Eliminating Code Duplication in Cascading Style Sheets

Davood Mazinanian

A Thesis
In the Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of
Doctor of Philosophy (Software Engineering) at
Concordia University
Montréal, Québec, Canada

August 2017

Concordia University

School of Graduate Studies

This is to certify that the thesis prepared

By: **Mr. Davood Mazinanian**

Entitled: **Eliminating Code Duplication in Cascading Style Sheets**

and submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Software Engineering)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

_____ Chair
Dr Jia Yuan Yu

_____ External Examiner
Dr Foutse Khomh

_____ Examiner
Dr Juergen Rilling

_____ Examiner
Dr Wahab Hamou-Lhadj

_____ Examiner
Dr Peter Rigby

_____ Supervisor
Dr Nikolaos Tsantalis

Approved by _____
Dr Volker Haarslev, Graduate Program Director

30 August 2017 _____
Dr Amir Asif, Dean
Faculty of Engineering and Computer Science

Abstract

Eliminating Code Duplication in Cascading Style Sheets

Davood Mazinanian, Ph.D.

Concordia University, 2017

Cascading Style Sheets (i.e., CSS) is the standard styling language, widely used for defining the presentation semantics of user interfaces for web, mobile and desktop applications. Despite its popularity, CSS has not received much attention from academia. Indeed, developing and maintaining CSS code is rather challenging, due to the inherent language design shortcomings, the interplay of CSS with other programming languages (e.g., HTML and JavaScript), the lack of empirically-evaluated coding best-practices, and immature tool support. As a result, the quality of CSS code bases is poor in many cases.

In this thesis, we focus on one of the major issues found in CSS code bases, i.e., the *duplicated code*. In a large, representative dataset of CSS code, we found an average of 68% duplication in style declarations. To alleviate this, we devise techniques for *refactoring* CSS code (i.e., grouping style declarations into new style rules), or *migrating* CSS code to take advantage of the code abstraction features provided by CSS *preprocessor languages* (i.e., superset languages for CSS that augment it by adding extra features that facilitate code maintenance). Specifically for the migration transformations, we attempt to align the resulting code with manually-developed code, by relying on the knowledge gained by conducting an empirical study on the use of CSS preprocessors, which revealed the common coding practices of the developers who use CSS preprocessor languages.

To guarantee the behavior preservation of the proposed transformations, we come up with a list of *preconditions* that should be met, and also describe a lightweight testing technique. By applying a large number of transformations on several web sites and web applications, it is shown that the transformations are indeed presentation-preserving, and can effectively reduce the amount of duplicated code in CSS.

Acknowledgments

First and above all, I praise God, the Almighty, for all the gifts that I had in my life: health, strength, a wonderful family, and all the remarkable people that I was inspired by and learned from.

My deepest gratitude goes to Dr. Nikolaos Tsantalis: thanks for accepting me as your student, believing in me, being always beyond just a supervisor, and helping me to grow with your continuous support and mentorship. I have learned a great deal from you, both as a person, and as a researcher.

I'm grateful for the support of my PhD committee, Drs. Juergen Rilling, Wahab Hamou-Lhadj, Peter Rigby, and Foutse Khomh. Especially, I'm grateful to Dr. Rigby for his outstanding course (Open Source Software and MSR); what I learned there significantly aided me throughout my studies.

I would like to express my gratitude to Dr. Ali Mesbah for his insightful feedback and collaboration on my first paper, Dr. Emad Shihab for exceptionally helpful discussions, and Dr. Danny Dig, for being an inspirational researcher who not only extraordinarily supported me when we collaborated on papers, but also took any opportunity for teaching me true life lessons.

Thanks to all the people who made the time at Concordia fantastically joyful. Especially, my incredible friends, Dr. Laleh Eshkevari, Shahriar Rostami and Matin Mansouri, from whom I learned a lot, together with whom I laughed a lot – and, of course, we also researched a bit!

Saeed Sarencheh, Sultan Wehaibi, Everton Maldonado, Asif AlWaqfi, Moiz Arif, Rabe Abdalkareem, Raphi Stein, Zackary Valenta, Andy Qiao, Giri Krishnan – and several others – thank you very much for all the friendship and support. Thanks also to the friends outside Concordia, including but not limited to Ameya Ketkar, for tirelessly working with me on our OOPSLA paper.

Even if I was a native English speaker, I wouldn't be able to express my feelings about certain people in words. My parents, Mahmood and Parivash: sorry for not being around when you needed me, and thanks for letting me live miles apart from you to follow what I liked the most. My lovely brothers, Danial and Dariush: thanks for being there for our parents instead of me. My brother in law, Mehran: thanks for being the best brother in law ever! And lastly, my wife, Mehrnoosh: this was simply impossible without you. I'm truly blessed to have you in my life. Thank you very much for all the support, kindness, care, understanding, patience, sacrifice, and love.

Contents

List of Figures **ix**

List of Tables **xi**

1 Introduction **1**

1.1 Problem Statement 2

1.2 Thesis Contributions 3

1.3 Thesis Organization 4

1.4 Related Publications 5

2 Background **7**

2.1 The History 7

2.2 The CSS Language 8

2.3 CSS Syntax 9

2.4 Value Propagation 11

2.5 At-rules 13

2.6 CSS Specifications 14

2.6.1 CSS Versions 14

2.6.2 CSS Specification Standardization Stages 15

2.7 Chapter Summary 15

3 Related Work **16**

3.1 The Analysis and Maintenance of CSS Code 16

3.1.1 General Studies 16

3.1.2 Code Quality 17

3.1.3 Alternative Language Proposals for CSS 23

3.1.4 Clone Detection in CSS 24

3.1.5 Migrating CSS to Preprocessor Languages 26

3.1.6 Other Related Studies . . . 26
3.2 Clone Detection in Web Applications . . . 30
3.3 Empirical Studies on Language Features Usage . . . 31
3.4 Automated Code Migration . . . 32
3.4.1 Migration of the Legacy Systems . . . 32
3.4.2 Migration of Web Systems . . . 32
3.5 Chapter Summary . . . 33

4 Refactoring Duplication Within CSS 34
4.1 Introduction . . . 34
4.2 Duplication in CSS . . . 35
4.2.1 Duplication Types . . . 36
4.2.2 Eliminating Duplications . . . 38
4.3 Method . . . 39
4.3.1 Abstract Model Generation . . . 39
4.3.2 Preprocessing . . . 41
4.3.3 Duplication Detection . . . 42
4.3.4 Extracting Refactoring Opportunities . . . 42
4.3.5 Ranking Refactoring Opportunities . . . 46
4.3.6 Preserving Order Dependencies . . . 47
4.4 Evaluation . . . 50
4.4.1 Experiment Design . . . 50
4.4.2 Results . . . 52
4.4.3 Comparison with Federman and Cook's approach [Dav10] . . . 55
4.4.4 Discussion . . . 58
4.5 Chapter Summary . . . 60

5 An Empirical Study on the Use of CSS Preprocessors 62
5.1 Introduction . . . 62
5.2 CSS Preprocessor Features . . . 64
5.2.1 Variables . . . 64
5.2.2 Nesting . . . 64
5.2.3 Mixins . . . 66
5.2.4 The "Extend" Construct . . . 66
5.3 Experiment Setup . . . 67
5.3.1 Subject Systems . . . 68

5.3.2 Data Collection . . . 68
5.4 Empirical Study . . . 72
5.4.1 Variables . . . 73
5.4.2 Nesting . . . 75
5.4.3 Mixin Calls . . . 77
5.4.4 The "Extend" Construct . . . 81
5.5 Threats to Validity . . . 83
5.6 Chapter Summary . . . 83

6 Migrating CSS to Preprocessors by Introducing *mixins* 85
6.1 Introduction . . . 85
6.2 Abstraction Mechanisms in CSS Preprocessors . . . 86
6.3 Automatic Extraction of Mixins . . . 87
6.3.1 Grouping Declarations for Extraction . . . 87
6.3.2 Detecting Differences in Style Values . . . 89
6.3.3 Introducing a Mixin in the Style Sheet . . . 94
6.3.4 Preserving Presentation . . . 98
6.4 Evaluation . . . 101
6.4.1 Experiment Design . . . 101
6.4.2 Results . . . 104
6.4.3 Limitations . . . 108
6.5 Comparison with Charpentier et al.'s Approach [CFR16] . . . 109
6.5.1 Summary of the Method . . . 109
6.5.2 Comparison . . . 109
6.6 Threats to Validity . . . 114
6.7 Chapter Summary . . . 115

7 CSSDev: A tool suite for the analysis and refactoring of CSS 116
7.1 Introduction . . . 116
7.2 Tool Design . . . 118
7.2.1 CSS Model Generator Module . . . 118
7.2.2 Duplication Module . . . 119
7.2.3 Crawler Module . . . 119
7.2.4 Dependency Module . . . 119
7.2.5 Preprocessor Module . . . 119
7.2.6 Refactoring Module . . . 119

7.3 Tool Features 120
7.3.1 Clone Detection 120
7.3.2 Extracting Order Dependencies 121
7.3.3 Clone Refactoring 124
7.4 Chapter Summary 126

8 Conclusions and Future Work **127**
8.1 Summary of the Findings 127
8.2 Future Work 129
8.2.1 Current Limitations 129
8.2.2 Other Possible Opportunities for Future Research 130

Bibliography **132**

List of Figures

1 Basic rule syntax in CSS . . . 9
2 Class selectors. . . . 10
3 Use of `@media` at-rules in CSS . . . 13
4 Type I duplication in Gmail's CSS . . . 37
5 Declaration duplication in Gmail's CSS . . . 38
6 Shorthand and individual declarations . . . 38
7 Grouping style declarations to remove duplication . . . 39
8 A hierarchical object model for CSS . . . 40
9 Clones extracted from a style sheet . . . 42
10 Dataset for the style sheet of Figure 9 . . . 44
11 FP-Tree for the dataset of Figure 10 . . . 45
12 Output of the FP-Growth algorithm for the style sheet of Figure 9 . . . 46
13 Order dependencies before and after refactoring . . . 48
14 Breaking presentation semantics with improper refactoring . . . 49
15 Characteristics of the analyzed CSS files . . . 51
16 Ratio of the duplicated declarations . . . 53
17 Statistics for the detected clones . . . 53
18 Initial refactoring opportunities vs. applied presentation-preserving refactorings . . . 54
19 Order dependencies and size reduction . . . 54
20 Variables in Less . . . 65
21 Nesting in Less . . . 65
22 Mixin in Less . . . 66
23 Extending style rules in Less . . . 67
24 Characteristics of the analyzed preprocessor files . . . 70
25 The workflow applied for the collection and analysis of the experimental data . . . 70
26 Variable types distribution (numbers represent percentages) . . . 76
27 Nesting depth . . . 77

28 Number of *mixin* calls . . . 78
29 Number of property declarations inside *mixins* . . . 79
30 Number of *mixin* parameters . . . 79
31 Parameter reuse across vendor-specific properties . . . 80
32 Percentage of websites using *extend* or parameterless *mixins* . . . 82
33 Mixin example . . . 88
34 Alternative ways for extracting a *mixin* . . . 90
35 Mixin for shorthand/individual properties . . . 94
36 Merging *mixin* parameters . . . 97
37 Intra-style rule order dependencies . . . 99
38 Characteristics of the analyzed CSS files . . . 104
39 Example of interpolated property name . . . 107
40 Example of `!important` use in arguments . . . 107
41 Comparison of the time taken for detecting migration opportunities . . . 114
42 Duplication view . . . 120
43 Crawler settings . . . 122
44 Overriding dependencies View . . . 123
45 Affected DOM elements view . . . 124
46 Refactoring options wizard . . . 125
47 Refactoring preview . . . 126

List of Tables

1 Different representations for the `rebeccapurple` color. 37
2 Selected subjects 51
3 Statistical model's estimated parameters 56
4 List of the websites used in the study 69
5 Overview of the collected data 71
6 Scope of variables 74
7 Categorization of value types 75
8 Use of *nesting* 76
9 Frequent itemsets of style properties 89
10 Individual Style Properties (ISPs) 92
11 List of the supported style properties 93
12 Overview of the collected data 103
13 Threshold-based filtering of opportunities 108

Chapter 1

Introduction

Cascading Style Sheets (henceforth, CSS) is the *lingua franca* for styling: it is extensively used for defining the presentation semantics (e.g., layout and typography) of user interfaces of web, mobile and desktop applications. CSS was originally designed for defining how structured documents developed using a markup programming language (e.g., HTML or SVG) should be presented. It is widely used in today's web development – over 90% of web developers use CSS [Moz10] in 90% of the web sites [Web16]. CSS is also increasingly used in mobile app development through frameworks (e.g., Apache Cordova, Ionic) that generate *hybrid* apps, i.e., mobile apps that look like native applications on mobile devices, yet they are actually developed using client-side web languages (HTML, CSS and JavaScript). There are also similar frameworks for developing *desktop* applications (e.g., Electron) which have resulted into a plethora of successful cross-platform applications (e.g., R-Studio, Atom test editor, Visual Studio Code). *Progressive web apps*, in which CSS plays a crucial role, are also gaining momentum: *web applications* that are installable on mobile devices, are connection-independent (i.e., they can run offline or on low-bandwidth connections), and can access to resources of mobile devices which were formerly allowed only in native apps (e.g., sending push notifications). As a result, CSS has become an important language with applications in many different domains. This has made CSS one of the most-used programming languages in the industry. In 2014, GitHub repositories containing CSS code outnumbered the ones containing PHP, Ruby and C++ [Car14].

Indeed, CSS now plays a vital role in businesses by directly affecting the perceived user experience of their online presence. At Dropbox, for example, there existed around 1200 CSS files (and other files that were used to generate CSS), exceeding 150K line of code. In one incident, a change in some of these files unknowingly broke the presentation of a revenue-generating page that the developers were not aware it even existed, and lack of adequate tools resulted to concealing the fact that the critical page depended on the modified CSS file. This could even damage Dropbox's professional

relationship with the business partner that relied on the broken page [Ede14].

CSS can frequently undergo maintenance activities. Boryana and Zaytsev found that in the course of January to April 2015, there were 2,282,788 commits pushed to GitHub where a CSS code was modified. While CSS has a relatively simple syntax [Con13], some of its complex features (e.g., inheritance, cascading, specificity, initial values [Lie05, Wor17]), its interplay with HTML and JavaScript, and the inherent inadequacy of code reuse mechanisms, make both the development and maintenance of CSS code cumbersome tasks [MM12] for developers. In addition, despite the popularity, CSS has received a very limited attention from academia, especially from the software engineering research community. This explains why CSS development is far from a rigorous and disciplined process, and lacks established design principles and effective tool support [GLQ12].

1.1 Problem Statement

Mature programming languages often provide a comprehensive list of language constructs which allow code reuse, e.g., functions and variables. There exist complementary mechanisms for code reuse in different programming paradigms. Object-oriented programming languages, for instance, offer *inheritance*, *object composition*, *mixins*, and *traits*.

In CSS, however, abstraction constructs are inadequate, immature, or even nonexistent. For example, there is no notion of functions in CSS. At the time of writing this thesis, the CSS specifications for variables (called custom properties in CSS), are in the *candidate recommendation* stage, meaning that variables are still considered as an experimental feature [Con15], and have not been fully-supported by some web browsers yet.

Consequently, duplicated code exists to a large extent in CSS. Nevertheless, no studies have looked into the *existence of duplicated code* in CSS, and *how it can be refactored*.

> **Thesis Statement 1**: Code duplication is prevalent in CSS, and refactoring can be a viable solution for eliminating duplication in CSS.

For avoiding duplicated code in CSS, developers sometimes tend to use a higher-level programming language that supports more mature abstraction mechanisms. CSS *preprocessors* have emerged as the de-facto solution to this aim: superset languages for CSS that augment it by adding constructs that facilitate code reuse (e.g., function-like constructs, which are called *mixins* in the preprocessors jargon). The code written in a CSS preprocessor is compiled (more precisely, *transpiled*) to pure CSS. The use of CSS preprocessors is a trend in the industry [Coy12, Uni15], and leading web companies have already adopted them. Some examples of popular preprocessors are Less, Sass,

Google Closure Style Sheets, and Stylus. However, it is unknown to us how the features of CSS preprocessors that do not exist in CSS are utilized by developers. In other words, we don't know what developers want to achieve when using CSS preprocessors that they can't easily fulfill in "vanilla" CSS (e.g. eliminating duplicated code using function-like constructs).

> **Thesis Statement 2**: CSS developers use CSS preprocessors to a large extent to avoid duplicated code (among the other goals).

Despite the gradual adoption of preprocessors in the web development community, there is still a large portion of front-end developers and web designers using solely "vanilla" CSS. An online poll with nearly 13,000 responses from web developers [Coy12] revealed that 46% of them develop only in "vanilla" CSS, mostly because they are not aware of preprocessors. Therefore, there is a large community of web developers that could benefit from tools that help them in automatically *migrating* their "vanilla" CSS code to a preprocessor of their preference. Specifically, function-like constructs in CSS preprocessors can be beneficial in eliminating duplicated code.

> **Thesis Statement 3**: Migration of CSS code to take advantage of function-like constructs in CSS preprocessors can be automated.

In the next section, we briefly describe what we are going to do in this thesis, in order to support the mentioned thesis statements.

1.2 Thesis Contributions

In this thesis, we make the following contributions:

- We study the problem of duplicated code in CSS and report to what extent it exists in the CSS code base of several web sites / web applications (Chapter 4).
- We define three types of duplication in CSS (that can be eliminated *within* CSS, i.e., by refactoring using a built-in CSS construct), and propose an efficient technique for detecting the instances of these three types of duplicated code (Chapter 4).
- We introduce an approach for refactoring the instances of the mentioned three types of duplicated code in CSS. We discuss how we can be sure that the proposed transformations are *presentation-preserving* (i.e., the behavior of the CSS code remains unchanged after refactoring), by providing a list of *safety preconditions* for the refactoring (Chapter 4).

- We empirically assess how developers take advantage of preprocessor languages (Chapter 5), in order to:
  - Gain an understanding of developers' practices that can help when developing automatic techniques for migrating CSS to preprocessor languages, so that the resulting code will look closer to what developers manually write,
  - Aid developers to take full advantage of CSS preprocessors, by spotting the features that are underused by developers, and
  - Reveal opportunities for CSS preprocessor language designers to revisit the design of these languages, e.g., by adding support for new features (which are currently implemented by developers in an ad-hoc manner), or making existing features easier to use, or eliminating features that are not adopted by developers.
- We propose a technique for identifying the instances of duplicated code in CSS that can be refactored to take advantage of the function-like constructs (the so-called *mixins*) in CSS preprocessor languages (i.e., migrating CSS code to preprocessors). These instances include the ones that can be refactored within CSS, however, there are instances that can be eliminated *only* by using a CSS preprocessor, which will be explained. We then discuss different ways for refactoring these instances, and also introduce techniques for testing the safety of the applied transformations (Chapter 6).
- We provide details about the implementation of the proposed techniques for refactoring and migration of CSS in a comprehensive tool suite and an Eclipse plug-in (Chapter 7).

1.3 Thesis Organization

To better understand the rest of this thesis, in Chapter 2, we will provide background information about the history and syntax of CSS, how it interacts with HTML and JavaScript, how web browsers read and understand CSS code and render web pages, and how the core features of CSS (e.g., cascading, specificity, inheritance) work. In Chapter 3, a summary of related works will be given. The works related to CSS itself are scarce in the literature. However, as we are dealing with *clone refactoring* for CSS, we will briefly cover the related works from the clone community. Also, we will look at empirical studies that aimed at understanding how developers use language features, i.e., the ones that have similar goal to our empirical study on the use of CSS preprocessors but in other programming languages. Finally, we will review a few works that devised similar techniques for *migrating* code written in various traditional programming languages.

Chapters 4, 5, 6 and 7 are dedicated to the main contributions of this thesis, which were mentioned earlier. The conclusions and some promising avenues for future work are discussed in Chapter 8.

1.4 Related Publications

Earlier versions of the work done in this thesis have been published in the following papers:

1. **Davood Mazinanian**, Nikolaos Tsantalis, and Ali Mesbah, "Discovering Refactoring Opportunities in Cascading Style Sheets," in the Proceedings of the 22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE), pp. 496-506, 2014.

2. **Davood Mazinanian**, and Nikolaos Tsantalis, "An empirical study on the use of CSS preprocessors," in the Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER), 2016. [**Best paper candidate award**]

3. **Davood Mazinanian**, and Nikolaos Tsantalis, "Migrating Cascading Style Sheets to Preprocessors by Introducing Mixins," in the Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), 2016.

4. **Davood Mazinanian**, "Refactoring and Migration of Cascading Style Sheets (Towards optimization and improved maintainability)," The 24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE), Doctoral Symposium Track, 2016.

5. **Davood Mazinanian**, and Nikolaos Tsantalis, "CSSDev: Refactoring duplication in Cascading Style Sheets," The 39th International Conference on Software Engineering (ICSE), Demonstrations Track, 2017.

The following papers were published in parallel to the abovementioned publications. While they are not directly related to this thesis, at the same time, they are not completely irrelevant, as their topics include clone refactoring (although for Java systems), improving the maintainability of programs written in another web language (i.e., JavaScript), and empirical studies on how developers use a newly-introduced language feature (i.e., lambda expressions in Java 8).

6. **Davood Mazinanian**, Ameya Ketkar, Nikolaos Tsantalis, Danny Dig, "Understanding the use of lambda expressions in Java," The 2017 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'17).

7. Laleh Eshkevari, **Davood Mazinanian**, Shahriar Rostami, and Nikolaos Tsantalis, "JSDeodorant: Class-awareness for JavaScript programs," The 39th International Conference on Software Engineering (ICSE), Demonstrations Track, 2017.

8. Nikolaos Tsantalis, **Davood Mazinanian**, and Shahriar Rostami, "Clone Refactoring with Lambda Expressions," The 39th International Conference on Software Engineering (ICSE), 2017. [**Distinguished paper award**]

9. Shahriar Rostami, Laleh Eshkevari, **Davood Mazinanian**, and Nikolaos Tsantalis, "Detecting Function Constructors in JavaScript," in the Proceedings of the 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME) - Early Research Achievements Track, 2016

10. **Davood Mazinanian**, Nikolaos Tsantalis, Raphael Stein, and Zackary Valenta, "JDeodorant: Clone Refactoring," 38th International Conference on Software Engineering (ICSE), Formal Demonstrations Track, Austin, Texas, USA, May 14-22, 2016.

11. Nikolaos Tsantalis, **Davood Mazinanian**, and Giri P. Krishnan, "Assessing the Refactorability of Software Clones," IEEE Transactions on Software Engineering, vol.41, no.11, pp.1055-1090, Nov. 1 2015

Chapter 2

Background

In this section, we briefly talk about the history of CSS language, its syntax and semantics, and how it interacts with HTML and JavaScript. Knowing the history of CSS will help to further understand the importance of CSS and its role in shaping the current state of practice in web (and, as mentioned, mobile and desktop) application development.

