS keyword that defines the *style* of the border that appears around an element). It also extracts "hidden" properties that are styled in a style declaration; e.g., the aforementioned `border` declaration implicitly defines style values for 12 individual style properties in total, based on CSS language specifications [11]. Such information is used in detecting dependencies between CSS style declarations. The lightweight model also enables the separation of analysis algorithms from the ASTs generated from CSS parsers. This is crucial, as CSS specifications change rapidly and a parser might become obsolete. Our model provides the necessary abstractions to make easy the replacement of CSS parsers with different capabilities.

Duplication module: It detects different types of duplicated style declarations in CSS, and identifies opportunities for refactoring. In a nutshell, CSSDEV applies a *frequent pattern mining* algorithm, namely FP-Growth [12], on the CSS model to detect and group duplicated declarations [4], [10]. The excessive amount of duplication in real-world CSS files warrants using such an algorithm for ensuring scalability.

Crawler module: It crawls HTML documents for which the CSS file under analysis is used. We use CRAWLJAX [13], a tool for crawling dynamic web applications relying on JAVASCRIPT to handle user interactions.

Dependency module: It is responsible for generating a dependency graph for CSS. The dependencies are extracted and used when refactoring CSS, to make sure that the transformations are *safe* to apply, i.e., the resulting CSS file produces the same presentation after refactoring.

Preprocessor module: It deals with CSS *preprocessor languages*. CSS preprocessors are superset languages for CSS, which support the features that are presently missing in CSS (e.g., functions). The module allows safely migrating existing CSS code to preprocessors by extracting duplicated style declarations to function-like constructs [10] [14]. Several preprocessors have been introduced in the industry, and they have been extensively adopted by developers [15]. The implementation of this module is mostly preprocessor-agnostic, i.e., it can generate transformations for virtually any CSS preprocessor language.

III. TOOL FEATURES

A. Clone Detection

Code duplication exists to a large extent in CSS code [4]. This can result to larger CSS files that have to be transmitted over the Internet, which definitely causes delays in downloading CSS files and rendering web pages. At the same time, excessive file size can hamper maintainability and understandability of CSS code. CSSDEV provides the functionality for detecting three types of equivalent style declarations within CSS code:

Type I declarations that assign the exact same values to the the same style property (e.g., the repetition of the style declaration `color: red`),

Type II declarations that style the same property with *equivalent values* (e.g., `color: red` and `color: #f00`),

Type III equivalent individual and *shorthand* style declarations (e.g., `margin: 0px 2px` is a shorthand style declaration for four individual style declarations `margin-left: 2px; margin-right: 2px; margin-top: 0px; margin-bottom: 0px`).

In addition, CSSDEV identifies duplicated style declarations that are not equivalent, i.e., style declarations with the same property, but with differences in style values. This type of duplication can only be eliminated by extracting a function-like construct, something that CSS currently does not support. Instead, the tool is able to extract a *mixin* for this type of duplication, i.e., a function in a preprocessor language.

The main plugin's view is shown in Figure 1. The user can initiate duplication detection by selecting a CSS file in the workspace, and clicking on the "Detect" command in the view (Figure 1(a)). This can also be done whenever the user saves the CSS file (Figure 1(f)). In either case, the duplicated style declarations are listed in a table, where each of the rows is an opportunity for refactoring (Figure 1(d)). The developer can investigate any opportunity by double clicking on it, which results in highlighting the duplicated style declarations. For each opportunity, the view also shows the type of the duplication, i.e., Type I through III, non-equivalent declarations, or a combination of them (Figure 1(g)).