As it will be discussed, the syntax of CSS is simple, but its semantics are more complicated to capture. Understanding the semantics of CSS is crucial for defining safety preconditions which guarantee that the transformations proposed in this thesis are presentation-preserving.

2.1 The History

As a crucial part of the World Wide Web, the Hyper Text Markup Language (i.e., HTML) was proposed by Tim Berners-Lee, which was initially a simple structured document format: the *markup tags* defined the *role* of the text in the document, in the form of HTML elements (e.g., paragraphs were enclosed in the `<p></p>` tags, and there were other tags for headings, hyperlinks, etc.). In HTML, however, there was no way to indicate how each element should be presented in the web browser. The presentation semantics of each element (e.g., with what font a heading or a piece of normal text should be displayed) was therefore determined solely by the web browser.

To address this deficiency, *presentational tags and attributes* were introduced in HTML. For example, the `<b></b>` tag was used for making a piece of text (enclosed in the tag) bold-faced, or the `<blink></blink>` tag for creating blinking texts.

However, the decision of having the means for defining presentation semantics inside HTML code hindered *content re-usability*. Web browsers are not the only media on which an HTML document could be displayed. One might want to *print* the document, or display it on a wearable device, or – in case of a visually-impaired person – use a text-to-speech device to read the document. In all

these situations, the same content should be logically re-used, but with different presentation on each presentational medium.

This issue led the people involved in the standardization of web technologies to agree that there is a need for adapting *style sheets* for web. In the traditional typography and printing industry, style sheets were used as *guidelines for consistent presentation of documents*. Similarly, style sheets on web would allow the creators and authors of web pages to define a consistent presentation for multiple presentation devises (i.e., content re-usability). More importantly, one set of style sheets could be re-used for multiple web pages, enabling *style re-usability*.

Among the several proposals for a style sheet standard for web, "Cascading HTML Style Sheets" by Håkon Wium Lie [Lie94] was gained a momentum, which later became the Cascading Style Sheets (CSS) which is now the *lingua franca for styling*. A detailed discussion of the history of CSS (and other proposals for style sheets in web) can be found in Lie's doctoral thesis [Lie05]. In the next section, we briefly explain the syntax and semantics of CSS.

2.2 The CSS Language

CSS is a style sheet language and is used to define the presentation semantics of structured documents (predominantly HTML and SVG). In this thesis, we call the structured documents on which CSS code is applied *target documents*. Henceforth, the terms CSS and *style sheets* might be used interchangeably in this thesis (while CSS is a specific language for creating style sheets, the extensive prevalence of CSS makes this decision natural).

The way CSS works is quite simple: CSS code is attached to the target document(s), and styles are defined for one or a group of elements of the target documents. There are three ways to attach CSS code to HTML files:

Inline CSS code is defined inside the `style` attributes for each HTML tag that needs to be styled (e.g., `<p style="...CSS style declarations...">...styled paragraph...</p>`).

Internal CSS code is defined inside the `<style>` tags in the HTML files. The `<style>` tags can appear almost everywhere in the HTML file, but developers usually put them close to the beginning of the file, inside the `<head>` tag.

External CSS code is defined in external files, and the files are attached to HTML pages using the CSS files' URLs declared in the `<link />` tags in the HTML files (the `<link />` tags are in turn defined in the `<head>` tag of the page).

For example, `<link href="theme.css" rel="stylesheet" />` attaches the CSS file `theme.css` to the enclosing HTML file.

As CSS is supposed to enable the separation of presentation from the content and structure, it is usually advisable to use *external* CSS files. The two other options – inline and internal – actually hamper style re-usability, because the styles defined inside one HTML document cannot be re-used in other HTML documents, whereas one CSS file can be attached to multiple HTML pages to define a consistent presentation for them.

2.3 CSS Syntax

In its core, CSS code is composed of a set of CSS *style rules*, when used either as *internal* or *external* CSS (for inline CSS code, the syntax is different, as we will see shortly). The syntax of a style rule in CSS is displayed in Figure 1.

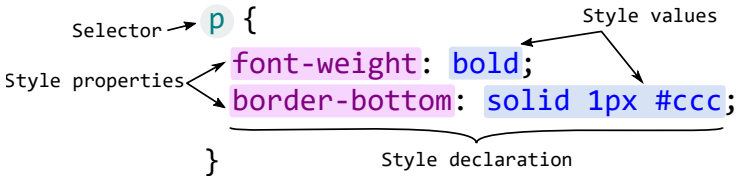


Figure 1: Basic rule syntax in CSS

As it is observed, the core CSS has a very simple syntax. The *selector* part in the CSS rule defines the elements of the target document on which the styles should be applied. For example, a `p` selector selects every paragraph element `<p>` in the document. The actual styles that will be applied on the selected elements are defined using one or more *style declarations*, declared inside the body of the style rule.

Each style declaration is in turn composed of a *style property* and one or more *style values*. The number of values is different for each style property. For example, style property `text-align` is single-valued and its only possible value should be one of the pre-defined values `start`, `end`, `left`, `right`, `center`, `justify`, and `match-parent`. In contrast, the style value `border` is a multi-valued style property, accepting three values that define the style (e.g., `solid` or `dotted`), thickness (e.g., `2px`), and color (e.g., `rgb(255, 128, 0)`) of the border appearing around the selected element. In CSS, style values are often separated by whitespaces, except for a few exceptions.

Note that, in case of *internal* and *external* CSS, we need CSS selectors to define what elements we want to apply styles on. However, in case of *inline* CSS, we mentioned that the CSS code is written inside the `style` attribute of each element. As a result, there is no need for selectors, thus, only style declarations appear inside the `style` attributes.

Note also that, the proposed techniques in this thesis focus only on the *external* CSS, as it is the advisable form of using CSS. All of the proposed techniques will also work for *internal* CSS.

CSS Selectors

CSS offers a comprehensive list of selectors, which we briefly discuss in this section. Understanding how selectors work is important to reveal the intricacies associated with detecting the *dependencies* between CSS style rules (e.g., when two style rules have – possibly lexically-different – selectors that select the same elements of the target documents, and assign values to the same style properties for the selected elements, there is a dependency between these two style rules). As we will see, dependencies between style rules can affect the refactorings done on the CSS code.

The universal selector (i.e., `*`): Selects all the elements of the target document). It is sometimes used with combinators (which will be discussed shortly), so that you can, for instance, select all the elements which are the children of a specific element.

Type selectors: A type selector selects elements of a specific tag. For example, as it is also mentioned earlier, selector `p` selects all the paragraph tags (`<p>`).

Class selectors: A group of declarations could be defined in a style rule for which the selector is a *class selector*. It is then possible to apply the same class to many different elements thus avoiding the duplication of declarations. Figure 2 shows an example of a class selector.

HTML	CSS

`<div class="class1">`
`  content`
`</div>`
`<span class="class1">`
`  content`
`</span>`
	`.class1 {`
	`  color: red;`
	`  font: 10pt tahoma;`
	`}`

Figure 2: Class selectors.

ID selectors: It is possible to set an ID for elements in the target documents, using the ID attribute of each element. For example, the HTML tag `<div id="toolbar"/>` corresponds to a `div` element for which the ID is equal to `"toolbar"`. The `#toolbar` CSS selector, which is called an *ID selector*, can be used to select the element(s) with this ID.

Attribute selectors: We can also add *attribute conditions* to a selector. If selector `S` selects a set of elements S, a selector of the format `S[attr operator value]` selects the subset elements of S for which, in the target document, attribute `attr` is defined and set to a substring of `value`. The `operator` defines the condition for the substring [Wor17]. For example, `a[target]` selects all `<a>` elements that have the `target` attribute set to any value, and `img[src $= ".png"]` selects all `<img>` elements that have the ".png" suffix in their `src` attribute value.

Pseudo Classes and Pseudo Elements: *Pseudo classes* can be used to filter the elements selected by a given selector. For example, `:not(div)` selects all elements except for `div` elements. There are

also *structural pseudo classes*, such as `tr:nth-child(2n+1)` which selects every odd row (denoted by `<tr>` tags in HTML) in every table (the `<table>` tag) of the target document.

Pseudo-elements create abstractions about elements in the target document, beyond those specified by the HTML standards [Wor17]. The `p::first-line` selector, for example, selects only the first line of the text inside every `<p>` element.

Combinators: We can *combine* various selectors to achieve more specific selectors, using different *combinators*. Assuming we have two selectors A and B, we can combine them as follows:

A B (descendant combinator) selects all elements selected by B, which are descendants of the elements selected by A.

A > B (child combinator) selects all elements selected by B, which are **direct** children of the elements selected by A.

A ~ B (general sibling combinator) selects all elements selected by B, which have an element selected by A as a sibling.

A + B (adjacent sibling combinator) selects all elements selected by B, which are directly preceded by a sibling element selected by A.

Grouping selectors (also known as *selector lists* or *multiple selectors*): Finally, there is the possibility of *grouping* different selectors in CSS. For instance, if we want to declare the same style declarations for all `h1` and `h2` HTML elements, we could use the `h1, h2` CSS selector. Note that, this allows re-using style declarations and thus avoids code duplication. In fact, the refactoring proposed in Chapter 4 uses grouping selectors to eliminate duplicated code in CSS.

2.4 Value Propagation

After the elements of target documents are selected using CSS selectors, the web browser tries to assign values to each style property of the selected elements. Knowing the way web browsers assign values to style properties is important, as it helps in understanding the later discussions which prove that the transformations introduced in this thesis are presentation-preserving.

For each target document being rendered in a web browser, style sheets come from the following sources (i.e., the style origins):

1. Each web browser has some embedded style sheets, called *user-agent style sheets*, that define the default style values for some CSS properties when there is no attached style sheet to the target document being rendered, or for the style properties for which the attached style sheets do not define values.

2. Web browsers also allow end users to make custom style sheets that override the user-agent style sheets (i.e., the *user style sheets*).

3. The CSS code that is attached to the target documents (also known as *author style sheets*). The author style sheets can be *external*, *embedded* or *inline*, as mentioned before.

For each selected element, there could be style declarations with the same style properties with values assigned to them. These conflicting style declarations can come from each of the mentioned origins. The web browser prioritizes the style declarations coming from each of these origins in the order given here (i.e., the so-called *cascading rules*). As a result, if an author style sheet defines value for the `font` property, it will override the possible definitions of the same property in the user and user-agent style sheets. For conflicting style declarations defined in author style sheets, web browsers give priority to the style rules declared in the inline style sheets, then embedded style sheets, and then external style sheets.

What if there are conflicting style declarations within the same external CSS file? In such cases, the *selectors specificities* of the style rules enclosing the conflicting style declarations determine the "winning" declaration, i.e., the more specific selector has priority over the less specific ones [Wor17]. For example, selector `.menu a` selects all the hyperlinks (denoted by `<a>` tags in HTML) which are the children of the elements with class `.menu` (e.g., `<div class="menu">...</div>`). This selector is more specific than selector `a`, which selects all hyperlinks regardless of their location in the target documents. As a result, any declaration in `.menu a` will override the declarations with the same property in `a`.

When for two conflicting style declarations the style rules' origins and selectors specificities are the same, the position of the style rules in the CSS file determines the winning declaration (i.e., the last declaration overrides the previous ones).

It is also possible to add the `!important` annotation as the last style value in a style declaration. Using `!important` will *invert* the specificity calculations, in order to make a balance between the ability of users and developers in overriding style declarations. For example, a user of a web page can create a style sheet which overrides the author style sheet, i.e., inverting the priority of user and author style sheets. Interestingly, unjustified use of the `!important` keyword is known to be a code smell in CSS [Gha14].

For some specific style properties (e.g., `color`, `font`), CSS supports *inheritance* in values by taking advantage of the hierarchical structure of target documents. For example, if we apply `color: blue` to the `<body>` element, all child elements of the `<body>` tag will be automatically styled with the blue color. For other properties where value inheritance is not applied automatically, one might explicitly enforce it by replacing the style values with `inherit` (e.g., `margin: inherit`).

Finally, if none of the mentioned methods does not yield a value for a specific style property, the web browser will assign the *initial* value to the style property, as defined in CSS specifications.

2.5 At-rules

There are specific statements in CSS called *at-rules*, which start with `@` (Unicode `U+0040`). At-rules can appear in embedded or external style sheets, just like normal style rules. Some examples of at-rule statements in CSS include `@font` (which allows to specify the characteristics of external fonts), `@import` (which allows to import styles from one CSS file into the other), `@keyframes` (for defining animations), and the `@media` at-rule, which allows to define style rules for specific media (e.g., printers, mobile and wearable screens).

The `@media` at-rule plays a crucial role when creating *responsive* user interfaces, i.e., user interfaces that can be adapted to different presentation media, even on-the-fly (consider a web page being displayed on a mobile phone, and the user rotates her screen to display the website in the landscape mode. The web browser can switch between style rules based on the conditions defined in the `@media` at-rules). An example of a `@media` at-rule is shown in Figure 3.

```
@media print {
  body {
    font-size: 12pt
  }

  input  {
    display: none
  }
}

@media screen {
  body {
   font-size: 13px
  }
}

@media screen, print {
  body {
    line-height:  1.3
  }
}

/* Retina iPad in Portrait and Landscape modes */
@media only screen
  and (min-device-width: 768px)
  and (max-device-width: 1024px)
  and (-webkit-min-device-pixel-ratio: 2) {
    body {
      line-height: 1.4
    }
```

Figure 3: Use of `@media` at-rules in CSS

In Figure 3, the first `@media` at-rule defines style declarations for the document when it is printed out. In this case, the font for the entire document is set at 12 points, and all the `<input>` tags are

set not to be displayed. Here, the developer did not want to show elements such as text boxes in the printed version of the document. The next `@media` at-rule defines the font of the text to be 13 pixels for the entire document on all screens. The line height of the text (i.e., the spacing between the lines) is set to be 1.3 both for screens and printers. However, the last `@media` overrides the value for the line height for specific devises with higher pixel densities (e.g., Retina iPads) to be 1.4. If the developer omitted the last `@media` at-rule, the text would be less readable, since the lines would be displayed closer together.

It is worth mentioning that, in all the proposed techniques in this thesis, style declarations defined in the `@media` at-rules are considered and duplicated style declarations are extracted having the `@media` at-rules in mind, as we will see in the next sections.

2.6 CSS Specifications

For anyone who deals with CSS, especially for the researchers working on developing analysis tools for it, it is crucial to know about the way CSS is standardized. The standards can change rapidly and new features (e.g., new style properties) can be introduced regularly. Consequently, a complete CSS analysis tool should be able to keep up with these quick changes to the specifications.

2.6.1 CSS Versions

The standardization of CSS is led by the CSS working group (i.e., the CSSWG) in the World Wide Web Consortium (i.e., W3C). The CSSWG's members are from browser vendors, several leading companies in the software industry, and independent CSS experts.

The versions of CSS specifications are called *levels*. At first, all the specifications of CSS were a monolithic unit. Since the monolithic specifications were difficult to maintain and develop, after CSS specifications level 2.1, the members of the CSSWG decided to divide the specifications into several *modules* (e.g., the `Backgrounds and Borders` or the `Text` modules). Each module follows its own development path; consequently, the term CSS3 is rather loose, since there is no single, monolithic specification for CSS3. At the time of writing this thesis, there are modules in CSS that are at the final stage of development in Level 3 (e.g., the `Backgrounds and Borders Module Level 3`), while some other are at the earlier stages (e.g., the `Animations Module Level 3`). There are even modules at the fourth level in their early development stages, e.g., the `CSS Selector Modules Level 4`.

2.6.2 CSS Specification Standardization Stages

The members of the CSSWG first create an *editor's draft* of the specifications when a change is proposed. When the specifications are ready from the internal members' point of view, it is published to the public as a *working draft*. The external bodies can then discuss it and comment on the possible difficulties of its implementation, and several versions of the working draft are created. The working draft specifications can be even rejected. Otherwise, the specifications reach the *First Public Working Draft*. The CSSWG will continue working on this version, and finally a deadline is set so that all the comments are collected from the involved parties, i.e., the *Last Call Working Draft* is created.

When specifications are thoroughly tested by the CSSWG and browser vendors, the specifications reach the *Candidate Recommendation* stage. To continue to the next stage (i.e., the *Proposed Recommendations*), the CSSWG demonstrates two correct implementations of the specifications. Then, a higher-level committee in the W3C organization (namely, the W3C Advisory Committee) decides whether the specifications can be elevated to the final stage of *Recommendation*.

Note that, the *Recommendation* stage does not mean that the specifications are *stable*. Web browsers start implementing the specifications at the *Candidate Recommendation* stage, therefore, CSS analysis tools should start supporting the new features from that point. The *Recommendation* stage in fact means that the specifications are *dead*, because there will be still errors in the specifications that are not going to be fixed in that *level* of the specifications. Instead, the CSSWG prefers to start the new level of the specifications (e.g., from level 3 to level 4), and this usually happens even before the specifications are at the *Recommendation* stage.

2.7 Chapter Summary

In this chapter, we provided a brief history of CSS, explained the syntax of its core, (partially) talked about its semantics, and discussed how the standardization process of CSS works. We will frequently refer to this chapter in Chapters 4, 5, 6, and 7, since the provided information will be extensively required.

In the next chapter, we look at the previous studies related to this thesis.

Chapter 3

Related Work

There has not been much work done on CSS (and, in general, style sheet languages) in the academic literature, and this is despite the widespread adoption of CSS in practice. It is safe to conclude that "style sheet languages are terribly underresearched". Surprisingly, this is stated in 1999 by Philip M. Marden and Ethan V. Munson [JM99], and this statement is still perfectly valid after almost two decades.

In this chapter, we have categorized the related works into four sections. First, we look at the few works done on the analysis and maintenance of CSS code. Then, since the goal of this thesis is to study the problem of code duplication in CSS, we briefly summarize the works done in the "software clones" community. Among the plethora of studies in this domain, we only focus on the works done specifically for web languages, since they are more related to the topic of this thesis.

Next, as we have conducted an empirical study on the use of CSS preprocessors, we discuss some of the studies that have similar goals to ours, yet in other domains (e.g., other programming languages and paradigms). Finally, considering that we have proposed an automatic and presentation-preserving approach for migrating CSS code to preprocessors, we will list some of the papers related to the automatic migration of source code.

3.1 The Analysis and Maintenance of CSS Code

3.1.1 General Studies

There are few works in the literature that investigate style sheet languages from different points of view, like their history, features, and shortcomings. While they are not directly related to the topic of this thesis, studying them can help in understanding the challenges that understanding, analyzing, and refactoring CSS code impose.

Of the first publications that looked into style sheet languages (including CSS) was a short article authored by Marden and Munson in the Computer Journal in 1999 [JM99]. The authors discuss the "accessibility" as the great vision of web, which can only be achieved by allowing content and presentation vary independently, so that the end users "have the final say on the presentation". This separation of concerns is allowed in the presence of mature style sheet languages. The authors discuss some of the flaws in the two main standard style sheet languages at the time, XSL[1] (the eXtensible Stylesheet Language) and CSS, and show how this great vision is "blinded" due to these flaws. These flaws, however, were not studied in academia, leading the authors to call style sheet languages as "unexplored terrain".

As mentioned, the language that we know today as CSS is based on the proposal by Håkon Wium Lie. Lie summarize the history of CSS, how it works, and a lot of the subtleties associated with its development in his PhD thesis [Lie05]. In addition, he lists other style sheet proposals before CSS, deeply analyzes their strengthens and weaknesses, and explains why they did not gain the momentum as CSS did. Today's CSS is, however, much more complex than what Lie describes in his PhD thesis.

Quint and Vatton [QV07] outline the state of the art (of course, in 2007) of techniques and tools for editing style sheets. They pinpoint challenges CSS developers face and conclude that there is a crucial need for robust CSS debuggers and rule analyzers. They also propose ways for aiding CSS developers in editing style sheets. The proposed methods are implemented in Amaya, a web authoring tool developed by the same authors [QV04]. The authors claim that using Amaya can solve many of the mentioned problems in editing style sheets. Today, all the major web browsers include built-in debugging tools (e.g., FireFox's FireBug) that share more or less the same features as Amaya.

3.1.2 Code Quality

The quality of CSS code has been of interest in a number of studies in the literature, which will be discussed in this section.

Quality Metrics

Keller and Nussbaumer [KN09, KN10] come up with an "abstractness factor" for CSS: a more abstract CSS code is the one that can be applied on target documents with different content, and changes to the content will not cause faulty presentation. For example, the authors argue that

[1] XSL is a family of standards by the World Wide Web consortium (i.e., W3C), that is recommended for transforming and presenting XML documents. The subset of XSL standard that is used for defining formatting (the XSL Formatting Objects or XSL-FO) is, however, rarely used in practice, and the development of its standard has stopped. Note that, XSL-FO was mainly used for formatting XML documents for printed media, which can now be done using CSS's paged-media features.

type selectors (e.g., `p` for paragraphs) are more abstract (and less specific) than class selectors (e.g., `.c1`). Consequently, a higher proportion of style rules with type selectors in a style sheet means a more abstract style sheet. Through conducting an empirical study, the authors compute and compare the abstractness factor of human-written versus machine-generated CSS code, and conclude that humans *beat* machines in authoring CSS code in making more abstract style sheets. The authors, however, use the CSS code generated by WYSIWYG web editors for the comparison. These tools generate CSS code when the developer drag-and-drops user interface elements into HTML documents. Notwithstanding, generating CSS code by using CSS preprocessors is a trend in the industry [MT16a], and there is no study about the quality of the generated code from CSS preprocessors. Note that, refactoring duplicated declarations in CSS does not have any effect on the abstractness factor of CSS, as it does not have any effect on CSS selectors.

Adewumi et al. [AMIO12] proposed six complexity metrics for CSS, inspired from traditional code complexity metrics. The metrics include:

- Sum of the lengths of the style rules in a CSS file. Length of a style rule is measured by counting the number of style declarations declared in it. This metric is a proxy for size in CSS code, and generally, size has a high positive correlation with complexity.
- The number of style rules in a CSS file, as another size metric,
- The entropy metric. For traditional code, previous works used entropy to measure variety in size, structure, connections between the elements, or other code attributes. Similarly, the authors used entropy to measure the variety of style rule *types* in a CSS file, as a proxy for complexity. Style rules with type selectors and style rules with id selectors are two different style rule *types* to name.
- The number of extended style blocks. Extended rule blocks have selectors that add additional conditions to the selectors of existing style rules, e.g., style rule with selector `a:hover` is an extended style rule in Adewumi et al.'s definition,
- The average number of style declarations defined per style rule, which is another size metric,
- The number of cohesive style rules , i.e., the ones that possess only one style declaration.