Moreover, for each refactoring opportunity, the user can see the unique *style property categories* to which the involved style declarations belong (Figure 1(h)). Each category consists of a set of related style properties; for instance, the Text category includes all CSS properties related to text manipulation, e.g., `hyphens`, `text-align` and `word-wrap`. The categories are extracted from the CSS specifications. This information can help the developer to pick the most relevant declarations for refactoring. Intuitively, the opportunity with the smaller number of style property categories is more *coherent*, and should be favored for refactoring. We previously showed that developers tend to group duplicated style declarations that are somewhat coherent, e.g., the ones that style the same properties for different web browsers [14]. Indeed, the plugin's view allows the developer to sort the detected opportunities

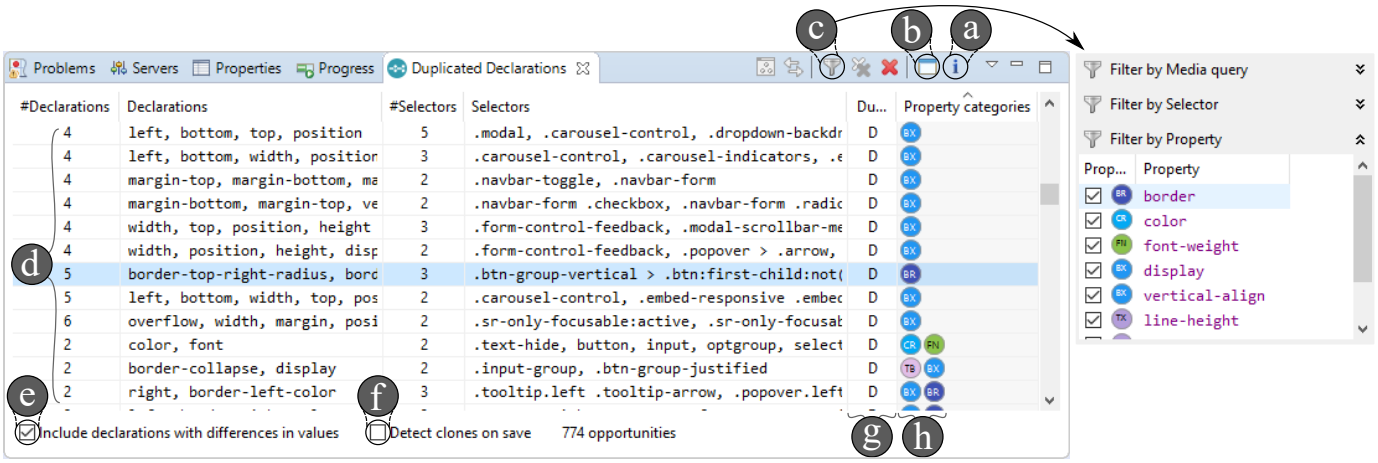

Fig. 1. Duplication View

based on different criteria, including the number of style property categories associated with each opportunity.

The developer also has the option to filter out opportunities, so that only the ones involving specific style declarations and/or selectors are shown (Figure 1c). The developer may also show/hide the opportunities that contain duplicated declarations having differences in their style values (Figure 1e).

B. Extracting Order Dependencies

Normally, the relative order of declarations in a CSS file does not matter, unless there exist *order dependencies* between different selectors [4], which force certain constraints in the selector positions within the CSS file. Order dependencies exist with respect to some target documents (e.g., the HTML documents on which the CSS file is applied). A refactoring that changes the order of style declarations (e.g., extracting a grouping selector for a set of selectors having duplicated style declarations and appending it at the end of the CSS file) might break the presentation of the target documents, if these order dependencies are overlooked and not handled properly.

Order dependencies can be statically extracted from static HTML files that are not manipulated at runtime. However, real-world scenarios are usually much more complex. For instance, in modern web applications, often JAVASCRIPT manipulates the elements of the HTML documents at runtime, through the Document Object Model (i.e., DOM) API (e.g., by adding or removing HTML elements). Thus, a complete CSS analysis tool should deal with this dynamism in order to extract dependencies even from the hidden states of HTML documents. Indeed, it has been shown that, on average, 62% of the DOM states in modern web applications are hidden [16].

CSSDEV uses an automatic crawler, CRAWLJAX [13], for exploring hidden DOM states in web applications. The developer needs to define a starting point for crawling. This could be the address of the first page of a web application hosted locally, or on a web server. The crawler mimics users' behavior by firing events (e.g., mouse clicks) on the HTML pages to explore new states. The developer can define, through a configuration wizard, how the crawling should be performed (e.g., which elements should not be clicked on, or the maximum number of states that should be explored). By default, the crawling is done *blindly* (i.e., the crawler clicks on all elements, even if it does not yield a state change). Thus, the crawling might take several minutes; however, the developer's knowledge of the web pages under analysis can help in providing appropriate values for the crawling options to significantly reduce the crawling time. Note that, the crawling is done in background (i.e., using a headless browser), so that the developer is able to continue working without interruption. Whenever a new state is explored, or the crawling is finished, the developer is notified. When the crawling is done, the developer can apply safe refactorings.