There are several limitations with this work that make using the proposed metrics impractical for any study that deals with CSS code quality. First, what these metrics actually measure should be intuitively understood by the reader, as they are not given by the authors. For example, we don't know the reason why the authors used selector *types* when measuring variety, or why *extended* style rules are more complex. Second, some of the defined metrics are not necessarily useful. For example,

the proposed size metrics potentially highly correlate, so one might use a general size metric (e.g., the number of style rules in the CSS file) instead of using the proposed metrics. Third, the definition of some of the metrics is not necessarily complete, for example, the *cohesiveness* metric: a style rule that only contains style declarations for defining values for text properties (e.g., `text-align`, `font`, `letter-spacing`) can still be considered as a very cohesive style rule, while the style rule can have more than one style declaration. Finally, and the most importantly, while the authors provide a very shallow discussion for validating these metrics, there is no empirical validation conducted to support the claim that the proposed metrics actually measure CSS code quality. Therefore, using these metrics is by no means reliable, and a similar study (yet in a much more complete manner), is certainly required.

Bad Practices and Code Smells

There are tools that can detect bad practices in CSS, such as CSS Lint and W3C Validator. At the time of writing this thesis, CSS Lint incorporated 32 rules for identifying problems in CSS code, organized in six categories. These categories include *possible errors* (practices that are known to be error-prone, e.g., setting size attributes for an HTML element, which can be tricky due to the special *box-model* of CSS), *compatibility* (setting style properties that may behave differently across different web browsers), *performance* (practices that are known to be bad performance-wise), *maintainability* (using CSS features, like `!important`, which make understanding CSS code difficult), *accessibility* (obvious style definitions that will lead to reduced accessibility for users, e.g., `outline: none` that removes the border of text boxes in target documents, making them difficult to be recognized), and *OOCSS* (not following OOCSS guidelines[2]). W3C Validator service is provided by the World Wide Web Consortium (i.e., W3C, of which the CSS Working Group is responsible for standardizing CSS), and tests CSS code against CSS specifications, identifies a number of potential usability problems (e.g., when an element does not have a background color, but the text on it has color, it warns the developers that the rendered text might be illegible), and also checks for syntactical problems (e.g., missing curly brackets).

In her Master's thesis, Gharachorlu [Gha14] proposed eight code smells in CSS, complementing what CSS Lint and W3C Validator can identify. These code smells are classified into three categories: *rule-based* (including non-external and overly long style rules), *selector-based* (including selectors having too much cascading, with high specificity, containing erroneous adjoining pattern where developers incorrectly drop a space character between class and id selectors in a combinator, and overly general selectors), and *property-based* (undoing styles, where style values are reset to zero or `none`, and hard-coded values). The detection rules for detecting the proposed code smells are

[2]OOCSS is an abbreviation for Object-Oriented CSS. As its name suggests, OOCSS is inspired from Object-Oriented Programming and promotes CSS code that is more reusable and maintainable [Sul13]

implemented in a tool called CSSNose, and the author investigated the prevalence of the proposed code smells, in addition to the smells detected by CSS Lint and W3C Validator (total of 26 smell types) in 500 websites. She observed that 499 out of the 500 websites contained at least one instance of CSS code smell, and CSS properties with hard-coded values and undoing styles are the most prevalent code smells, found in 96% of the websites. Moreover, the author fit a regression model to predict the existence of code smells in CSS code.

While the mentioned tools provide a starting point toward having a complete repertoire of CSS bad practices, it turns out that the definitions of some of the code smells that they detect should be revisited. As an example, Gharachorlu's definition of *undoing style* code smell is not necessarily correct, as stated by Punt et al. [PVZ16]. Gharachorlu counts every style property set to zero or `none` as an instance of the *undoing style*. However, Punt et al. argues that this code smell essentially happens when a developer sets a value to a style property that already has a value (e.g., through inheritance, initial values, or cascading), and then resets it back to the original value. This definition (called the A?B*A pattern by Punt et al.) is broader than Gharachorlu's, and also makes more sense, as resetting styles in this way can have negative impact on understanding where a style value actually comes from. The instances of this code smell with this definition cannot be identified by CSSNose (i.e., false negatives). On the other hand, an example of a false positive for the *undoing style* code smell can happen in CSSNose when a developer wants to hide an element, so she correctly and logically sets its `display` property to `none`. This is detected as a code smell in CSSNose, while it is not. Punt et al. proposes a tool for detecting the instances of the A?B*A code smell and refactoring it within the web browser.

In any case, the prevalence of various smells in CSS code bases can be alarming. However, we are yet to see any work that investigates the severity of these code smells and quantifies their adverse impact. As a result, such a study is needed before making any attempt to eliminate code smells in CSS trough refactoring. Among the code smells, however, refactoring duplicated code – which is the topic of this thesis – can have immediate return of investment (e.g., smaller CSS file size that has to be transmitted over the Internet). Nevertheless, the infrastructure developed in this thesis can accommodate any refactoring in CSS.

Code Conventions

Having code conventions facilitate the *information exchange* between the developers [PJ15], leading to better code understandability and readability [Spi11]. Goncharenko and Zaytsev [GZ15, Gon15, GZ16] look into the existence of code conventions for CSS. The authors explain that CSS code conventions exist, although not from the World Wide Web Consortium, but from the CSS developer community. They used a special search engine that aggregates the results of multiple other search

engines to collect data about CSS code conventions, and come up with a catalog of 143 code conventions for CSS. The authors further developed a tool, namely CssCoco, that uses ontologies for detecting violations from code conventions in CSS code.

Detection and Refactoring of Dead Code

Dead (i.e., unreachable) code has been a hot topic of research in academia. For CSS code, dead code means style rules and declarations that are ineffective. This happens, for example, when a style rule has a selector that selects nothing in the target documents, or a style declaration that is always overridden by other style declarations through cascading. Detecting dead code is a challenging task in CSS, as parts of one CSS file can become used or unused, depending on the target document on which the CSS file is applied. For instance, consider a piece of JavaScript code that adds the following element to a login page of a web application, after the user enters a wrong password:

```
<span class="error">The password is not correct!</span>
```

This is done at runtime on a specific event. Now, consider that the attached CSS file to the login page defines a style rule like this:

```
.error {
  color: red;
 }
```

This piece of CSS code is ineffective (i.e., dead) when there is no authentication error, since there is no element for which the `class` attribute is equal to `error`. This practice (i.e., adding elements at runtime to the target documents) is indeed extensively done in today's web applications. An empirical study [BM13] showed that 95% of the websites (out of a corpus of 500 websites) contain client-side content that is initially hidden and JavaScript is used to inject the content at runtime. The study found that, in these websites, 62% of the states of target documents are initially hidden.

Using `@media` rules also makes detecting dead code challenging. Consider a piece of CSS code inside a `@media` rule defined for a specific device (e.g., a printer, or a smart watch). The enclosed code will be ineffective unless the target document is rendered on the specific device for which the media is defined.

Mesbah and Mirshokrae [MM12] developed Cilla, a tool that detects unmatched and ineffective selectors, overridden style properties, and undefined classes. The tool employs an automated technique which analyzes the runtime relationship between the CSS rules and the elements in target documents of dynamic web applications. This is done by crawling the web application using Crawljax, a tool that mimics the behavior of users by clicking on different *clickable* elements (e.g., hyperlinks, buttons, and elements for which the an event handler is attached for the `click` event). Crawljax explores new states of the web application caused by the events, and then the proposed

approach identifies ineffective CSS code with respect to the explored states. The authors report that, on average, 60% of style rules in today's CSS files are redundant. For the work done in this thesis, we used CRAWLJAX to identify dependencies between style declarations, which are necessary to consider when refactoring CSS code.

Genevès et al. [GLQ12] showed that *tree logics* can be used to apply static analysis on CSS: target documents are encoded as binary trees (as the approach works with binary trees), and CSS selectors and properties are translated to logical formulas. This representation makes several static analyses possible using tree logics, for example, the *emptiness (i.e., ineffectiveness) of selectors* – which basically means dead code in CSS – is checked when a selector's logical formula is not *satisfiable* for a given target document. The authors also mention similar use cases of the approach, like checking for the *equivalence of selectors* (i.e., two selectors select the same element), the *coverage without properties nor inheritance* (i.e., whether there are elements in the target document that are not covered by any CSS selector), and the *coverage with inheritance for a given property* (i.e., whether some style value is set to a given style property for all the elements of a target document, considering the propagation of values defined by the inheritance mechanism of CSS).

The refactorings introduced in this thesis require detecting the dependencies between CSS declarations to be done safely. Some of these dependencies can be detected using Genevès et al.'s proposed technique. However, this approach does not consider the presence of JAVASCRIPT or server-side programming languages for the analysis (e.g., the former given example of the addition of an HTML element at runtime using JAVASCRIPT). Not only this can lead to false positives in their approach, but it misses dependencies that exist in specific states of the target documents. In addition, employing Genevès et al.'s approach that requires encoding target documents to binary trees, modeling the problem as a logical formula, and using a special solver for detecting the dependencies would be an overkill; as we will see, the dependencies can be detected by using straightforward rules defined on the style rules' selectors and declarations, after attaching the CSS file under analysis to the target documents and mapping its style rules to the target documents' elements.

Hague et al. [HLO15] developed a tool, called TREEPED, with the goal of detecting redundant (i.e., dead) style rules in CSS. In contrast to CILLA that uses dynamic analysis, TREEPED attempts to detect redundant style rules using static analysis. The authors proposed a *tree-rewriting model* of the updates done on the target documents' tree structure at runtime by JAVASCRIPT (e.g., injecting new elements, adding CSS classes to the existing elements in the target documents, or removing target documents). The proposed approach outperforms CILLA in correctly detecting some of the cases that might be invisible to dynamic analysis, e.g., due to the sensitivity of dynamic analysis to the configuration of the crawler. However, as the work stands as a proof of concept, the model captures only the features of JQUERY, a popular JAVASCRIPT library, that is used for modifying

target documents at runtime, and does not consider the modifications done using pure JavaScript or other JavaScript libraries. That's why a tweaked version of this approach was not used in this thesis for detecting dependencies between CSS declarations.

Bosch et al. [BGL14a, BGL14b, BGL15] introduced an approach for reducing the size of CSS files by removing redundant style declarations and rules based on static analysis. In their approach, redundant CSS rules are the ones that can be detected within the CSS file, without the need for mapping the CSS file to any target document, i.e., where *reasoning* from the CSS file alone is possible. For example, the authors define *verbose declarations* as two declarations which are equal, but they are defined in style rules with equivalent selectors (e.g., `li.foo` and `li[class='foo']` are equivalent). Verbose declarations are redundant and can be removed from the style sheet. The authors propose a technique for eliminating such redundant style rules, and ague why the existence of `@media` rules can affect the refactorings. As mentioned before, we also take care of the `@media` rules in our refactorings.

Note that, in addition to detecting redundant style rules (which is also done by Cilla and TreePed), the goal of Bosch et al. is to *eliminate* the redundant CSS style rules, and the elimination is done aiming at reducing CSS file size. The proposed refactoring in Chapter 4 of this thesis also seeks size reduction by removing duplicated style declarations in CSS files. As a result, the approach of Bosch et al. is complementary to ours, as it will be discussed later.

3.1.3 Alternative Language Proposals for CSS

Both academia and the industry have come up with tools and approaches for augmenting CSS to compensate for its shortcomings. From the industry, CSS *preprocessors* have almost become the de facto way of developing style sheets. CSS preprocessors are discussed in more detail in Chapter 5. The proposals from the academia, however, have not gained much momentum, but we briefly introduce the reader with some of them in this section.

Badros et al. [BBMS99] introduced the notion of the Constraint Cascading Style Sheets (i.e., CCSS) that extend CSS by allowing the developers to define arbitrary linear arithmetic constraints (e.g., to control elements' positions and sizes), and finite-domain constraints (e.g., to control font properties). An example of a constraint in CCSS is `@constraint #c1[width] = #c2[width]`, which forces the two elements to have the same width. The proposal has been implemented in the Amaya web browser, that we formerly talked about.

Wieser [Wie06] proposed CSS^{NG}, an enhanced version of CSS for supporting some dynamic features in CSS. For example, in CSS^{NG}, the developer is allowed to write:

```
*::before {
  content: element("span", "elem");
}
```

This style rule selects all the elements in the target document (i.e., using the universal selector `*`), and an element in the following form is injected *before* each selected element:

```
<span>elem</span>
```

Note that, the element is added before each element as the selector uses the `::before` pseudo element. Normally, this is done using JavaScript, because the `content` property in CSS does not allow adding HTML *tags* before (or after) the selected element (although normal text is allowed). CSS^{NG} allows this by using the `element()` function.

Serrano [Ser10] proposed HSS, a preprocessor language for CSS that supports custom properties (acting like variables), user-defined functions, conditional expressions, user-defined types (i.e., variable selectors) and the support for arithmetic calculations. Like any other CSS preprocessor, the HSS code should be first compiled to pure CSS. Note that, most of the features supported in HSS are also supported in the industrial CSS preprocessor languages (e.g., Less and Sass). However, HSS does not support *selector nesting*, a handy and extensively-used feature supported by the industrial CSS preprocessor languages (as we will see in Chapter 5, *nesting* is one of the most popular language features used by the developers).

HSS has not been adopted by the industry. This could be possibly explained by the fact that the syntax of the HSS-specific features is radically different from pure CSS, in contrast to the industrial CSS preprocessors that have a very similar syntax to pure CSS. That's also why we did not use HSS in our empirical study (Chapter 5), because we couldn't find any website that used HSS.

3.1.4 Clone Detection in CSS

Mao et al. [MCD07] proposed an approach for the automatic migration from *table-based structure* to the *style-based structure* for web pages, and in a step of this approach, duplicated style rules are identified using a traditional clone detector. The authors first use *table recognition* techniques to detect portions of web pages that use tables (i.e., the `<table></table>` elements) for layout. This was a frequent anti-pattern in web application development, because CSS did not provide any layout mechanism[3]. This anti-pattern is an abuse of tables that brings several performance and maintenance problems. Detecting tables used for defining layouts is first done on single web pages, and then the HTML tables are transformed to a nested, hierarchical structure based on `<div></div>` elements. In the next step, for each web page, the corresponding CSS code is generated for styling the `<div></div>` elements. Then, the clone detection approach proposed by Cordy et al. [CD04]

[3] As mentioned in Section 3.1.6, CSS specifications added the layout module to fill this gap.

is used in order to find duplicated code across the generated CSS files to consolidate them into a unified CSS file, which could be applied to different web pages.

In Mao et al.'s technique, a set of style rules that are exactly the same (with the exception of style rule's selector) are detected as clones. Only one instance of the cloned style rules are kept, and the rest are removed. The locations in the HTML files under analysis where the removed style rules are used are in turn updated to use the single style rule that is kept. This kind of detection and analysis that is used by Mao et al. is sufficient for finding duplicated code with no differences in style declarations (except for white spaces and comments). However, the approach proposed in this thesis is able to detect more advanced types of duplicated code in CSS files at a finer granularity, as will be discussed in the next chapter.

In a technical report, Federman and Cook [Dav10] show the applicability of the Formal Concept Analysis (i.e., FCA) for grouping style declarations that are exactly the same into *grouping selectors* in CSS. FCA allows analyzing data which describes relationships between a set of objects, and a set of attributes that those objects might possess. Using FCA, one can investigate the data with queries like "what is the set of objects that all share a set of particular attributes?". In the Federman and Cook's approach, style rules act as *objects*, and style declarations as *attributes*. With this representation, the aforementioned query will essentially result into the set of style declarations that are repeated in some style rules, and can be in turn refactored to eliminate duplication.

As we will discuss in Chapter 4, this is very close to what we have proposed in this thesis for refactoring duplication code within CSS. There are, however, several shortcomings in the Federman and Cook's approach, that our work has attempted to solve. We list some of these shortcomings in detail in Chapter 4.

There are also tools in the web development community for detecting or removing duplicated code in CSS. CSSCSS [Moa13], for example, is a clone detection tool designed specifically for CSS. It detects declaration-level refactoring opportunities in a manner similar to our technique; however, it supports only exactly copied-and-pasted CSS code, and does not detect the majority of the advanced types of duplication that the proposed approach in this thesis can detect. It also does not provide any way for refactoring the detected duplication instances. CSSPurge [Qua13] detects duplicated style rules (i.e., the ones with exactly same selectors) and merges all of them into a single style rule, removing style declarations that it assumes will not be applied, with the assumption that they will be always overridden. In any case, CSSPurge does not guarantee a safe code transformation that will have the same styling effect on target documents.

3.1.5 Migrating CSS to Preprocessor Languages

One contribution of this thesis is proposing an automatic technique for migrating CSS code to preprocessor languages (which is discussed in Chapter 6). We noticed that Charpentier et al. have worked in parallel with us on the same problem [CFR16]. While our work was published in the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE'16), their work was published in the 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME'16). We have discussed, in detail, the differences between our work and Charpentier et al.'s in Chapter 6. In a nutshell, we used an association rule mining technique for detecting duplicated code in CSS (built on top of the work done in Chapter 4), discussed how safety preconditions can lead to presentation-preserving transformations, and further provided a testing technique for assuring the safety of the transformations. Charpentier et al., on the other hand, used Formal Concept Analysis for grouping duplicated declarations in CSS. As mentioned, using FCA with the goal of grouping duplicated declarations was not a novel idea, as it appears in a technical report by Federman and Cook in 2010 [Dav10]. Moreover, and more importantly, the technique proposed by Charpentier et al. neither provides nor checks any preconditions, i.e., the proposed transformations are not necessarily safe. However, Charpentier el al. conducted a user study with four developers to assess the acceptance of the transformations, something that we lack in this thesis, but is planned for the future work. There are more differences between the two approaches that we will discuss in Chapter 6.

3.1.6 Other Related Studies

Alternative implementations

Acebal et al. [ABRC12] provided an implementation of the CSS *Layout Module* using JavaScript. At the time before CSS 3 modules were emerged, CSS did not provide any means for specifying the *layout* of the web pages, while it already supported a rich set of properties for defining other presentation facets (e.g., fonts and colors). Web developers often *misused* existing CSS style properties (e.g., the `float` property) with additional HTML markup to achieve the desired layout. To fill this gap, the CSS Layout Module was proposed by Acebal in his PhD thesis [Ace10]. Acebal provided the JavaScript implementation as a proof of a concept to show the layout module's advantages.

Performance

Jovanovski and Zaytsev [Jov16, JZ16] discuss *critical* CSS *rules*: style rules in the *external* CSS files that have to be loaded so that the web browser can start rendering the web page. By refactoring critical CSS rules and moving them from *external* CSS files to become *embedded*, the authors

observed an average of 1.3 seconds speed-up in loading web pages in a corpus of 1000 web sites.

Jones et al. [JLM+09], and in a follow-up work, Meyerovich and Bodik [MB10] propose efficient and parallel algorithms for several time-consuming tasks that the web browsers have to fulfill when rendering web pages with CSS. These tasks include CSS selector matching, layout solving, and font rendering. Using the proposed algorithms, different browsers' layout engines gained performance ranging from 3x to 80x.

Testing

When maintaining existing code (e.g., adding new features, fixing bugs, improving code quality by applying refactorings), testing is a crucial activity to make sure that code modifications do not break any existing functionality. In the Introduction of this thesis (Chapter 1), we mentioned a story showing how some critical pages at Dropbox were unknowingly broken due to modifying a shared CSS file [Ede14] – an example of why testing is also very important for CSS code.

Surprisingly, *there is no study in the literature dedicated to testing* CSS *code* [GMBCM13]. In the industry, two rather immature approaches have been proposed to test CSS code:

The Frozen DOM In this approach, CSS regression testing essentially includes checking the equality of the final style values that are applied to each style property, for all the elements of the target documents, before and after modifying the CSS code. This approach needs maintaining a list of *static* (i.e., *frozen*) target documents (or multiple states of a single target document) to apply the CSS code on, and updating it whenever the structure of the web pages changes. The comparison of style values is usually done by employing JavaScript in a web browser. In JavaScript, developers have access to the elements of the target documents represented as a tree structure, using a standard Application Programming Interface (i.e., API) that is called the Document Object Model (DOM). The term DOM is sometimes used interchangeably with the tree representing the target document, like in *the Frozen DOM* technique. The elements of target documents are in turn called *DOM elements*. Similarly, different states of a single target document generated by manipulating it using JavaScript at runtime are called *DOM states*. There are several tools that facilitate applying the frozen DOM technique, e.g., Hardy [Mad13], and css-wrangler [Nei17].

There are some shortcomings with the Frozen DOM approach. First, it is required to keep the frozen DOM states in sync with the actual web application, which is a tedious task. To overcome this problem, developers sometimes configure a web crawler to run the web application and extract the underlying DOM states automatically. In this case, the explored DOM states are more realistic, as they have the real content. Note that, a web page can appear with no problem with dummy content, while it can break with large content, or in the presence

of multimedia elements in the text (e.g., images, or embedded video players). In a large web application, however, configuring the crawler to cover all DOM states with diverse-enough content can also be cumbersome.

The Frozen DOM technique might be also unable to spot the possible behavior alternations for web pages presented on different media. A state of a web page can be rendered correctly on one device and incorrectly on the other, while the style values being assigned to all the elements are the same before and after modifying the CSS code. As a result, the Frozen DOM technique should be always complemented, either by manual investigation or another testing method.

Using image processing techniques In this technique, instead of using DOM states for comparing styles after modifying CSS code, image snapshots are taken from different states of the web application, and image comparison techniques are used for detecting differences between the snapshots. PhantomCSS [CtHdt13] is one of the tools that employs this technique for automating *visual regression testing*. Liang et al. [LKL+13] used this technique for developing a tool, namely SeeSS, which is able to track the visual impact of code changes in CSS across a website.

The *Achilles' heel* of the approach is the underlying algorithm for image comparison; PhantomCSS, for instance, uses a simple RGB pixel differentiator. This can lead to detecting very low-level differences that developers (or users) might tolerate, or even not notice. In addition, having dynamic content (which is the case, usually) can lead to false positives, i.e., tests failing due to the changes in the content, but not the style or layout. As a result, it is important to feed the technique with only *meaningful* places of the screen shots to avoid false positives. To solve these problems, Mahajan and Halfond proposed WebSee [MH15] that uses *Perceptual Image Differencing*, a computer vision technique that compares two images using computational models that make comparing two images similar to what humans' visual system does, to detect visual differences. It is also possible to define exclusion regions for testing in WebSee. Mahjan et al. further improved WebSee by employing a probabilistic model based on the Bayes' theorem to connect the detected problems to their root causes in the HTML code [MLBH16]. A root cause, in their definition, means the HTML element that is found to be faulty, and one of its attributes or style properties that has different value compared to the test oracle.

Cross-browser compatibility testing: Sometimes a web application exhibits presentational and functional inconsistencies when it is viewed on different web browsers. A study shows that more

than 20% of web applications suffer from cross-browser compatibility issues [RCPO13]. An important quality aspect of web applications to test, therefore, is *cross-browser compatibility*. These issues sometimes have roots in CSS. Knowing some of the approaches in detecting cross-browser compatibility issues might be helpful in devising methods for CSS testing to accompany refactoring and migration techniques.

WebDiff [RCVO10] locates cross-browser issues, including differences in the presentation and the structures (i.e., DOM trees) of web pages across different web browsers. WebDiff compares screen shots taken from the unvarying parts of web pages (i.e., parts that do not change in successive reloads, e.g., parts that are not ads or videos) for detecting presentational changes. Also, it utilizes a non-exact comparison algorithm for comparing DOM trees landed to different web browsers to detect structural differences.

WebDiff finds issues on single web pages. In contrast, CrossT [MP11], tracks the behavior of web applications in different web browsers to detect functional inconsistencies. This is done by comparing the *state-flow* graphs, which Crawljax generates when crawling the web applications, across different web browsers. CrossT does not aim for detecting presentational inconsistencies that arise on the same content on different web browsers. CrossCheck [CPO12] is proposed to take advantage of the techniques proposed in both WebDiff and CrossT, and augments them by employing a machine learning-based classifier to decide whether two screen elements are different. X-Pert [RCPO13] further enhances CrossCheck by improving its differencing technique for detecting layout issues. Similarly, Browserbite [SDKS14, SDS13] uses image comparison techniques for detecting cross-browser compatibility issues, but utilizes machine learning methods (including both a classifier and a neural network) to remove false positives from the results.

Defect Prediction

There are a plethora of works in the literature, studying the approaches for predicting the existence of defects, with the hope of discovering them before shipping the code to the end users. To our knowledge, for CSS, there is only one recent study that investigates the possibility of defect prediction. Biçer and Diri [SBD16] trained Naive Bayes, Logistic Regression, and Random Forests classifiers to predict whether a style rule is going to be buggy or not. The authors used several metrics extracted from each style rule as predictors, including but not limited to the number of *simple selectors* in a combinator selector of the style rule, the specificity of the style rules' selector and the number its pseudo-classes, and the number of declarations defined in the style rule. The training was done using a dataset of four open-source projects, and the defective rules (i.e., the *1* class in the classification) were found by marking every style rule that were changed in a bug-fixing commit in the history of the projects. The results showed a prediction performance comparable to

the state-of-art prediction techniques. Moreover, using the classifiers reduced the cost required for inspecting the defect prone rules by 8% to 29%. However, the study does not take into account the dependencies between the style declarations (caused by the cascading feature of CSS), which can intuitively be the reason for several bugs (indeed, that's why all web browsers provide a feature in their debugging tools for displaying overridden style values for any selected element).

3.2 Clone Detection in Web Applications

Code duplication has been extensively studied in procedural and object-oriented languages [RBS13, RC07], leading to a variety of detection techniques. Several researchers have also developed techniques for the detection of duplication in web artifacts. Most of the studies in the area of web applications have focused on the detection of duplicated content in web pages [BK01], or finding web pages with similar structure [DLDP01, DLDPF02]. Boldyreff et al. [BK01] replace the content of web pages (i.e., the text inside different tags) with hash values and compare them to find duplicated content in web pages. Lanubile and Mallardo [LM03] propose a semi-automatic approach to find *function clones* in the source code of web applications. Their approach first compares the names of the functions written in either JavaScript or VBScript. If the names are the same, they compute various size metrics and report the functions with similar metric values as candidate clones. In their follow-up work, they evaluated this approach on four web applications and found out that 21% to 80% of functions were duplicated and could be refactored [CLM04].

De Lucia et al. [DFST05] use the Levenshtein edit distance to quantify the structural similarity between web pages. Rajapakse and Jarzabek [RJ05] use CCFinder (a tool which detects clones by applying token-to-token comparison [KKI02]), to find code clones in the source code of web applications written in various languages. They examined 17 web applications and found a duplication rate of 17% to 63%. Synytskyy et al. [SCD03] use an *island grammar* in order to define smaller portions of the HTML syntax for elements, such as forms and tables, that might be cloned across different pages. The grammar is used to extract those structures from web pages and examine whether their structure is repeated in other pages.

Cordy et al. [CD04] propose an approach that is language-independent and can detect exact and near-miss clones using island grammar extraction, pretty-printing and textual differencing of the clone candidates. In their study, they used this approach for detecting clones in HTML code. This work led to the introduction of NiCad [RC08], which is an exact and near-miss (i.e, Type 2 and 3) clone detector. Muhammad et al. [MZYR13] also use NiCad to find clone patterns in the PHP code of two industrial systems.

The extracted duplication information can be used to re-engineer web applications, i.e., to create

dynamic pages from static ones [BK01, SCD03], generate more-generalized dynamic web pages to minimize the duplication [DFST04, RJ07], or find similar functionalities across different web pages [DFST05]. None of the aforementioned works investigated the existence of duplication in CSS code, or developed a technique specialized in the safe elimination of duplication in CSS code.

3.3 Empirical Studies on Language Features Usage

In Chapter 5 of this thesis, we look at how CSS developers take advantage of CSS preprocessor language features. To the best of our knowledge, this is the first empirical study on the use of CSS preprocessors. The gained knowledge, as we will see, is helpful for several audience. For tool builder, for example, it helps in achieving the ultimate goal of designing recommendation systems that migrates pure CSS code to preprocessors, as a means to improve the maintainability of existing CSS code.

In the literature, there are several empirical studies on the use of language features in different languages and technologies, with similar goals to our work, e.g., understanding how developers have adopted these language features. For instance, Ernst et al. [EBN02] investigated how C preprocessors are used in practice, by conducting an empirical study on 26 publicly available C programs, using a tool which includes approximate, Cpp-aware parsers for expressions, statements, and declarations. Tempero et al. [TNM08] studied the use of inheritance in Java programs. They used different metrics, such as Depth of Inheritance, extracted from the bytecode of the subject systems for their analysis.

Grechanik et al. [GMD+10] conducted a large-scale study on the use of object-oriented features including classes, methods, fields and conditional statements on 2000 open-source Java projects. They represented the information about the source code in a relational database and used SQL to extract the required metrics about different features. Gil and Lenz [GL10] conducted an empirical study on how Java developers take advantage of method overloading in 99 open source Java programs. Similar to Tempro et al.' work [TNM08], they also used bytecode for data collection.

Xiaoyan et al. [ZWSS14] investigated the frequency of different statement types (e.g., `if`, `return`, function declarations) in 311 projects written in C, C++ and Java. They extracted this information from an XML representation (i.e., srcML) of the source code of subject systems. Dyer et al. [DRNN14] conducted a very large-scale study on 31K open-source Java projects to find usages of new Java language features over time. This is done on the Abstract Syntax Tree (i.e., AST) of the source code of the subject systems. Richards et al. [RLBV10] studied the use of dynamic language features in JavaScript applications, using an instrumented web browser. Callaú et al. [CRTR11] conducted an empirical study on the use of the reflection feature in 1000 Smalltalk projects by statically tracing the features being used from the AST of the source code.

Martin et al. [MCAA15] examined the use of GNU Make's language features (such as functions, macros, lazy variable assignments and the Guile embedded scripting language) in around 12k make files of 250 open source projects. They used TXL to define a custom grammar for Makefiles to extract and count instances of features. In our analysis, we used the AST of the parsed preprocessor files to extract the required information, similar to other works including [CRTR11, DRNN14].

In a recent work, we looked at how Java developers take advantage of *lambda expressions*, which have been retrofitted into Java 8 [MKTD17]. Similar to the study done in this thesis, we have provided implications for developers, tool builders, language designers, and researchers.

3.4 Automated Code Migration

In Chapter 6, we investigate the possibility of migrating existing CSS code bases to take advantage of CSS preprocessor language features. As mentioned, to our knowledge, there is only one study having the same goal of migrating CSS to preprocessor languages, which was published in parallel with our work [CFR16]. We will investigate the work in more detail in Chapter 6.

3.4.1 Migration of the Legacy Systems

There are numerous works in the literature proposing migration techniques for legacy systems in order to improve their maintainability. The migration activities can be done either within a language (e.g., for taking advantage of a new feature added to the language), or from one language to another one. Several incentives drive academia and practitioners to develop such techniques, including but not limited to improving the maintainability of code bases.

For example, several researchers developed techniques for migrating procedural code to the object-oriented paradigm, such as automatic or semi-automatic translators from C to C++ [ZK01], Eiffel [TFN+12], or Java [MM01]. Migration is also performed when there is a lack of human resources for maintaining existing software systems written in an extinct language, e.g., migrating Lisp to Java [Lei07]. Other works proposed approaches for detecting opportunities to use constructs introduced in a newer version of a programming language. For Java, there are techniques for introducing parameterized classes from non-generic ones [KETF07], the enumerated type [KSR07], and Lambda expressions [FGLD13].

3.4.2 Migration of Web Systems

As mentioned before, some of the studies that investigated the duplication in the content or structure of web pages, proposed techniques for migrating duplicated static web pages to dynamic, server-side web applications [BK01, SCD03]. The proposed work of Mao et al. [MCD07] for the automatic

migration of HTML pages having table-based structures to the style-based structure is one more example of migration activities in web systems.

3.5 Chapter Summary

In this chapter, we looked at several works related to the topic of this thesis. As mentioned, while there are numerous works in the literature that investigated clones in traditional code, the problem of duplicated code in CSS has not been deeply studied. For CSS, refactoring duplicated code (and in general, applying any kind of refactoring) has received no or very little attention from academia, while CSS is an extensively-used programming language. In the next chapter, we will look at the problem of duplication in CSS in more detail.

Chapter 4

Refactoring Duplication Within CSS

4.1 Introduction

CSS development is far from being a rigorous and disciplined process. One instance of undisciplined development is the definition of new CSS rules by copying and modifying existing code instead of reusing already defined ones, i.e., code duplication. There is empirical evidence that duplicated code in software systems developed with procedural or object-oriented languages is associated with increased maintenance effort [LW08], higher error-proneness [JDHW09], and higher instability [MRS12] in terms of change frequency and recency. We believe that the development and maintenance of CSS code is also subject to the same problems caused by code duplication. A small change in the presentation of a website might require tremendous effort for locating and understanding parts of the CSS code that need to be consistently updated.

The problem of duplication might even be more intense in CSS code, because the CSS language lacks many features available in other programming paradigms that could enable code reuse. For instance, there is no notion of *functions* in CSS to build reusable blocks of code.

In addition, CSS code has to be transferred over the network from a server to a large number (sometimes, millions) of clients. Extensive code duplication increases the size of the transferred data, resulting in a large network load overhead.

Once on the client side, CSS code has to be processed by the web browser. Extensive code duplication increases the size of the CSS code that has to be processed by the browser (e.g., CSS code has to be parsed and the selectors have to be matched to the DOM elements), resulting in a computational overhead. This could affect more mobile or wearable devices that have limited computation, memory, and power resources available. Previous studies [MB10] have shown that the visual layout of web pages, performed by analyzing the CSS code, consumes 40–70% of the average

processing time of the browser.

In this chapter, we propose an automated technique to (1) analyze and detect various types of CSS duplication, and (2) discover and recommend refactoring opportunities to eradicate duplicated CSS code. In summary:

1. We define various types of duplication in CSS code and propose a technique for the detection of duplication instances. Additionally, we provide empirical evidence on the extent of duplication in the CSS files of several web applications. To the best of our knowledge, this is the first study that investigates the problem of duplication in CSS code in such an extensive manner.

2. We present a technique for eliminating CSS code duplication through presentation-preserving refactorings. Additionally, we provide a ranking mechanism based on the size reduction that can be potentially achieved by each suggested refactoring to help CSS developers prioritize their maintenance efforts by focusing on the refactorings with higher impact.

3. We describe preconditions that should be met to *preserve* the CSS styling after the application of a refactoring.

4. We perform an empirical study to assess the efficacy of our approach using 38 real-world web sites/web application that use 91 CSS files in total.

Our results show that the extent of duplication in CSS code is indeed very intense ranging from 40% to 90% for the vast majority of the examined CSS files. On average, we found 165 refactoring opportunities in the examined CSS files, out of which 62 could be applied by preserving the styling of the web pages. Finally, the average size reduction achieved by applying only presentation-preserving refactorings was 8%, while the highest reduction was 35%.

Note: Earlier version of the work done in this chapter has been published in the 22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2014) [MTM14a].

4.2 Duplication in CSS

The code duplication problem is expected to be more potent in CSS due to the lack of reuse constructs. As a result, many common style declarations have to be repeated in multiple style rules. In this section, we define different types of duplicated declarations in CSS, and provide examples from real-world web applications. The proposed duplication types for CSS code are inspired from the software clones research area. Compared to procedural and object-oriented languages like C and Java, CSS has a very simple syntax, which makes the detection of duplication an easier task. However, the detection of more advanced types of duplication in CSS requires a CSS-specific technique.

We then show how we can eliminate the duplication through a simple refactoring that happens *within* CSS code (i.e., without using another programming language, like a CSS preprocessor).

4.2.1 Duplication Types

The software clone research community has defined different types of duplication in procedural and object-oriented code, based on the textual or functional similarity of two code fragments [RC07]:

Type 1 (Or exact clones) Identical code fragments, except for variations in whitespace, layout, and comments,

Type 2 Structurally/syntactically identical fragments except for variations in identifiers, literals, types, layout and comments,

Type 3 Copied fragments with statements changed, added or removed in addition to variations in identifiers, literals, types, layout and comments,

Type 4 Code fragments that perform the same computation, but implemented through different syntactic variants.

In this work, we focus on duplicated CSS declarations inside CSS style rules. By eliminating this kind of duplication, we can reduce the size of the CSS code that has to transferred over the network and be maintained in the future. We define three different types of declaration-level duplication in this section.

Type I: *Declarations having lexically identical values for given properties.*

An extreme example of type I duplication can be seen in Figure 4, which is taken from the main CSS file of Gmail's inbox page. In this file, there are 23 declarations that are repeated in three style rules. Figure 4 shows only a subset of these declarations for two of the style rules.

Note that the definition of type I duplication considers *only* the equality of the property values and disregards the value *order*. For instance, consider the two `border` declarations in Figure 4. If one of them was defined as `border: transparent solid 1px` (i.e., same values in different order), we would still consider them as an instance of type I duplication, because based on the CSS specification, the browser will interpret both declarations in the same way.

Type II: *Declarations having equivalent values for given properties.*

```
.z-b-V {
    -webkit-box-shadow:
       0 1px 0 rgba(0, 0, 0, .05);
    box-shadow:
       0 1px 0 rgba(0, 0, 0, .05);
    background-color: #fff;
    color: #404040;
    cursor: default;
    font-size: 11px;
    font-weight: bold;
    text-align: center;
    white-space: nowrap;
    border: 1px solid transparent;
    border-radius: 2px;
    ...
}
```

```
.z-b-G {
    -webkit-box-shadow:
       0 1px 0 rgba(0, 0, 0, .05);
    box-shadow:
       0 1px 0 rgba(0, 0, 0, .05);
    background-color: #fff;
    color: #404040;
    cursor: default;
    font-size: 11px;
    font-weight: bold;
    text-align: center;
    white-space: nowrap;
    border: 1px solid transparent;
    border-radius: 2px;
    ...
}
```

Figure 4: Type I duplication in Gmail's CSS

In CSS, we may have the same values for properties with alternative representations. Font size, color, length, angle and frequency values are representative cases. For instance, Table 1 shows alternative representations for the same color. We consider all these different representation values as *equivalent* values.

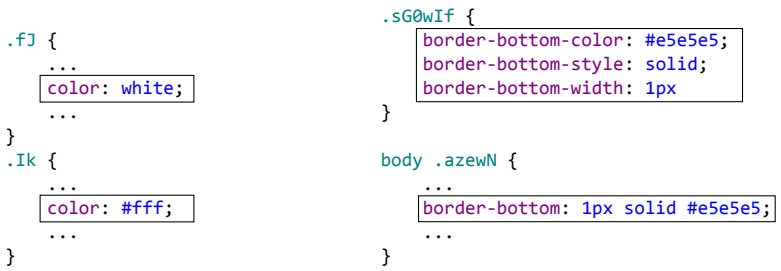
Table 1: Different representations for the `rebeccapurple` color.

Representation[†]	**Value**
Named Color	`rebeccapurple`
Hexadecimal	`#663399`
Red, Green, and Blue values	`rgb(102, 51, 153)`
Red, Green, Blue, and the Alpha Channel values	`rgba(102, 51, 153, 1)`
Hue, Saturation, and Lightness values	`hsl(270, 50%, 40%)`
Hue, Saturation, Lightness and the Alpha Channel values	`hsla(270, 50%, 40%, 1)`

[†] In the CSS Color Module Level 4 specifications, there are more ways added to represent colors, but our current implementation does not support them.

If two or more declarations have equivalent values for the same properties, we consider them as an instance of type II duplication. In Figure 5a, we can see an example of type II duplication in the CSS file of Gmail's inbox page. Note that the declarations with `color` property are duplicated.

In addition, there are some default values for certain properties, which are applied when explicit values are missing. For example, based on the CSS specifications, the declaration `padding: 2px 4px 2px 4px;` can be also written in a shorter version as `padding: 2px 4px;` with



(a) Type II (b) Type III

Figure 5: Declaration duplication in Gmail's CSS

the same effect. Such cases are also considered as equivalent declarations and thus constitute instances of type II duplication.

> **Type III**: *A set of individual-property declarations is equivalent with a shorthand-property declaration.*

Some CSS properties, such as `margin`, `padding`, and `background` are called *shorthand properties*. With these properties, we can define values for *a set of properties* in a *single style declaration*. For instance, the `margin` shorthand property could be used in order to define values for `margin-top`, `margin-right`, `margin-bottom` and `margin-left`, as shown in Figure 6.

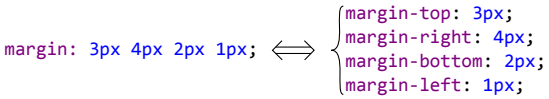


Figure 6: Shorthand and individual declarations

If a style rule contains a set of individual style declarations, which is equivalent to a shorthand declaration of another style rule, we consider those declarations as an instance of type III duplication. Figure 5b, shows an example of type III duplication in the CSS file of the Gmail's inbox page.

4.2.2 Eliminating Duplications

The aforementioned types of duplication can be eliminated directly in CSS code without changing the target documents by extracting a style declaration with *grouping* selectors. If a set D of declarations is duplicated (in the form of type I, II, III duplication) in a set of n style rules with selectors $S_1, S_2, ..., S_n$, we can create a new style rule with a grouping selector $S_1, S_2, ..., S_n$ and move D to this new style rule. Such refactoring has been depicted in Figure 7. Here, the CSS code snippet on the left side could be refactored to what is shown on the right side.

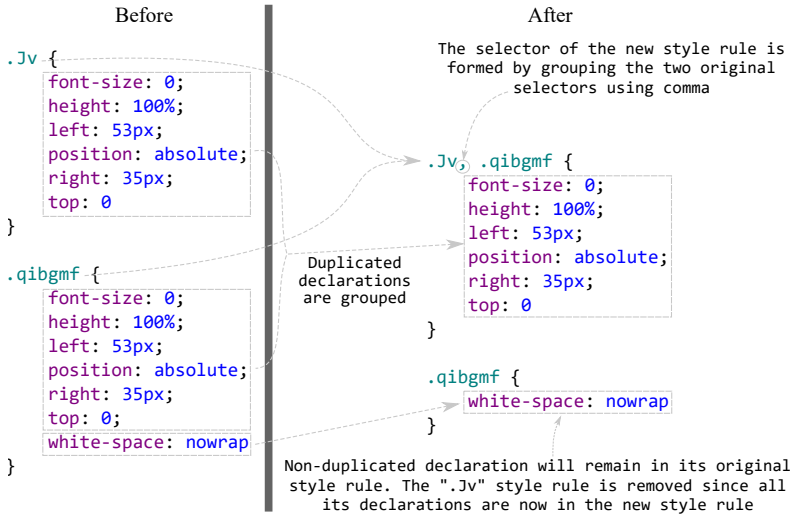


Figure 7: Grouping style declarations to remove duplication

Another possible solution, based again on grouping, is to create a common *class* for the repeated declarations and assign that class to the target elements. However, this solution requires also to update the target documents, so that they make use of the newly defined class.

4.3 Method

Our method for the detection of duplication in CSS and the extraction of refactoring opportunities is divided into four main steps discussed in the following subsections.

4.3.1 Abstract Model Generation

To find duplicated style declarations, we first parse all the CSS files of a given web application. Our method then generates an instance of the abstract model shown in Figure 8, which represents a high-level structure of the application's CSS code. All the analysis and refactoring activities, as we will see, are done on this hierarchical model, and therefore are independent from the abstract syntax trees created by any CSS parser. This is necessary, because CSS parsers can become obsolete due to fast evolution of CSS specifications. We will give more details about this design decision in Chapter 7.

Note that, the World Wide Web Consortium (i.e., W3C) has a standard object model for CSS called the CSS Object Model (i.e., CSSOM) [Con16a]. Using the API provided by CSSOM, one can access and modify CSS code, e.g., using JavaScript in the web browser. As mentioned, W3C has also standardized the Document Object Model (i.e. DOM) for accessing and modifying HTML documents. While DOM has a very rich set of APIs that allow advanced analysis of HTML documents,

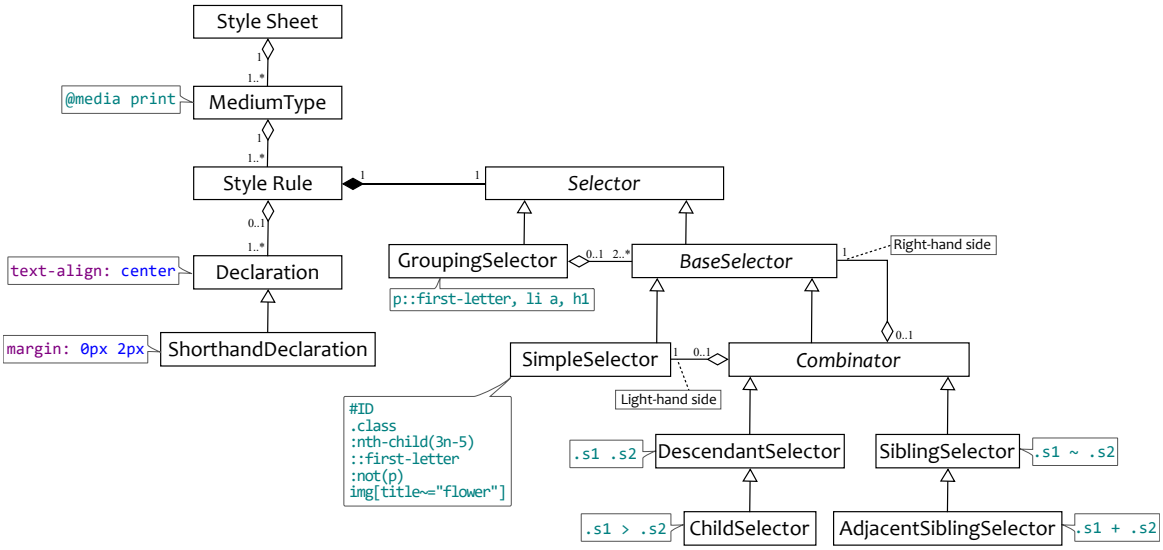


Figure 8: A hierarchical object model for CSS

CSSOM is rather immature. For instance, a selector is represented as a string[1] in CSSOM, and there is no API for accessing different parts of a complex selector (e.g., `.menuitem > li a:hover`). This is the same for style values, i.e., even a list of style values in a style declaration is represented as a string.