C. Clone Refactoring

Once an opportunity is selected, the developer can initiate a refactoring by right clicking on it. Two scenarios are possible:

1) If the opportunity contains declarations with non-equivalent style values, as mentioned, the refactoring can be done only by extracting a *mixin* in a preprocessor language.
2) Otherwise, the declarations can be grouped in a style rule with a *grouping selector* (i.e., a selector that selects multiple elements, e.g., `table, img` selects both tables and images). Alternatively, a *parameterless mixin* can be extracted from the duplicated declarations.

In the first case, a dialog will be shown (Figure 2), giving the developer the freedom to change several options, including:

a The name of the extracted *mixin*,
b The name of each of the extracted *mixin*'s parameters,
c The selectors from which the *mixin* should be extracted,
d The declarations that the developer wants the *mixin* to include. In other words, the user can select a *sub-opportunity* to be applied, if she finds that some of the declarations suggested by CSSDEV are not coherent enough to be extracted together.

As it can be observed, this dialog also highlights the differences existing between the corresponding style values. Hovering on each style property and value also gives more information about them. For instance, for a style value, the tool displays the role of the value in the style declaration.

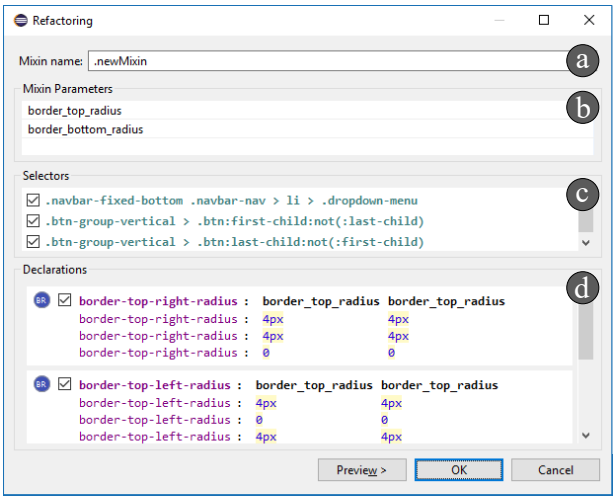

Fig. 2. Refactoring Options Wizard

In case of an opportunity with only equivalent declarations, a similar dialog will be shown, if the developer selects to extract a parameterless *mixin*. However, if she chooses to extract a grouping selector, only options (c) and (d) will be available, as the two first ones are not applicable for a grouping selector (i.e., a grouping selector is automatically named by separating the individual selectors that are grouped by comma, and there are no parameters to name, because there are no differences in style values).

After finalizing the options, CSSDEV checks the refactoring preconditions [4], [10], and generates the actual source code transformations. In some cases, CSSDEV needs to reorder some of the style declarations or selectors, in order to make sure that the changes will preserve the presentation semantics of the resulting code [4], [10]. The developer gets a preview of all the changes (Figure 3). This allows her to perform a final investigation of the changes to be performed. In any case, the IDE allows to undo the changes after a refactoring is applied. We have also implemented the required code for taking advantage of Eclipse Refactoring History feature, so that the developer can keep track of the applied refactorings.