Any advanced analysis on CSS code requires having fine-grained access to each of the building blocks of the CSS code. For example, when analyzing type II duplications we need to compare style values for certain style properties regardless of the order they appear in the style declaration. Therefore, a single string representation for the list of style values will make the comparison difficult, if not impossible. Due to several similar shortcomings, we did not use CSSOM, and instead designed our own model for representing CSS code.

As shown in Figure 8, every CSS style sheet may be bound to some *medium type*. This identifies the target *presentation medium* for which the style sheet is defined. For instance, one may distinguish styles for printing and displaying in a mobile device [Wor17]. For defining presentation mediums, we use the `@media` at-rules. It is also possible to define the same style rules for different media types within a given style sheet.

In this model, a `BaseSelector` represents selectors that do not perform grouping. A `SimpleSelector` represents *type selectors* or the *universal selector* (`*` selector, which selects every element). The class `SimpleSelector` has special attributes for specifying properties like the element ID, class identifier, pseudo class, pseudo element, and attribute conditions. Finally, a `Combinator` represents *combinator selectors*, which can be formed by combining a `SimpleSelector` with a `BaseSelector`. Examples

[1] More accurately, the type is `DOMString`, a sequence of characters encoded using UTF-16. In Java and ECMAScript, `DOMString` is equal to the `String` type, since both languages use UTF-16 as their encoding.

of each of these selectors can be seen in Figure 8.

4.3.2 Preprocessing

The detection of Type I duplications does not require any preprocessing. However, to facilitate the detection of type II and III duplication instances, we perform three separate preprocessing steps.

Normalization of property values

In this step, we replace values that can be expressed in different formats or units (e.g., colors and dimensions) with a common *reference* format or unit. For example, every color in named, hexadecimal, or HSL format (see Table 1) is replaced with its equivalent `rgba()` value. Every dimension specified in centimeters, inches, or points is converted to its equivalent pixel value[2]. All applied conversions are based on the guidelines provided by the CSS specifications [Wor17]. Replacing values with a common representation is known as *normalization* and has been used in traditional code clone detection techniques for finding clones with differences in identifiers, literals, and types [KKI02, RC08]. Our motivation is to find declarations using alternative formats or units for the same property value. Such cases constitute type II duplication instances.

Addition of missing default values

As we discussed in Section 4.2, CSS developers sometimes omit values for some of the multi-valued properties, in order to have shorter declarations. In this step, we have some predefined rules based on the CSS specifications [Wor17] that add the *implied* missing values to the properties in the model. For instance, `margin` property should normally have 4 values. We enrich the declaration `margin: 2px 4px` with the two missing implied values as `margin: 2px 4px 2px 4px`. This allows the comparison of declarations based on a complete set of explicit values enabling the detection of type II duplication instances.

Virtual shorthand declarations

Detecting type III duplication instances requires the comparison between shorthand declarations in one style rule and an equivalent set of individual declarations in another style declaration. To facilitate this task, we add "virtual" shorthand declarations to the model. We examine the declarations of every style declaration to find sets of individual declarations that can be expressed as equivalent shorthand declarations. For every set of such individual declarations, we generate the corresponding

[2]In CSS, an inch is *always* 96 pixels, regardless of the resolution of presentation devices. In fact, the definition of pixel in CSS is different than the conventional definition of physical pixels in presentation devices. To know more about this, please refer to [Con16b].

shorthand declaration, and add it as a "virtual" declaration to the corresponding style rule in the model. These virtual shorthand declarations will be compared with "real" shorthand declarations to detect type III duplication instances.

4.3.3 Duplication Detection

Duplication instances can be found by comparing every possible pair of declarations in the CSS model and checking whether they are equal (for type I) or equivalent (for type II and III). The notions of equality and equivalence were discussed in Section 4.2. Note that in our approach we consider the declarations present in the CSS model as they have been formed after the preprocessing steps to allow for the detection of type II and III duplication instances.

Our detection approach is summarized in Algorithm 1. The algorithm receives as input a preprocessed CSS style sheet and returns a set of *clones*, where each *clone* is a set of equal or equivalent declarations, $D = \{d_1, d_2, ..., d_n\}$, and each declaration belongs to a style rule $s_i \in S = \{s_1, s_2, ..., s_n\}$ of the analyzed CSS style sheet.

Figure 9 depicts an example of a style sheet and the corresponding clones extracted from the application of Algorithm 1. The first two clones contain instances of type II duplication, while the last clone contains an instance of type III duplication.

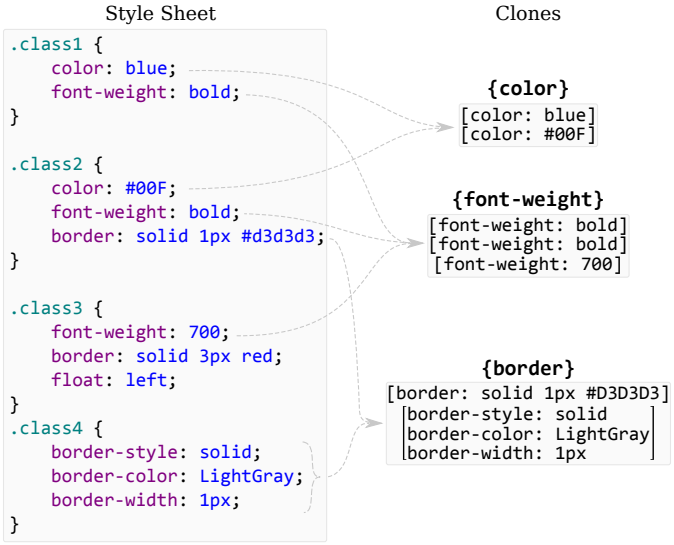


Figure 9: Clones extracted from a style sheet

4.3.4 Extracting Refactoring Opportunities

A *clone*, as defined in subsection 4.3.3, can be directly refactored by extracting a single declaration $d_i \in D$ to a new style rule, which groups all selectors in S, and then removing all declarations in

Algorithm 1: Detection of type I, II & III clones

Input : A preprocessed style sheet styleSheet

Output: allClones including Type I, II & III clones

```
1  mediumTypes ← all medium types in the styleSheet
2  allClones ← ∅
3  foreach m ∈ mediumTypes do
4      D ← all declarations in m
5      clones_m ← ∅
6      for i ← 1 to |D| do
7          clone ← D_i
8          for j ← i + 1 to |D| do
9              if identical(D_i, D_j) ∨ equivalent(D_i, D_j) then
10                 clone ← clone ∪ D_j
11             end
12         end
13         if |clone| > 1 then
14             merged ← false
15             foreach clone_k ∈ clones_m do
16                 if clone_k ∩ clone ≠ ∅ then
17                     clone_k ← clone_k ∪ clone
18                     merged ← true
19                 end
20             end
21             if not merged then
22                 clones_m ← clones_m ∪ clone
23             end
24         end
25     end
26     allClones ← allClones ∪ clones_m
27 end
```

D from the original style rules they belong to. The refactored version of the CSS code will contain $|D| - 1$ less declarations, but should have exactly the same effect in terms of the styles applied to the selected elements. As such, the larger the clone (i.e., the cardinality of D), the more beneficial the corresponding refactoring is, since a larger number of declarations will be eliminated by the application of the refactoring.

The detected clones constitute the "building blocks" for extracting more advanced and higher impact refactoring opportunities. For instance, there may exist style rules that have multiple declarations in common (i.e., style rules involved in multiple clones). A set of common declarations shared among a group of style rules constitutes a *clone set*. In that case, all declarations in the clone set could be extracted into a single style rule with grouping selector (or having a *class selector*) reducing significantly the repetition of declarations. Figure 7 in Section 4.2.2 presents an example of such a case. In general, the more clones are common in a larger set of style rules, the higher the impact of the corresponding refactoring opportunity in the reduction of repeated declarations.

In this work, we use a data mining metaphor to extract clone sets as refactoring opportunities from the initially-detected declaration-level clones. Let us assume that the style sheet is a transactional dataset, in which every style rule s_i is a transaction, and the clones corresponding to the declarations of s_i are the *items* of transaction s_i. Based on this mapping, Figure 10 shows the resulting dataset for the style sheet of Figure 9. Note that the clones are sorted according to their size (i.e., the number of duplication occurrences).

Transactions (Selectors)	Items (Corresponding clones)
`.class1`	`{font-weight}` `{color}`
`.class2`	`{font-weight}` `{color}` `{border}`
`.class3`	`{font-weight}`
`.class4`	`{border}`

Figure 10: Dataset for the style sheet of Figure 9

In the data mining domain, a set of items which is repeated in different transactions is called an *itemset*. If an itemset is repeated in more than a certain number of transactions, which is called the *minimum support count*, the itemset is known to be *frequent*. Our goal is to extract all frequent itemsets with a minimum support equal to 2 (i.e., the minimum size for a duplication instance), because a frequent itemset in our case represents a clone set that is repeated in more than one style rule. Therefore, every frequent itemset is a potential *grouping refactoring opportunity*.

In our method, we use the FP-Growth association rule mining algorithm [HPY00], which finds the frequent itemsets using a structure called *frequent-pattern tree* (FP-Tree) [TSK05]. The FP-Tree is essentially a compact representation of the dataset, since every itemset association within the dataset is mapped onto a path in the FP-Tree. Figure 11 displays the FP-Tree for the dataset

of Figure 10.

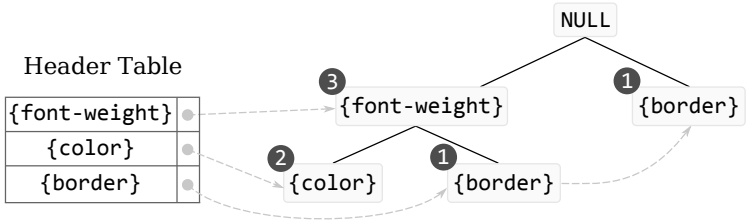


Figure 11: FP-Tree for the dataset of Figure 10

The FP-Tree has a *header table*, which includes all distinct items that exist in the FP-Tree. The items in this table are sorted in descending order based on their support count. There is a link between every item in this table to the first occurrence of that item in the FP-Tree (represented as a dotted arrow from the header table to the nodes of the FP-Tree in Figure 11). To enhance the traversal of an item in the header table to all nodes containing that item, nodes that contain the same item are also linked (e.g., the `border` nodes in the FP-Tree).

The number next to a node represents the number of transactions (style rules in our case) involved in the portion of the path from this node to the root of the tree. For example, the path from node `color` to the root represents that there are two style rules that contain both items `color` and `font-weight` (i.e., style rules with selectors `.class1` and `.class2`). The path from node `border` nested under node `font-weight` to the root represents that there is only one style rule that contains both items `border` and `font-weight` (i.e., style rule `.class2`). Finally, the path from node `font-weight` to the root represents that there are three style rules that contain item `font-weight`.

It is worth mentioning that, in the data mining domain, usually it is not important *in which transactions the itemsets are frequent*; instead, it suffices to know *how many times the itemsets appear together*. As a result, in the original FP-Tree representation, only the number of transactions in which the itemsets appear is kept. In our application, however, we have to know the style rules in which the set of repeated style declarations repeated. Consequently, we have tweaked the FP-Tree data structure to also store this information.

Once the FP-Tree is constructed, the FP-Growth algorithm generates all frequent itemsets with the minimum support specified as input. Figure 12 shows all frequent itemsets (i.e., grouping refactoring opportunities) generated with a minimum support value equal to 2.

In the refactoring scheduling literature, two refactorings are considered as *conflicting* if they have a mutually exclusive relationship [MTR07], i.e., the application of the first refactoring disables the application of the second refactoring and vice versa. Within the context of CSS, two refactoring opportunities are conflicting if their application affects a common subset of declarations. For instance, in Figure 12, if the last refactoring opportunity is applied, the third one becomes infeasible and vice

	Frequent Itemsets/ Refactoring Opportunities	Involved Selectors
1	`[{border}]`	`.class2, .class4`
2	`[{color}]`	`.class1, .class2`
3	`[{color}, {font-weight}]`	`.class1, .class2`
4	`[{font-weight}]`	`.class1, .class2, .class3`

Figure 12: Output of the FP-Growth algorithm for the style sheet of Figure 9

versa, because these two refactoring opportunities affect two common `font-weight` declarations. In the same manner, the second and third refactoring opportunities are also conflicting, because they affect two common `color` declarations. However, in that case, the third refactoring opportunity *subsumes* the second one, since the set of declarations affected by the latter is a subset of the declarations affected by the former. Our approach filters out subsumed refactoring opportunities, if the ones subsuming them can be safely applied (Section 4.3.6). For the problem of conflicting refactoring opportunities, we provide a ranking mechanism explained in Section 4.3.5.

4.3.5 Ranking Refactoring Opportunities

Although a refactoring operation affects several quality aspects of the code, such as understandability, maintainability, and extensibility, in this work we focus on the size of the CSS code, because size is directly associated with the other aforementioned higher-level quality attributes (in general, a code with small size can be more easily maintained). Hence, in order to prioritize the refactoring opportunities and allow developers to focus on the most important ones, we define a ranking formula based on the number of characters that can be removed from the CSS code by applying a given refactoring opportunity.

Let RD_r be the set of duplicated declarations that will be removed from the style sheet by applying the refactoring opportunity r, S_r be the set of style rules that contain the duplicated declarations of set RD_r (i.e., the style rules that will be grouped after applying r), and AD_r be the set of declarations that will be added to the new style rule with the grouping selector. It should be noted that AD_r contains the declaration with the *minimum* number of characters for each set of equal/equivalent declarations within RD_r. The size reduction (SR) achieved by refactoring opportunity r is calculated as follows:

$$SR(r) = \sum_{d \in RD_r} c(d) - \sum_{s \in S_r} c(s) - \sum_{d \in AD_r} c(d) \quad (1)$$

where the function c counts the number of characters of the declaration (or the selector of the style rule) passed as an argument. The higher the $SR(r)$ value, the higher the impact of r will be

on reducing the size of the CSS code. A negative $SR(r)$ value indicates that the size of the CSS code will increase after the application of r. A negative value is possible if the textual size of the resulting style rule with grouping selector is larger than the textual size of the declarations being removed. Of course, this would not be an issue if the duplicated declarations were placed under a newly defined class; however, this solution would require to update the target documents to make use of the new class. As mentioned before, in this work we aim to avoid modifications of the target document, i.e., all refactorings should be merely within the style sheets. Consequently, when size reduction is the objective, the refactoring opportunities should be applied in a descending order of SR value excluding those having a negative SR value.

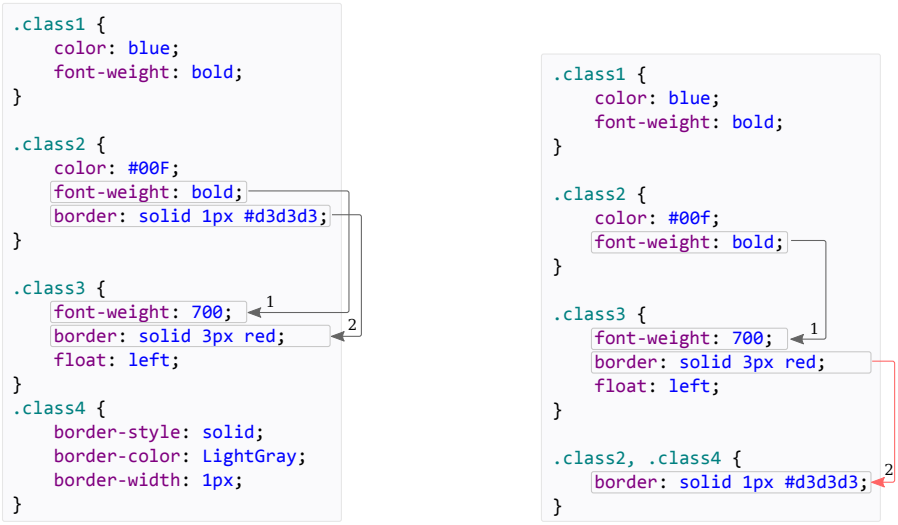
Based on Equation 1, the size reduction values for the four refactoring opportunities shown in Figure 12 are 46, -3, 14, and 13 characters, respectively. The second refactoring opportunity would actually increase the size of the CSS code, if it was applied. The first refactoring opportunity corresponds to the highest size reduction, and the CSS code resulting after its application is shown in Figure 13b (the new style rule with grouping selector is appended to the end of the file).

4.3.6 Preserving Order Dependencies

Behavior preservation is a crucial property of refactoring [Opd92]. The refactored program should have exactly the same functionality as the original program. Within the context of CSS, the notion of *behavior* corresponds to the *presentation* of the target documents (i.e., the style values that are eventually applied to each of the style properties of the target document elements). Therefore, a refactoring can be considered as valid, if its application preserves the *presentation* of the target documents.

Let us assume that the CSS code of Figure 13a is applied to the target document shown in Figure 14a. As we can observe from Figure 14a, the second `div` element uses the style declarations from both style rules corresponding to `.class2` and `.class3` selectors. As we can see from Figure 13a, the declaration of the `border` property in `.class3` overrides the corresponding declaration in `.class2` and as a result, the second `div` element is styled with a red color border as shown in Figure 14b.

Now, let us assume that the first refactoring opportunity shown in Figure 12 is applied to the CSS code of Figure 13a resulting in the CSS code of Figure 13b. In the refactored CSS code, the declaration of the `border` property in the extracted style rule having the grouping selector `.class2,.class4` overrides the corresponding declaration in `.class3`. As a result, the second `div` element is no longer styled with a red color border as shown in Figure 14c, which is a clear indication that the applied refactoring did not preserve the presentation of the target document. This inconsistency is caused by the inversion of the original *overriding relationship* between the style



(a) Order dependencies in the original CSS file (b) Order dependencies after refactoring

Figure 13: Order dependencies before and after refactoring

declarations defined in the style rules `.class2` and `.class3` after the application of the refactoring.

We define an *order dependency* from style rule s_i containing declaration d_k to the style rule s_j containing declaration d_l due to property p, denoted as $\langle s_i, d_k \rangle \xrightarrow{p} \langle s_j, d_l \rangle$, iff:

a) the selectors of the style rules s_i and s_j select at least one common element having property p in the target document,

b) declarations d_k and d_l set a value to property p and have the same importance (i.e., both or none of the declarations use the `!imporant` rule),

c) declaration d_k precedes d_l in the style sheet,

d) the style rule s_i and s_j have selectors with the same specificity.

As it can be observed, the order dependencies are extracted based on the *cascading rules* defined in the CSS specifications, discussed in detail in Chapter 2.

To ensure that the presentation of the target documents is preserved, we define the following *precondition*:

> *The extraction of the style rule having the grouping selector should preserve all order dependencies among the style declarations of the style sheet.*

```
...
<div class="class1 class4">
    content1
</div>

<div class="class2 class3">
    content2
</div>
...
```

(a) Sample HTML document

content1
content2

(b) Styling using the CSS code of Figure 13a

content1
content2

(c) Styling using the CSS code of Figure 13b

Figure 14: Breaking presentation semantics with improper refactoring

The problem of finding an appropriate position for the extracted style rule g in the style sheet can be expressed as a *Constraint Satisfaction Problem* (CSP) defined as:

Variables The positions of the style rules involved in order dependencies including g.

Domains The domain for each variable is the set of values $\{1, 2, ..., N+1\}$, where N is the number of style rules in the original style sheet.

Constraints Assuming that g contains style declarations for the set of properties P, an order constraint is created in the form of $pos(s_i) < pos(s_j)$ for every order dependency $\langle s_i, d_k \rangle \xrightarrow{p} \langle s_j, d_l \rangle$ where $p \in P$.

In the example of Figure 13a, the order dependencies are

$$\langle \texttt{.class2}, \texttt{font-weight: bold} \rangle \xrightarrow{font-weight} \langle \texttt{.class3}, \texttt{font-weight: 700} \rangle$$

and

$$\langle \texttt{.class2}, \texttt{border: solid 1px \#d3d3d3} \rangle \xrightarrow{border} \langle \texttt{.class3}, \texttt{border: solid 3px red} \rangle$$

and we extract the following constraint:

$$pos(\texttt{.class2}) < pos(\texttt{.class3})$$

Based on this constraint, the extracted style rule with grouping selector `.class2, .class4` should be placed at any position before the style rule with selector `.class3` (i.e., `.class3` should be the last style rule in the style sheet after refactoring) in order to preserve the presentation of that target document in Figure 14a.

If we assume that there is an additional order dependency from `.class3` to `.class4` due to property `border`, then the CSP would be unsatisfiable due to the new following constraint:

$$pos(\texttt{.class3}) < pos(\texttt{.class4})$$

In that case, the extracted style rule with the selector `.class2,.class4` has to be placed before `.class3` to satisfy the first constraint and after `.class3` to satisfy the second constraint, and thus there is no solution satisfying both constraints. Refactoring opportunities leading to an unsatisfiable CSP violate the defined precondition, and therefore are excluded as non presentation-preserving.

4.4 Evaluation

To assess the efficacy of our approach, we conducted a case study addressing the following research questions:

RQ1: What is the extent of declaration-level duplication in CSS files?

RQ2: What is the number of refactoring opportunities that can be potentially applied in CSS files and how many of them are actually presentation-preserving?

RQ3: What is the size reduction we can achieve by applying presentation-preserving refactorings in CSS files?

Our tool and empirical data are all available online [MTM14b].

4.4.1 Experiment Design

Selection of subjects

In total, our study contains 38 subjects. In order to select representative real-world web applications, we adopted the web-systems included in the study conducted by Mesbah and Mirshokraie [MM12], in which they investigated the presence of unused CSS code. The list includes 15 (in total) open-source, randomly-selected, and author-selected online web applications. We included 14 subject systems from that list (one of them was not available online anymore, at the time of conducting this study). We extended the list with 24 more subjects including web applications developed by companies considered leaders in web technologies, such as Facebook, Yahoo!, Google, and Microsoft, in addition to a subset of the top-100 visited web sites based on Alexa ranking. The complete list of the selected systems is shown in Table 2.