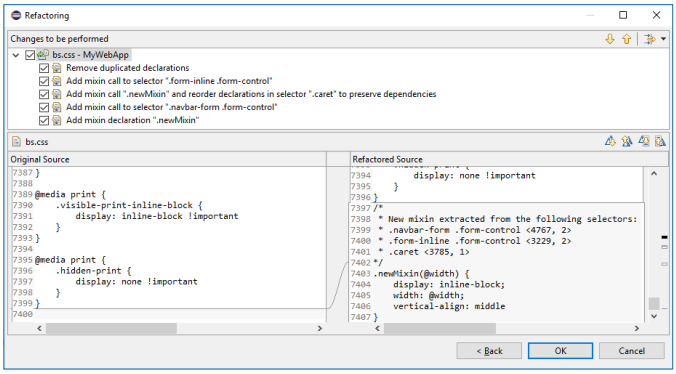

Fig. 3. Refactoring Preview

IV. Validation, Conclusions and Future Work

The soundness of the proposed refactorings has been already validated in previous works [4], [10]. As an ongoing research, we are investigating ways for helping developers in *prioritizing* opportunities for refactoring, e.g., by ranking them based on some criteria, and filtering out the ones that will be unlikely to apply (i.e., those that might actually deteriorate maintainability, rather than improving it). To this aim, we are taking advantage of the knowledge obtained from an empirical study that we conducted to find out how developers use CSS preprocessor language features (e.g., *mixins*) [14]. To further complement this knowledge, we are aiming at conducting a user study, to achieve a deeper understanding of what developers really need when they refactor duplicated code in CSS. This can be done, for instance, by asking the developers to rate the refactoring opportunities proposed by our approach, and seeking their reasoning behind the ratings. We will then look into training statistical models for ranking opportunities, based on the gained knowledge. The developers will also provide feedback on the usability of the CSSDEV Eclipse plug-in and suggest ways to improve it.

References

[1] Web Technology Surveys. Usage of CSS for websites. http://w3techs.com/technologies/details/ce-css/all/all.

[2] Mozilla Developer Network. Shorthand properties. https://developer.mozilla.org/en-US/docs/Web/CSS/Shorthand_properties. Accessed: 26 Dec. 2015. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/CSS/Shorthand_properties

[3] "CSS Syntax Module Level 3," http://www.w3.org/TR/css-syntax-3/, World Wide Web Consortium, Tech. Rep., November 2013.

[4] D. Mazinanian, N. Tsantalis, and A. Mesbah, "Discovering Refactoring Opportunities in Cascading Style Sheets," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2014, pp. 496–506.

[5] G. Gharachorlu, "Code smells in Cascading Style Sheets: an empirical study and a predictive model," Master's Thesis, University of British Columbia, 2014.

[6] S. R. Choudhary, H. Versee, and A. Orso, "WEBDIFF: Automated identification of cross-browser issues in web applications," in *Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM)*, 2010, pp. 1–10.

[7] S. Roy Choudhary, M. R. Prasad, and A. Orso, "X-PERT: Accurate Identification of Cross-browser Issues in Web Applications," in *Proceedings of the 2013 International Conference on Software Engineering (ICSE)*, 2013, pp. 702–711.

[8] A. Sellier. LESS - The dynamic stylesheet language. http://lesscss.org/.

[9] H. Catlin. SASS: Syntactically Awesome Style Sheets. http://sass-lang.com/.

[10] D. Mazinanian and N. Tsantalis, "Migrating Cascading Style Sheets to Preprocessors by Introducing Mixins," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 672–683.

[11] World Wide Web Consortium. CSS specifications. http://www.w3.org/Style/CSS/current-work.

[12] J. Han, J. Pei, and Y. Yin, "Mining Frequent Patterns Without Candidate Generation," *SIGMOD Record*, vol. 29, no. 2, pp. 1–12, 2000.

[13] A. Mesbah, A. van Deursen, and S. Lenselink, "Crawling Ajax-based web applications through dynamic analysis of user interface state changes," *ACM Transactions on the Web*, vol. 6, no. 1, pp. 3:1–3:30, 2012.

[14] D. Mazinanian and N. Tsantalis, "An Empirical Study on the Use of CSS Preprocessors," in *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2016, pp. 168–178.

[15] C. Coyier. Popularity of CSS Preprocessors. http://css-tricks.com/poll-results-popularity-of-css-preprocessors/.

[16] Z. Behfarshad and A. Mesbah, "Hidden-Web Induced by Client-side Scripting: An Empirical Study," in *Proceedings of the 13th International Conference on Web Engineering (ICWE)*, 2013, pp. 52–67.