Figure 15 shows the size characteristics of the CSS code, selectors (i.e., style rules), and declarations of the subjects included in our study. As it is observed, the subjects are quite diverse in terms of size.

Table 2: Selected subjects

ID	Web app / Website	#CSS Files	ID	Web app / Website	#CSS Files
1	Facebook	6	20	Pinteerst	2
2	YouTube	4	21	Reddit	1
3	Twitter	2	22	Tumblr.com	2
4	YahooMail	3	23	Wordpress.org	1
5	Outlook.com	6	24	Vimeo.com	3
6	Gmail	5	25	Igloo	2
7	Github	2	26	Phormer	1
8	Amazon.ca	3	27	BeckerElectric	1
9	Ebay	2	28	Equus	1
10	About.com	1	29	ProToolsExpress	1
11	Alibaba	3	30	UniqueVanities	3
12	Apple.ca	3	31	ICSE12	3
13	BBC	3	32	EmployeeSolutions	3
14	CNN	1	33	SyncCreative	3
15	Craiglist	1	34	GlobalTVBC	5
16	Imgur	2	35	Lenovo	1
17	Microsoft	1	36	MountainEquip	2
18	MSN	1	37	Staples	2
19	Paypal	1	38	MSNWeather	3

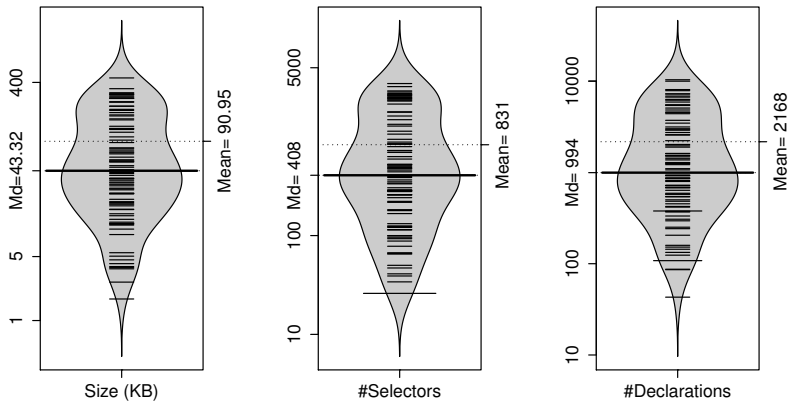


Figure 15: Characteristics of the analyzed CSS files

Extraction of CSS code and DOM states

As mentioned, CSS code can be directly embedded in the web documents (i.e., embedded or internal CSS), linked to web pages as external files, or dynamically-generated (or linked) at runtime through JavaScript code. For our experiments, we focus on the external CSS files, since the refactoring of the other sources of CSS code requires the modification of other web artifacts (such as HTML documents), which is not the focus of our technique.

We take advantage of the dynamic analysis features provided by Crawljax [MvDL12] and developed an external CSS file extractor plug-in on top of it. Additionally, we use Crawljax to dynamically capture different DOM tree instances (i.e., DOM states) from the examined web applications and use them for the extraction of order dependencies between the CSS style rules.

Detection of presentation-preserving refactorings

In order to collect the set of presentation-preserving refactorings that can be applied on a CSS file f styling the set of DOM states S collected from a web application, we:

1. Extract the order dependencies between the style rules of f by analyzing the DOM states in S, as described in Section 4.3.6.
2. Extract the set of refactoring opportunities R that can be potentially applied to f.
3. Sort R based on size reduction (Formula 1) and remove the refactoring opportunities having a negative value.
4. Iterate through the elements of R and apply the first refactoring opportunity for which the CSP defined in Section 4.3.6 is satisfiable.
5. If step 4 results in the application of a refactoring, repeat steps 1-5 with the refactored CSS file f'.

4.4.2 Results

Extent of duplication in CSS declarations (RQ1)

The results of our empirical study confirm the expectation that duplication is more extensive in CSS code compared to procedural and object-oriented code (with 5–20% duplicated code [RC07]). Figure 16 displays a violin plot with the percentage of the declarations that are involved in at least one clone (i.e., they are at least once duplicated) in the analyzed style sheets. The median value for the percentage of duplicated declarations is 68%, while the average is 66%. The vast majority of the examined style sheets exhibits a duplication ranging from 40% to 90%. Note that in the reported

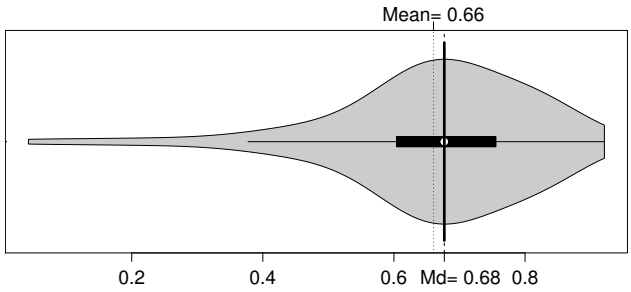
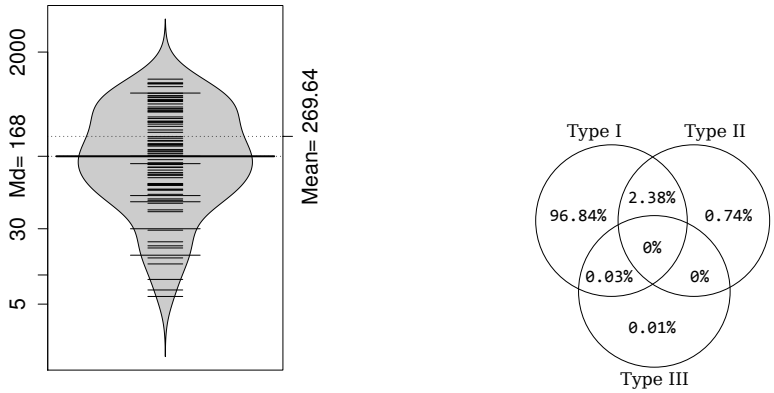


Figure 16: Ratio of the duplicated declarations

results we have set the minimum support count (i.e., the minimum number of style rules that should share a common declaration) to the lowest possible value (equal to 2); setting a larger minimum support value would lead to lower duplication rates. As we will see in Chapter 5, developers are interested in using abstraction techniques provided by CSS preprocessors to avoid repeating style declarations, even when they are repeated across as few as two style rules.

Figure 17a shows the number of clones detected in the analyzed CSS files. On average, there are 270 distinct declarations being repeated more than once in the examined style sheets that could be used as building blocks for extracting more advanced refactoring opportunities. The Venn diagram shown in Figure 17b displays the percentage of the clones including different combinations of the duplication types defined in Section 4.2. As it can be observed, 97% of the clones include only type I duplication instances, while 2% of the clones include a combination of type I and II duplication instances. Furthermore, the existence of type III duplication instances within the clones is very rare.



(a) Total number of detected clones (b) Duplication types in the detected clones

Figure 17: Statistics for the detected clones

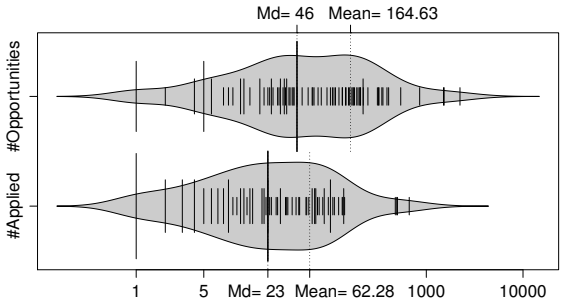
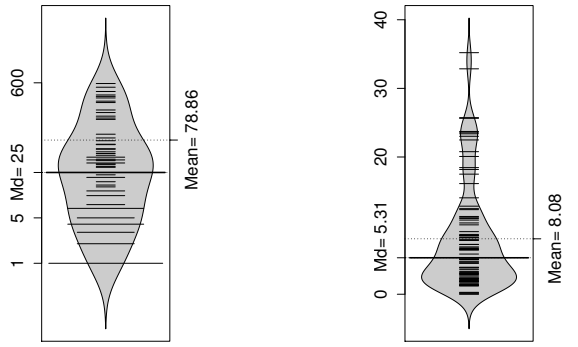


Figure 18: Initial refactoring opportunities vs. applied presentation-preserving refactorings



(a) #Order dependencies (b) %Size reduction

Figure 19: Order dependencies and size reduction

Refactoring opportunities in CSS (RQ2)

Figure 18 shows on top, a bean plot of the number of refactoring opportunities that were initially extracted from the original CSS files, excluding refactoring opportunities being subsumed and/or having a negative size reduction value. On the bottom of Figure 18, we can see a bean plot of the number of presentation-preserving refactorings, which we actually applied on the CSS files. As it can be observed, our approach was able to detect, on average, 165 refactoring opportunities in the original version of the examined CSS files, while the average number of presentation-preserving refactorings was 62. Additionally, we found out that the examined CSS files had 79 order dependencies on average between their style rules, as shown in Figure 19a.

Size reduction (RQ3)

In Figure 19b, we have depicted a bean plot with the percentage of the size reduction achieved by applying *only* presentation-preserving refactorings. In the examined CSS files, the average size reduction was 8%, while the maximum achieved value was 35%. Overall, in 12% of the examined CSS files (11 out of 91) the size reduction was over 20%, while in 27% (25 out of 91) the size

reduction was over 10%.

In order to determine the factors that influence the applicability of refactorings in the examined CSS files, we decided to build a statistical regression model. Regression models are mostly used for the purpose of prediction, where the values of one or more *predictor variables* can be used to predict the value for the *response variable*. However, a multiple linear regression model can be also used to assess the impact of one predictor on the response variable, while controlling the other predictors [DWC04]. Using regression, we estimate a coefficient for each predictor, which shows the magnitude and direction of the effect of the predictor on the response variable.

We built a model with the *number of applied refactorings* as the response variable, and *size* and the *number of order dependencies* as predictors (note that, size is measured in Kilobytes, on the syntactically-formatted CSS files). Intuitively, we expect a positive relationship between the number of applied refactorings and the size of the CSS files, since larger files exhibit more duplication and thus offer more opportunities for refactoring. On the other hand, we expect a negative relationship between the number of applied refactorings and the number of order dependencies detected for a given CSS file, since a larger number of order dependencies implies a higher probability for a precondition violation and thus rejecting a candidate refactoring opportunity. To this end, we created a generalized linear model of the *Poisson* family with the *log* link function [ZKJ08], which is a reasonable choice due to the nature of the response variable, which is *count data*.

As it is shown in Table 3, all estimated coefficients are statistically significant, and as we expected, the coefficient for the size of the CSS files is positive, while the coefficient for the number of order dependencies is negative. More precisely speaking, an additional order dependency that exists in a CSS file will multiply the number of applicable refactoring opportunities by $e^{-1.195e-03} = 0.9988057$. Similarly, one Kilobyte increase in the size of a CSS file will multiply the number of applicable refactoring opportunities by $e^{8.149e-03} = 1.008182$. From this result, we can conclude that for CSS files with a similar size, the number of applicable refactorings decreases as the number of order dependencies increases. Additionally, we can conclude that our approach is more effective in terms of size reduction for large CSS files with a limited number of order dependencies.

4.4.3 Comparison with Federman and Cook's approach [Dav10]

As mentioned in the related works chapter (i.e., Chapter 3), Federman and Cook applied Formal Concept Analysis (FCA) on CSS style sheets in order to group CSS declarations that are repeated across different CSS rules, and published the results in a non-peer-reviewed technical report [Dav10]. Unfortunately, the implementation of the Federman and Cook's approach (henceforth, the FCA-based approach) is not available, and therefore, we cannot make a fair comparison of the two approaches, with respect to the output (i.e., whether the FCA-based approach identifies the same

Table 3: Statistical model's estimated parameters

Parameter	**Estimate**	**p-value**
Intercept*	2.989	<2e-16
Size coefficient	8.149e-03	<2e-16
Number of order dependencies coefficient	-1.195e-03	<2e-16

* The intercept is the constant term in the regression model, which makes the residuals have a mean of zero.

refactoring opportunities as ours) or the scalability/performance. Therefore, we can only extract the differences/commonalities from the technical report.

The very first difference between the two approaches is that, instead of using FCA, we use a frequent itemset generation algorithm to group duplicated declarations. In the FCA-based approach, a CSS file is seen as a formal *context*. A context in FCA is composed of a set of *objects*, a set of *attributes* of those objects, and a binary relation describing whether an object possesses some certain attributes or not. Treating a CSS file as a formal context means seeing style rules as objects and style declarations as attributes, and the context's binary relation means whether or not some style declarations appear in a style rule. Each *FCA concept*, on the other hand, includes a subset of objects (i.e., the extent) that share a subset of all attributes (i.e., the intent). Consequently each concept in the FCA-based approach is equivalent to a duplication refactoring opportunity, where a set of style declarations are shared by a set of style rules.

Formal concepts can be organized in a hierarchical manner: a super-concept in the hierarchy contains a superset of the objects of its sub-concepts. This creates a partial order which satisfies the axioms of a *lattice*, i.e., the *concept lattice*. Federman and Cook take advantage of the concept lattice of a CSS file to extract duplicated code. When one concept (i.e., a refactoring opportunity) is selected for extraction, its relationship with other concepts is inferred from the lattice (e.g., conflicting refactorings). The concepts for extraction are selected by traversing the concept lattice. Different traversals can lead to different sequence of refactorings, as discussed by Federman and Cook.

While in theory FCA can also be used to generate frequent itemsets [Smi09] (and, as a result, the two approaches can generate the same refactorings), the FCA-based approach of Federman and Cook appears to be suffering from several shortcomings, namely:

1. The definition of the *equality* of style declarations in the FCA-based approach is based on the *exact* similarity. In other words, two style declarations `color: red` and `color #f00` are deemed to be different in their approach, while these two style declarations are semantically

the same. In our approach, however, style declaration equality has a broader definition that includes the *equivalence* of style declarations, as well as equality.

2. While FCA-based approach attempts to keep the presentation semantics of the refactored CSS code intact, it only considers obvious dependencies between style declarations within the CSS code, and does not take into account target documents (and also JavaScript) and the hidden dependencies that they might create in CSS code. The FCA-based approach, consequently, can re-order style declarations in the CSS file in a presentation-breaking way.

3. The FCA-based approach does not take into account media at-rules. As mentioned before, today's CSS code bases use media at-rules to a large extent to implement responsive web pages that are presented consistently across different media. Our approach, in contrast, takes care of media at-rules.

4. The FCA-based approach pre-processes CSS code and converts all the existing style rules with *grouping selectors* to multiple rules with *simple selectors*, repeating all the style declarations for each style rule with simple selector, and then apply FCA on them. As an example, consider the following style rule:

```
.c1, .c2  {
  color: red;
}
```

   This preprocessing converts the mentioned style rule to the following two style rules:

```
.c1 {
  color: red;
}
.c2 {
  color: red;
}
```

   The refactoring approach is actually supposed to do the reverse. Indeed, the real motivation behind this decision is unclear in the report, and seems to have root in a technical limitation in the implementation of the used CSS parser/FCA library. This essentially can lead to changing the order of selector names in the style rules. In our work, in contrast, we try to make as little changes as possible in the CSS file, by not allowing such changes.

5. Lastly, the FCA-based approach has been evaluated on artificial CSS code, while we tested our approach on real CSS files collected from several popular web applications.

4.4.4 Discussion

CSS duplication and refactoring opportunities

Our case study shows that CSS code duplication is prevalent in today's web systems. The majority of the clones we found pertain to type I duplication instances, and type II and III duplications are relatively less common. This indicates that developers use the same representation for style values consistently throughout their style sheets. Additionally, they make use of shorthand-property declarations consistently within different style rules. The results of our evaluation also show that our method is able to successfully detect many CSS refactoring opportunities that remove duplications and preserve the initial presentation of the target documents. These refactorings, when applied, allow for a much cleaner CSS code and considerable size reduction.

Size reduction

There are some considerations regarding the use of size reduction as a measure for evaluating our approach, which we discuss in this subsection.

- In the industry, there are several CSS *minifiers* available for the developers that attempt to reduce the size of CSS files by applying simple transformations, e.g., removing unnecessary white spaces or semicolons. As also mentioned in the related works section (Chapter 3, page 16), Bosch et al. [BGL14a, BGL14b, BGL15] also introduced an approach for reducing the size of CSS files by removing redundant style declarations and rules based on static analysis. To our knowledge, both CSS minifiers and the Bosch et al.'s approach only analyze CSS files and do not consider target documents for exploring other possibilities for size reduction. Our method complements (and not replaces) the mentioned tools, by suggesting refactorings which are presentation-preserving since the DOM states of the target documents are also considered.

- In our method, we focus on refactoring opportunities that extract the same set of equivalent declarations in a style rule with a *grouping selector*. We mentioned that an alternative approach would be to extract and group the declarations in a new style rule with a *class selector* instead. By selecting an appropriate name for the class selector, we can reduce even further the size of the CSS file (i.e., by replacing a set of selector names with a single class name), and at the same time improve its understandability (the class name could represent a common concept being extracted). However, this approach requires making use of the new class in the DOM elements of the target document. From the refactoring point of view, this approach should update the corresponding HTML documents for static web sites, or even the source code that generates the HTML elements for dynamic web sites. Alternatively, there could be a shorter

(e.g., one *simple* selector) that selects exactly the same elements as what the grouping selector of the newly-extracted style rule selects.

- Finding a sequence of refactoring applications that optimizes size reduction is somehow *greedy*. Applying always the refactoring with the highest *immediate* size reduction does not guarantee that the resulting sequence of refactorings leads to the maximum size reduction. A possible explanation is that the application of some refactorings at the beginning of the sequence could make infeasible the application of subsequent refactorings eventually leading to a solution with higher size reduction. The ordering of refactoring applications can be treated as an optimization problem that can be solved using search techniques.

- Most of the today's web applications use web servers that employ general-purpose content compression algorithms (e.g., GZIP) for reducing the size of the files sent to the clients. These compression algorithms actually work better in the presence of more redundancy in files. As a result, when using compression algorithms, the proposed size reduction algorithm in this chapter cannot always lead to smaller files. Conversely, the original files compressed by the web server can have smaller size compared to the compressed version of the refactored file using our approach. Indeed, in a preliminary study, we observed that applying only the GZIP algorithm will create smaller CSS files in 69% of cases, and only in 31% of cases applying our approach followed by the GZIP algorithm lead to smaller files.

  Nevertheless, not all web servers/clients provide compression features. Moreover, using CSS classes instead of grouping selectors (or equivalent selectors with fewer characters) for the proposed approach may yield a better size reduction after compression. In addition, using content compression on the client and server sides to compress and decompress CSS files has performance and energy overhead, and using a content-specific size-reduction approach (like our proposed approach) can decrease the required computation resources.

  Alternatively, in the cases where the proposed refactorings cannot lead to smaller file sizes, one can extract the duplicated style declarations into *mixins* (i.e., function-like constructs in CSS preprocessor languages, that will be discussed in detail in the next sections).

Limitations

In our current implementation, we have only considered CSS files linked to the HTML documents. In order to provide complete CSS refactoring support, in the future, we will also include in our analysis CSS styles embedded inside the `<style>` tags of the web pages (i.e., internal CSS).

In addition, the CSS files used in our study could be generated code, and generated code intuitively has more duplication than hand-written code. We do not have access to the production

code of several web applications used in this study (e.g., Facebook or Google), and we do not have a general way to understand whether a CSS file attached to a web application is generated or not.

Moreover, one should note that refactoring CSS for removing duplicated code by *only* using its internal features is inherently limited, since style declarations with differences in style values (e.g., `color: red` and `color: blue`) cannot be grouped together. For removing such duplications, we will need to be able to *parameterize* style declarations, i.e., we would need function-like constructs that do not exist in CSS.

Threats to the validity

A threat to the *internal validity* is that the DOM states collected from each web application may be insufficient to extract all possible order dependencies between the style rules of the examined CSS files, since for some dynamic web applications the number of DOM states is practically infinite [MvDL12]. Missing order dependencies from unvisited DOM states could make some of the applied refactoring opportunities to be non-presentation-preserving for this particular set of unvisited DOM states. This has root in the way we configure the crawler to visit different DOM states. We used the default configuration of CRAWLJAX for this study.

To avoid selection bias, we selected 14 subjects from the list of web sites analyzed in a related CSS study[MM12]. To mitigate threats to the *external validity* and make the results of the experiment as generalizable as possible, we included 24 additional web sites developed by leading companies in web technologies applying the current state-of-the-art CSS development practices. Finally, the developed tool and the collected data are all available online to enable the replication of the experiment by other researchers.

4.5 Chapter Summary

In this chapter, we presented a technique for the detection of refactoring opportunities that can eliminate duplicated CSS declarations in a presentation-preserving manner, i.e., without side-effects in the styling of the target web documents. We performed an experiment on 38 real web applications and found that (1) code duplication is extensive in CSS files; on average 66% of the style declarations are repeated at least once, (2) there is a significant number of presentation-preserving refactoring opportunities in CSS files (62 on average) that is associated positively with the size of the CSS files and negatively with the number of order dependencies between the style rules of the CSS files, and (3) on average a 8% reduction in the size of the examined CSS files can be achieved by applying the detected refactoring opportunities.

As we discussed, some types of duplicated style declarations cannot be refactored directly within

CSS. However, CSS preprocessor languages offer several abstraction mechanisms that allow eliminating more advanced types of duplicated code, and as a result, higher maintainability might be gained by using these languages.

Notwithstanding, it is first interesting to see how the features that CSS preprocessor languages offer on top of pure CSS (e.g., the abstraction mechanisms) are utilized. If developers consistently take advantage of these features, we are motivated to devise automatic techniques for *migrating* existing CSS code bases to gain the maintainability improvements that using CSS preprocessor languages offer.

In the next chapter, we will describe some of these CSS preprocessor language features, and will provide the results of a large-scale empirical study on the use of these features by developers.

Chapter 5

An Empirical Study on the Use of CSS Preprocessors

5.1 Introduction

As discussed in the previous chapters, CSS preprocessor languages were introduced by the industry as a response to the missing features of CSS. The code written in a CSS preprocessor can include variable and function declarations, which can be used, e.g., inside CSS style rules. The preprocessor compiler essentially transforms (i.e., *transpiles*) the function calls and variable uses to pure CSS.

Currently, there is a long list of CSS preprocessors offering very similar features with a different syntax (e.g., Sass [Cat06], Less [SF10], Google Closure StyleSheets [Goo15], HSS [Ser10]), and their use is becoming a fast growing trend in the industry. An online survey with more than 13,000 responses from web developers, conducted by a famous website focusing on CSS development showed that around 54% of web developers use a CSS preprocessor in their development tasks [Coy12]. United States Federal Government advises front-end web developers who design websites for government services to use Sass as their Style Sheet development language in order to get "resources such as frameworks, libraries, tutorials, and a comprehensive styleguide as support" [Uni15].

While CSS preprocessors are popular among developers and they include several useful features, we do not have enough knowledge about how developers take advantage of these features in real web applications. Having such information can be useful for different reasons:

- A considerable number of web developers is still coding directly in pure CSS. Therefore, *migrating* existing CSS code to take advantage of preprocessor features (e.g., extracting duplicated declarations to a function in a CSS preprocessor) is greatly demanded in the industry. There are indeed several storied about such migration activities – among others, at GitHub [Ble15]

and Etsy [Na15], two leading companies on the web. Knowing the practices applied by web developers when coding in preprocessors will certainly help in developing more useful and efficient migration strategies.

- CSS preprocessors might be sub-optimally used, because web developers miss opportunities to further eliminate existing duplicated code and other bad practices. Therefore, there is a need for *refactoring recommendation* systems to help developers in improving the quality of their CSS preprocessor code. Knowing developers' practices will help in prioritizing the refactoring opportunities leading to the most commonly used solutions/patterns.

- Finally, the knowledge of developers' practices can also guide the CSS preprocessor language designers to revisit the design of these languages, e.g., by adding support for new features (which are currently implemented by developers in an ad-hoc manner), or making existing features easier to use, or eliminating features that are not adopted by developers.

These reasons motivate us for conducting the *first empirical study on the use of CSS preprocessors*. We have analyzed the preprocessor code of 150 websites, having their CSS code written in Less or Sass. We focused on these two preprocessors because, according to the results of an online survey [Coy12], Less and Sass are the most popular CSS preprocessors among web developers (92% of the developers who used a CSS preprocessor in their careers, preferred either Less or Sass). Additionally, to achieve more generalizable results, we analyzed Style Sheets written using both of the two dialects that Sass provides: 1) The initial syntax of Sass, which is closer to Python (decreases development effort by removing braces and commas, and relying on the indentation to show code blocks and nesting); and 2) The so-called *SCSS* syntax, which is more similar to the syntax of pure CSS. We selected 50 websites for each of these two dialects (accumulating to 100 websites for Sass preprocessor), in addition to 50 websites for Less.

In our analysis, we took into account the features of CSS preprocessors which are common in almost all preprocessors. These features include *variables*, *nesting*, *mixin* (i.e., function) calls and the *extend* construct.

Overall, in this chapter:

- We conduct the first empirical study on the use of CSS preprocessors and report our findings on 4 major preprocessor language features. We plan to use these insights to design refactoring/migration techniques for CSS and preprocessors.

- We make publicly available the dataset compiled from 150 websites to enable the validation and replication of our study, and facilitate future research on CSS preprocessors. We plan to use this dataset to evaluate the effectiveness and accuracy of our refactoring/migration techniques.

Note: Earlier version of the work done in this chapter has been published in the proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER 2016) [MT16a].

5.2 CSS Preprocessor Features

In this section, we briefly demonstrate some of the common features of CSS preprocessors, which are widely used by developers. All code examples are given using the Less syntax; the other CSS preprocessors use a similar syntax.

5.2.1 Variables

Supporting variables is one of the most basic features of traditional programming languages. At the time of writing this thesis, the CSS specifications for variables (called custom properties in CSS jargon) have not reached the "Recommendation" stage, and are not supported by some web browsers [Con15]. CSS preprocessors, however, have supported variables for a long time. Preprocessor variables can be defined, e.g., to store one or more *style values*, for instance `@color: red` (i.e., a single-value variable), or `@margin: 1px 2px 4px 3px` (i.e., a multi-value variable). Variables can be used for various purposes, such as theming (i.e., one style sheet representing different themes/colors).

Preprocessor variables are type-less; a value representing a color (e.g., `#FF00FF`) can be assigned to a variable which currently stores a dimension value (e.g., `2px`). Interestingly, some preprocessors also let developers to manipulate the value of variables by using arithmetic operators or by passing them to preprocessor built-in functions (e.g., making a color value darker using the `darken()` function in Less). Preprocessors also support the notion of *variable scope*. A variable can be defined in the *global* scope (i.e., visible in the entire style sheet), or in some *local* scope (i.e., visible inside the body of a style rule or a *mixin*).

In Figure 20 (left), a piece of Less code from the Semantic-UI[1] (version 1.6.2) is shown. The result of compiling this code is shown in Figure 20 (right).

5.2.2 Nesting

Preprocessors support a feature called *nesting*, which generates selectors using the following constructs in pure CSS:

[1]Semantic-UI is a CSS library used for building adaptive user interfaces for websites (`https://github.com/Semantic-Org/Semantic-UI`).

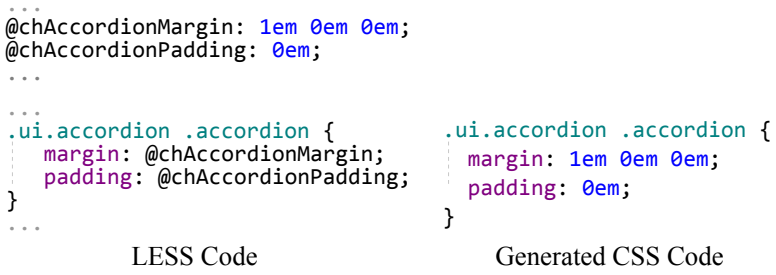


Figure 20: Variables in Less

Combinators make an existing selector `B` more specific, with respect to another selector `A`. As discussed in Chapter 2, CSS supports four combinators, namely the *descendant combinator* (denoted as `A B` – with a space between the two selectors), the *child combinator* (`A>B`), the *general sibling combinator* (`A~B`), and the *adjacent sibling combinator* (`A+B`).

Pseudo-Classes like `:hover` in the (`tr:hover`) selector.

Pseudo-Elements like `::first-line` in `p::first-line`.

As a real example of *nesting*, in Figure 21 (left), a code snippet from the Bootstrap CSS library[2] (version 3.3.1) is shown. The generated CSS code is shown in Figure 21 (right). As it can be observed, nesting avoids the repetition of `.navbar-toggle` selector and organizes relevant style rules in a hierarchical manner. The use of *nesting* leads to a more organized code by keeping relevant style rules in the same location. As we will see in Section 5.4, *nesting* is a very popular preprocessor feature.

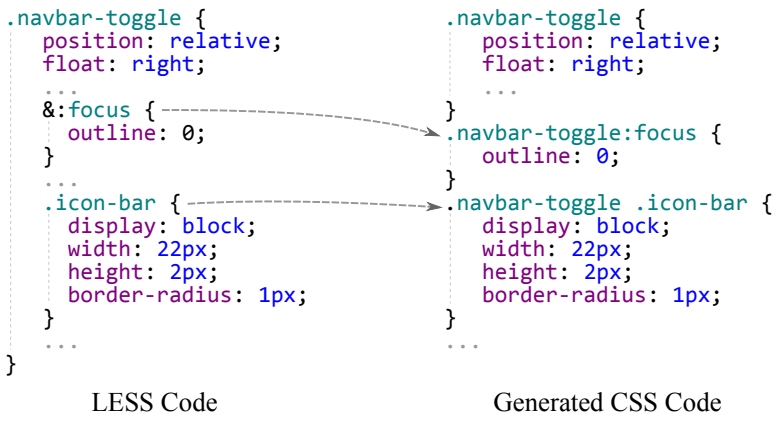


Figure 21: Nesting in Less

[2]The most famous CSS library which includes predefined classes for facilitating designing complex multi-column, responsive web pages, designed by developers at Twitter (`https://github.com/twbs/bootstrap`)

5.2.3 Mixins

As it was mentioned earlier, pure CSS does not support the notion of functions. CSS preprocessors have introduced a specific construct, called *mixin*, to mimic the behavior of functions. A *mixin* can be defined as a set of declarations, and can be called inside other constructs (such as a style rule or another *mixin*). The construct in which the *mixin* is called will include all the declarations of the called *mixin*. The declarations inside a *mixin* may have parameterizable values, therefore a *mixin* declaration can have parameters (just like a function in traditional languages). These parameters are preprocessor values, with the characteristics that were explained in Section 5.2.1. Arguments can be omitted, if *default values* are provided in the parameter declarations of a *mixin*.

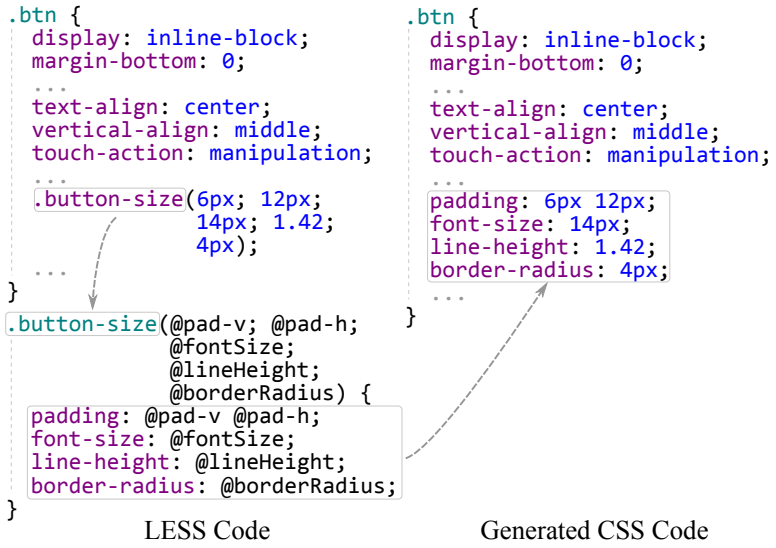


Figure 22: Mixin in Less

In Figure 22, a *mixin* is shown from the Bootstrap CSS library. As it is observed, after compiling this code, the declarations inside the *mixin* body appear in the style rule corresponding to selector `.btn`, and the parameters are replaced with the arguments passed to the *mixin*.

5.2.4 The "Extend" Construct

Remember that pure CSS includes two main mechanisms to avoid duplicated style declarations across style rules:

Creating classes A set of declarations can be grouped in a style rule with a *class selector*, associated with a class name. Such style rule will be applied on the elements in the target document having the same class name in their `class` attribute. For instance, the selector `.class1` can select the element `<div class="class1">` in the target document.

Grouping style declarations Discussed in detail in the previous chapter, a style rule having a *grouping selector* (which consists of a comma-separated list of two or more *base selectors*) can be used to apply a set of style declarations to all the elements selected by each of the simple selectors.

In CSS preprocessors, the *extend* construct is designed to play the same role towards avoiding duplication. The name of this construct was chosen to remind the extension feature of object-oriented programming languages like Java. In other words, the *extend* construct is used to "extend" the behavior of an existing style rule by adding more style declarations, while inheriting the existing style declarations from the extended style rule. When using the *extend* construct, the common declarations are placed inside a style rule with grouping selector in the generated CSS code.

Figure 23 (left) demonstrates the use of the *extend* construct in a piece of code from the Flat-UI design framework[3]. The compiled CSS code is shown in Figure 23 (right). As it is observed, the style rule which is extended (with selector `.dropdown-menu`) and the extending style rule (with selector `.select2-drop`) are grouped to share some common declarations in the generated CSS code, while the extra declarations appear in a separate style rule.

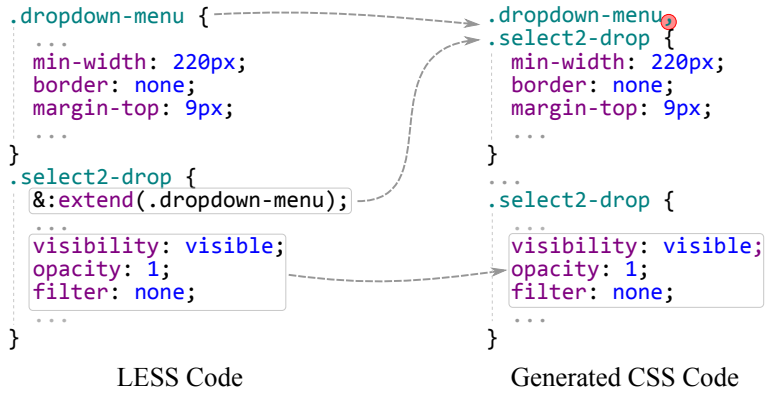


Figure 23: Extending style rules in Less

5.3 Experiment Setup

In this section, we provide information about the subject systems used in the study, as well as the process we followed for collecting the experimental data.

[3]A design framework based on Bootstrap which includes a set of easy-to-use predefined UI elements (`https://github.com/designmodo/Flat-UI`)

5.3.1 Subject Systems

Within the context of this study, we focused on websites, which make use of preprocessor languages and have their preprocessor code publicly available. We have deliberately avoided the analysis of preprocessor libraries and frameworks (such as Bourbon[4]), because their code is meant to be used externally by other projects, in the same way that public APIs are used. Adding such libraries in our analysis would affect negatively the validity of this study, since a large number of *mixin* declarations developed to be used externally, would appear as not being used at all (i.e., unreachable or dead code). Therefore, we decided to focus on websites having their own internal preprocessor codebase, and study them in isolation from potential external dependencies.

While it is not necessary for websites to make their preprocessor codebase publicly available to the end users, some web developers intentionally upload the preprocessor code along with the generated CSS code on the web. This might be done to enable the compilation of the preprocessor code on demand (either on the server- or client-side). We used Google's advanced search feature to find these preprocessor files. Particularly, we searched the Internet for files with the extensions `*.less`, `*.scss` and `*.sass`. This search query allowed us to find websites satisfying our selection criteria:

1. the website should have its CSS code generated by Less or Sass/SCSS, and
2. the website should publicly provide its preprocessor code along with the generated CSS code.

When we found a preprocessor file on a website, we manually attempted to extract the contents of the file's parent directory. If this directory was accessible, we collected all the preprocessor files inside it, and recursively all the preprocessor files inside its sub-folders. This was necessary to make sure that all the files which are imported using the `@import` directive are also collected. Then we manually found and marked the *main* Style Sheet files, i.e., the files that are passed to the preprocessor compiler to get the generated CSS files. This step is crucial, as we start our analysis from these main files and recursively parse and analyze the files which are imported from them.

More specifically, we collected 1266 preprocessor files, containing 255 Less, 427 Sass, and 584 SCSS files. The full list of the websites used for this study is given in Table 4.

In Figure 24, we have included box and violin plots of various size metrics for the collected files. The scale of the figures is logarithmic. Violin plots are useful for presenting the distribution of data. As it is observed, the examined Less, Sass and SCSS files have similar size characteristics.

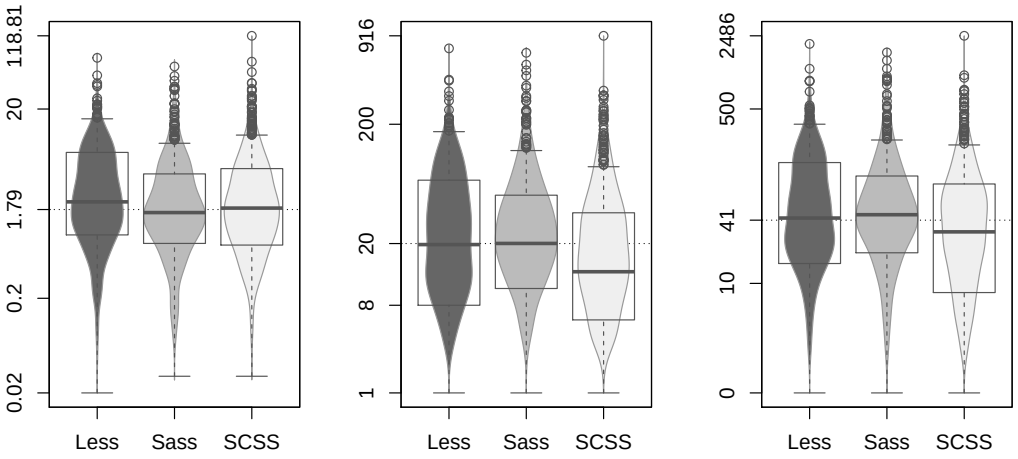
5.3.2 Data Collection

After collecting the preprocessor files, we applied the process depicted in Figure 25.

[4] A simple and lightweight mixin library for Sass (`http://bourbon.io`)

Table 4: List of the websites used in the study

	Less	**Sass**	**SCSS**
1	abundance.org	addare.ro	3x3mag.com
2	adriel.org	alexiadesigns.com	acphs.edu
3	aisandbox.com	allergycosmos.co.uk	ashrammoseleyha.org.uk
4	alexanderradsby.com	assets.nilsology.net	barajasinformatica.com
5	auroraplatform.com	bbonline.ro	benjaminclementine.com
6	bcemsvt.org	billings365.com	blog.davidstea.com
7	binaryvibes.org	binaryvibes.org	brevini.com
8	brentleemusic.com	chemis.org	buy.thegenerationofz.com
9	campinglasiesta.com	creativepeak.org	cavetubing.bz
10	campnewmoon.ca	das-deutsche-institut.de	ccv.edu
11	chainedelespoir.org	denimrefinery.com	cpansearch.perl.org
12	chunshuitang.com.tw	dnavision.com	css-tricks.com
13	colintoh.com	dpress.hu	demo.workflower.fi
14	colintoh.coms	ewcd.org	docutrax.com
15	corraldelamoreria.com	exclusivo.com.br	euvox.eu
16	den-kudryavtsev.com	files.kennison.name	folioapartments.com
17	eatlocal.org	forgetrac.com	glocalnumber.com
18	enyojs.com	fossamusic.com	goiena.eus
19	eseclog.de	fpp.net	greatjewishmusic.com
20	first-last-always.com	giftcompany.de	greenmagichomes.com
21	florahanitijo.com	glennkessler.com	grupaproducts.com
22	gibraltarcompany.ca	globalsoftservices.com	happy-shala.com
23	greatlakeshybrids.com	gschristian.org	hotel-berlin.de
24	grind2energy.com	hellohanqi.com	jayscatering.com
25	hotel-knoblauch.de	hopeww.org	jintsume.com
26	hotelhorizontal-dogo.com	josf.se	keysurgical.com
27	infinit.io	lab.nicholasfrota.com	lab.nicholasfronta.com
28	intertelecom.ua	lastfrontierheli.com	locomotion.fi
29	jutta-hof.de	lawntonac.com.au	luminus.org.uk
30	karlen-stavegren.se	loftusrecreationcentre.com.au	mce.ie
31	kko.com	macnet.com.mx	meltingelements.com
32	med.uio.no	maistrali-apartments.gr	mustelgroup.com
33	mjrcg.com	maristane.com	oceanarraysystems.com
34	naeaapp.com	maxslob.website	pantonism.com
35	neofuturists.org	mf.contropa.com	pizzahut.com
36	organowood.com	minds-africa.org	planeducationnb.ca
37	paulsprangers.com	openkh.rubyforge.org	purplezack.com
38	proteytemen.com	piekutowkis.org	sarahcapecci.com
39	qgis.org	polonel.com	seeability.org
40	reelworld.com	projets.nicodeur.fr	senseofitaly.com
41	ruzgarguluprogrami.com	radiomillenium.ru	spyproof.org
42	schwimmschule-spawala.de	rpg-x.net	thedegenfoundation.org
43	shamsen-assistans.se	sologic.com	tilde.club
44	sibven.ru	storageforyourlife.com	ubuntu.com
45	spartanapp.com	termalliget.hu	udel.edu
46	summit.webrazzi.com	tigu.hk.tlu.ee	vbarbershop.com
47	theiotrevolution.com	timberland-securities.com	vidoons.com
48	tinsnailwines.com	wallawallaclothing.com	vieinternational.com
49	tomsolo.com	waltercatter.com	web-rev.com
50	trv.ee	westantenna.sakura.ne.jp	whogetsmyvoteuk.com



(a) Size (KB) (b) #Style rules / file (c) #Declarations / file

Figure 24: Characteristics of the analyzed preprocessor files

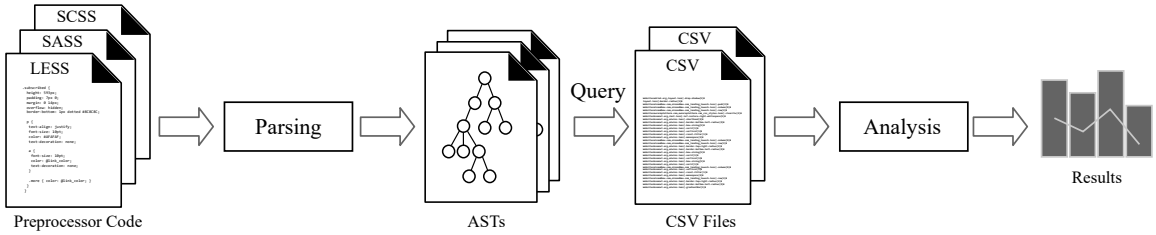


Figure 25: The workflow applied for the collection and analysis of the experimental data

First, we parsed each preprocessor file to obtain its Abstract Syntax Tree (AST). For parsing, we used the corresponding compilers for Less and Sass/SCSS. The Less compiler is originally written in JavaScript, but we used a Java implementation of this compiler, called Less4j[5]. For Sass/SCSS, we used the original compiler written in Ruby. In both cases, we developed additional code for querying the ASTs. The results of the queries were exported to CSV files for further statistical analysis. In Table 5, we provide an overview of the collected data in the subject systems. We will refer to this table in Section 5.4 and discuss the numbers in more detail.

For each examined preprocessor feature we create a separate CSV file. Every CSV file contains the website name, the preprocessor file name, and the line number in which a particular AST element (e.g., variable declaration, *mixin* declaration, *mixin* call) was found. According to the specific characteristics of the AST element type, we include the following additional information in the corresponding CSV file:

1. For variable declarations, we include

[5] `https://github.com/SomMeri/less4j`

Table 5: Overview of the collected data

	Less	Sass	SCSS
# Websites	50	50	50
# Files	255	427	584
Average number of style rules / file	57	52	40
Average number of defined variables / file	16	14	16
Average number of *nesting* usages / file†	43	44	35
Average number of *mixin* calls / file	11	6	12
Average number of *mixin* declarations / file	4.7	4.7	3.7
Average number of *extend* construct usages / file	0	5.2	5
Average number of calls to parameterless *mixins* / file	8	4	6

† Includes all style rules nested under another style rule, or had at least one style rule nested under them.

   (a) the scope of the variable (global or local scope)

   (b) the type of the value stored in the variable. This type can take one of these possible values: color, number, identifier, string, function call, and "other" for all other types of values, as discussed in Section 5.4.1

2. For *mixin* calls, we include

   (a) the name of the called *mixin*

   (b) the total number of arguments passed to the *mixin*

3. For *mixin* declarations, we include

   (a) the name of the *mixin*

   (b) the number of times the *mixin* is called

   (c) the number of its parameters

   (d) the number of declarations which directly or indirectly (i.e., using *nesting*) exist inside the body of the *mixin*

   (e) the number of declarations in the body of the *mixin* which use at least one of the parameters of the *mixin*

   (f) the number of declarations styling vendor-specific properties (e.g., `-webkit-column-gap` for Chrome and Safari, `-moz-column-gap` for Firefox)

(g) the number of distinct parameters which are used for two or more different property types (e.g., a parameter used for styling the `top` and `margin` properties)

(h) the number of style declarations using only hard-coded (i.e., literal) values

(i) the number of vendor-specific style declarations which share at least one of the *mixin*'s parameters

4. For each *nested style rule*, we include

   (a) the selector (as string) of the nested style rule,

   (b) the number of *base* selectors the style rule's selector consists of (e.g., the *grouping* selector `H1, A > B` consists of two *base* selectors, namely `H1` and `A > B`)

   (c) the number of *combinators* in the list of the nested style rule's *base* selectors (note that, the presence of a *combinator* indicates a missed *nesting* opportunity)

   (d) the nested style rule's parent selector (as string)

5. For each use of the *extend* construct, we include the selector (as string) of the target style rule which is extended.

In order to count the number of times a *mixin* is called, we analyze the CSV file containing the *mixin* calls in order to extract the number of calls having the same name with that of the *mixin* declaration. If multiple *mixin* declarations have the same name (i.e., *mixins* with an identical name declared in different preprocessor files), then we count only the number of calls having the same name and belonging to the same file with that of the *mixin* declaration.

5.4 Empirical Study

In this study, we investigate the use of the following preprocessor features: *variables*, *nesting*, *mixin calls* and *extend constructs*. Targeting the goals mentioned in Section 5.1 (developing better migration and refactoring recommendation systems and giving feedback to preprocessor language designers), we attempt to answer the following research questions:

RQ1 ***How do developers use variables in preprocessors?***

We aim at investigating whether developers have a particular preference to global or local scope variables, and the types of style values stored in the variables.

RQ2 ***Do developer use nesting whenever possible?***

We are going to investigate whether developers use *nesting* in every possible situation, or only when the benefits to maintainability are stronger (e.g., in deep hierarchies of elements).

RQ3 ***How and why do developers use mixins?***

For *mixins*, several dimensions will be investigated, namely:

a) Are *mixins* created to be reused in a style sheet?

b) Do *mixins* tend to have a large number of parameters?

c) Are *mixin* parameters reused in multiple style properties?

d) What is the nature of declarations inside the body of *mixins*? For instance, do developers use *mixins* for grouping a set of *related* declarations (e.g., declarations which style the same property for different web browsers)?

RQ4 ***Do developers use the extend construct whenever possible?***

Given the fact that an *extend* construct can be used in place of a parameterless *mixin* (because they are both used to remove duplication of declarations), we are going to investigate whether developers have a preference to use parameterless *mixins* over the *extend* construct or vice versa.

In the following subsections, we answer the abovementioned research questions.

5.4.1 Variables

We investigate whether developers declare variables in the global scope (i.e., for the entire style sheet), or they mostly prefer local variables (e.g., inside a *mixin* or a style rule). Gaining such knowledge can be beneficial in devising migration or refactoring techniques, because, as mentioned before, variables can be used to store one or more style values repeated across different style rule, and thus facilitate the maintainability of the code. Therefore, a migration (or refactoring) algorithm can detect such value-level duplications in pure CSS (or preprocessor code) and suggest the introduction of appropriate variables. Based on our empirical findings, we can align the refactoring recommendations with the practices which are more commonly applied by the developers, when there are multiple alternative Introduce-Variable refactoring opportunities in the local or global scope.

As shown in Table 6, out of 3,651 total variable declarations in the dataset, there are 3,260 global variables (89.29% of the total variable declarations). On the other hand, only 10.71% of the variable declarations are in the local scope (note that we do not count *mixin* parameters as variable declarations). This clearly shows a preference of the developers to define variables in the global scope.

In addition, we are interested in understanding the types of the values stored in the variables. We categorized all possible value types that are allowed in preprocessors, as shown in Table 7, and counted the instances of the variables belonging to each category.

Table 6: Scope of variables

	Global (%)	Local (%)	Total
Less	956 (95.79)	42 (4.21)	998
Sass	917 (84.67)	166 (15.33)	1,083
SCSS	1,387 (88.34)	183 (11.66)	1,570
Total	3,260 (89.29)	391 (10.71)	3,651

In Figure 26, we have demonstrated the percentage of variable instances in each value category, for each of the analyzed preprocessors. As it can be observed, most of the variable declarations are used for *color* values. This accounts for 45.98% of all variables defined in the three datasets. Values in this category consist of the named and Hexadecimal colors, in addition to color functions, such as `rgb()` and `rgba()`. This observation shows that variables are mostly used for facilitating the modifications to the *theme* of web pages (i.e., same structural layout with different color themes).

As it can be observed in Figure 26, there is a considerable use of expressions for the initialization of preprocessor variables. These expressions are either direct references to previously-defined variables, or mathematical expressions manipulating the values of existing variables (e.g., `@opac2: @opac1 + 0.2`). In this way, the preprocessor developers can easily modify existing themes and layouts.

It should be mentioned that, there was one file in the Sass dataset in which the developer used variables for *all* the declarations defined in the file. This practice is definitely uncommon in developing CSS preprocessor code, and it can negatively affect the results of this study by changing the number of times a certain value type is used. Thus, we excluded this single file from this specific analysis for counting variable types.

> **RQ1 Conclusions**: Developers mostly declare global variables (89.29% of the variable declarations have a global scope), and especially variables storing color values (45.98% of the variable declarations have a color value). Hence, any **migration/refactoring technique** should rank higher the suggestions that introduce variables for identical values across different style rules and *mixins*, leading to the introduction of global variables. Recommendations can also be prioritized based on the types of the involved values, giving higher priority to those involving color values.

Table 7: Categorization of value types

Category	Value Type	Example
Number	Angle	`45deg`
	Integer	`-13`
	Length	`18px`
	Number	`4.01`
	Percentage	`50%`
	Resolution	`72dpi`
	Time	`5ms`
	Number function	`floor()`
Color	Named color	`red`
	Hexadecimal	`#FF00FF`
	Color function	`rgb(50, 0, 0)`
Identifier	User-defined	`nice-animation`
	CSS keyword	`top`
String	Unicode string enclosed in `" "` or `' '`	`"Concordia"`
	String function	`replace()`
Function call	Excluding number, color, string functions	`svg-gradient()`
URL	Resource path, using the `url()` function	`url()`
Expression	Expression involving other variable(s)	`@opac1+0.2`
List	Space-separated list of the above-mentioned types of variables	`solid 1px red`

5.4.2 Nesting

In this subsection, we examine how developers take advantage of *nesting* in preprocessors. Our investigation shows that *nesting* is a construct that is widely used by the developers. In Table 8, we present the collected data for *nesting* usage in the three subject preprocessors.

As it can be observed in Table 8, out of all 49,187 style rules, there were 38,605 style rules which were either already nested or could be potentially nested. Out of this number, there were 30,120 style rules (78.02%), which were actually involved in some nesting hierarchy, i.e., they had at least one style rule nested under them, or were nested under another style rule. On the other hand, in the whole dataset, there were 8,485 style rule which could be nested, but developers did not apply

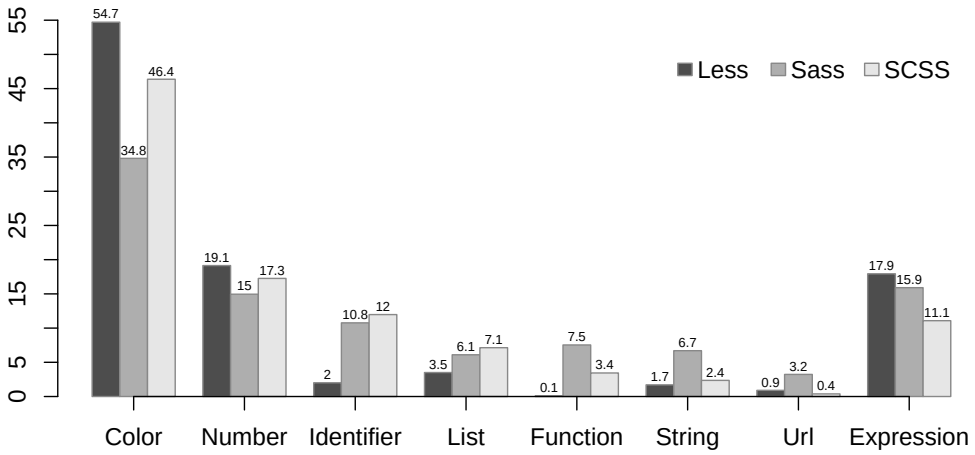


Figure 26: Variable types distribution (numbers represent percentages)

Table 8: Use of *nesting*

	Less	**Sass**	**SCSS**	**Total**
Number of all style rules	12,390	18,555	18,242	49,187
Number of style rules involved in *nesting*	6,481	13,370	10,269	30,120
Number of potential *nesting* opportunities	2,685	1,939	3,861	8,485
Number of all *nestable* style rules	9,166	15,309	14,130	38,605

nesting for them. These selectors of these style rules are basically *combinators*, *pseudo-classes*, and *pseudo-elements* (Section 5.2.2), which can be refactored to take advantage of *nesting*.

To gain more knowledge about *nesting* practices, we also investigated the *nesting depth* in preprocessor files. We define the *nesting* depth of style rule `s`, which is nested under style rule `p`, as the depth of style rule `p` plus one. The depth of a top-level style rule (i.e., a style rule which has no parent in the *nesting* hierarchy) is equal to zero.

Figure 27 demonstrates the box plots along with the violin plots (for exhibiting the distribution of values) for the *nesting* depth of style rules in the examined style sheets. As it can be observed, the median of the *nesting* depth is 2 in all three datasets (for the SCSS dataset, the third quartile is the same as the median, both equal to 2). This means that, in half of the cases, style rules are nested only one or two levels deep, which is a clear indicator that developers prefer to nest style rules even for very shallow *nesting* hierarchies.

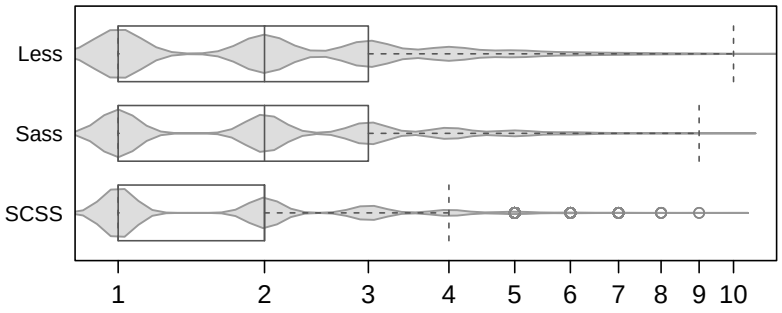


Figure 27: Nesting depth

RQ2 Conclusions: *nesting* is a very popular preprocessor feature that is widely used by the developers (78.02% of the style rules are nested), even in very shallow *nesting* hierarchies consisting of one or two levels. Given this result, any **migration/refactoring technique** should support the recommendation of *nesting* refactoring opportunities, wherever it is possible.

5.4.3 Mixin Calls

We examined the use of preprocessor *mixins*, taking into account four different dimensions.

Number of *mixin* calls

Our goal is to understand whether *mixins* are created to be reused (i.e., called by multiple style rules or other *mixins*), or whether they are created to decompose style rules by extracting a subset of relevant declarations from them (i.e., called by only one style rule). In the former case, *mixins* are used to eliminate duplication of declarations in the CSS code.

For answering this question, first we counted the *mixin* calls for each *mixin* declaration. As shown in Figure 28, the median value for number of times each *mixin* is called is 2 for Less and Sass, and 3 for SCSS. Overall, we found out that 63% of the *mixins* are called more than once.

In addition, we applied the Wilcoxon Signed-Rank Test on the paired samples of the numbers of *mixins* being called just once and of those being called more than once in each website with the following null hypothesis: "the number of *mixins* being called once is larger than the number of *mixins* being called more than once". The null hypothesis was rejected with significance at 95% confidence level (p-value = 0.00003), and thus we can conclude that the *mixins* being called more than once are more than the *mixins* being called only once.

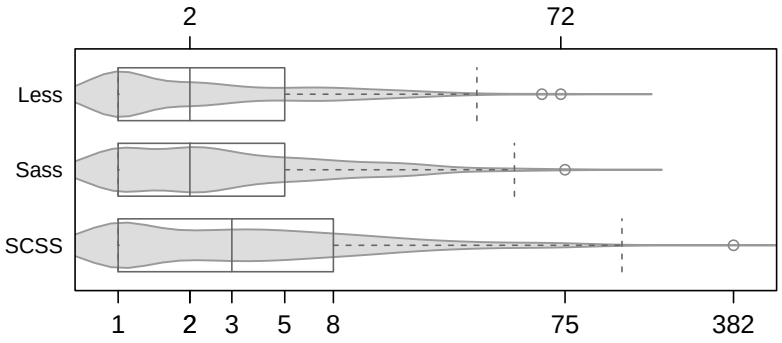


Figure 28: Number of *mixin* calls

There were some interesting cases that we found during the analysis of the results. In the SCSS dataset, there was a *mixin* which was called 382 times. Closer investigation revealed that this case was a *mixin* which was used for generating style rules belonging to different `@media` at-rules (i.e., having different *Media Queries* [Con12]). We discussed that `@media` at-rules provide the possibility of defining alternative styles for different media, e.g., a high-resolution monitor, or the display of a mobile or wearable device. It turned out that the developer called this *mixin* inside the majority of the style rules to avoid the effort needed to rewrite the complete `@media` declaration. On the other hand, in the Sass dataset, there was a website for which the designers used the same animation for several elements in the web pages. Consequently, 75 *mixin* calls referred to a *mixin* which included style declarations for these animations. Finally, the maximum number of calls to a *mixin* in the Less dataset was 72, which occurred for a *mixin* that was used for defining the size of fonts in the target documents. In other words, this *mixin* was called whenever a `font-size` was to be defined. These cases essentially show that *mixins* can be employed for a wide range of purposes when developing style sheets.

Size of *mixins*

We counted the style declarations which were placed directly or indirectly inside each *mixin*, as a measure for *mixins* size. By *indirectly*, we refer to the declarations which belong to the style rules being nested under the examined *mixin*. Here, the goal is to investigate whether developers tend to keep *mixins* short, similar to what is suggested for their counterparts in traditional programming, i.e., functions. As shown in Figure 29, the median of the number of declarations is 3 in all three datasets. Further analysis shows that only 20% of the *mixins* include more than 5 declarations in the whole dataset, suggesting that developers mostly prefer to develop *mixins* having 5 declarations or less.

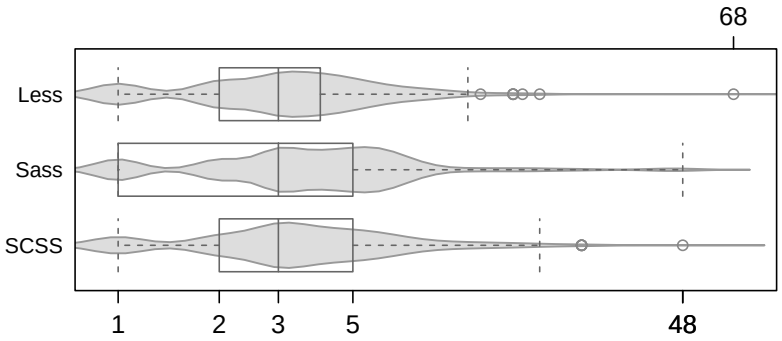


Figure 29: Number of property declarations inside *mixins*

Number of parameters

We are also interested to investigate whether *mixins* tend to have a large number of parameters or not. As it is exhibited in Figure 30, the median value for the number of parameters in *mixin* declarations is equal to one in all datasets. We further found that 68% of the *mixins* have either one or no parameters. The difference in the number of declarations inside *mixins* and the number of *mixin* parameters possibly shows that, in most of the cases, *mixins* either have hard-coded values for the majority of the properties defined inside their body, or their parameters are *reused* in multiple property declarations. We will investigate the reuse of parameters in the next subsection.

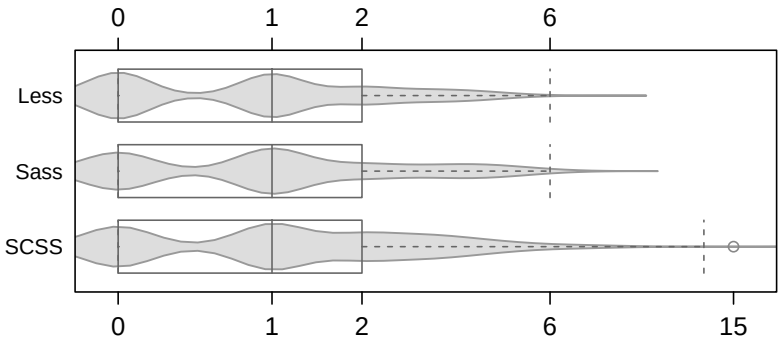


Figure 30: Number of *mixin* parameters

Parameter reuse

We attempted to examine the hypothesis that parameters are reused in multiple declarations. We should first note that style properties in CSS are divided into two categories:

1. Properties which are common across different web browsers;

2. Properties which are specific to one web browser (i.e., vendor-specific properties).

```
.rounded(@radius: 2px) {
  -webkit-border-radius: @radius;
  -moz-border-radius: @radius;
  border-radius: @radius;
}
```

Figure 31: Parameter reuse across vendor-specific properties

As an example, to style font size in different browsers, one will only need to define the `font` property. On the other hand, when styling the `border-radius` property, the developer would need to define a different property for each web browser; for instance, `-webkit-border-radius` for Google Chrome or Safari and `-moz-border-radius` for Mozilla Firefox. Otherwise, the presentation of the target documents will differ across different web browser (Note that, web browsers use these prefixes in the property names when the specifications are not stabilized and the properties are experimental. In such cases, different web browsers might expose varying behavior from the specifications or from other web browsers. Using prefixes is a way to warn the developers who use these properties to thoroughly test the web application on different web browsers before deploying them).

As a result, *mixins* can serve as a solution for grouping vendor-specific properties. In this situation, the same *mixin* parameter would be *reused* across different declarations corresponding to vendor-specific properties. An example of such a case is depicted in Figure 31.

Vendor-specific properties can be easily distinguished by examining whether the property name starts with one of the predefined prefixes by Word Wide Web Consortium (W3C) [Con11]. Our investigation showed that 42% of the *mixins* are used for grouping declarations associated with vendor-specific properties. When a *mixin* has at least one set of vendor-specific properties, on average only 6.6% of the declarations inside that *mixins* are not related to a vendor-specific property. As for parameter reuse, it turned out that 88.81% of the declarations associated with vendor-specific properties shared at least one of the *mixin*'s parameters. This indicates excessive amount of parameter reuse for vendor-specific properties. On the other hand, on average 19% of the *mixins* parameters were reused across properties which did not style the same property in the target documents (for instance, the variable `@w` is used both for `margin` and `padding` properties). This demonstrates that parameter reuse is also taking place for non-vendor-specific properties, although to a much smaller extent.

RQ3 Conclusions: Two thirds of the *mixins* are reused two or more times. Given that, any **migration/refactoring technique** should suggest extracting *mixins* even when there is a small number of style rules sharing the same set of declarations (i.e., to avoid declaration-level duplication). In addition, such a technique should rank higher the suggestions which have small number of parameters (i.e., small number of differences in property values), and include declarations for vendor-specific properties. Moreover, the **preprocessor language designers** should consider creating built-in *mixins* for vendor-specific properties, because a considerable amount of *mixins* (42%) are used for styling this kind of properties.

5.4.4 The "Extend" Construct

Finally, we examine the usage of the *extend* construct. As mentioned in Section 5.2, the *extend* construct is used to eliminate declaration-level duplication, similar to *mixins*. While *mixins* can have parameterized declarations in their body (in contrast to the *extend* construct), a parameterless *mixin* may be thought to have the same use as the *extend* construct. However, one should note that these constructs will result to different CSS code. A use of the *extend* construct will compile to a style declaration with a grouping style rule (as shown in Figure 23), while the code inside a *mixin* will be *duplicated* in the generated CSS code in all the places where the *mixin* is called. In other words, the use of *mixins* introduces duplication in the generated CSS code; consequently, the developer may be tempted to use the *extend* construct over parameterless *mixins*.

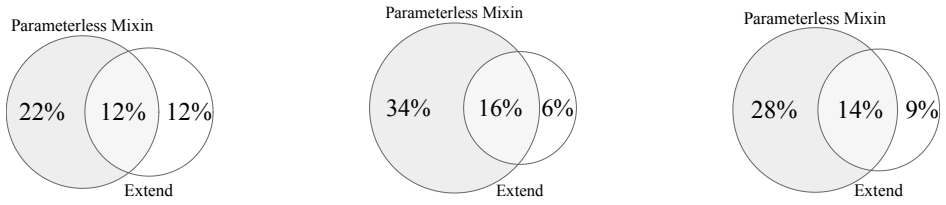
On the other hand, when using the *extend* construct, the preprocessor compiler places the resulting style rule with the grouping selector in the position of the style rule being extended in the generated CSS code (Figure 23). This changes the relative order of the style rules in the style sheet, which may result in changing the presentation semantics of target documents, due to the existing *order dependencies* between the style rules. As a result, developers should take extra caution when using the *extend* construct, and make sure that these order dependencies will not break. This might be a factor that makes developers reluctant to use the *extend* construct.

As shown in Table 5, on average there were around 5 usages of the *extend* construct per file, in the Sass and SCSS datasets (in total 204 and 676 usages, respectively). At the same time, we did not find any use of the *extend* construct in the Less dataset. This could be justified by the fact that the *extend* construct was more recently introduced in the Less preprocessor (version 1.4 released in June 2013), so developers might have not started yet using this feature in a systematic way.

On the other hand, we observed that the average number of calls to parameterless *mixins* in each file is 8, 4 and 6, respectively for Less, Sass and SCSS datasets (Table 5). The higher value for the Less dataset might be explained from the fact that developers did not use the *extend* construct as an alternative solution, because it was not supported by the Less preprocessor until recently.

For the Sass and SCSS datasets where developers used the *extend* construct, we conducted further analysis to understand if there is any preference for using *extend* construct over the parameterless *mixin* or vice versa. Figure 32 displays the Venn diagrams showing the percentage of the websites (out of the total number of websites in the corresponding dataset) which only used one of the constructs or both of them (the overlapping area). As it can be observed, both in Sass (Figure 32a) and SCSS (Figure 32b) datasets, the websites that used only parameterless *mixins* outnumber the ones which used only the *extend* construct.

Figure 32c shows the Venn diagram for both Sass and SCSS datasets combined together including 100 websites in total. We can clearly see that developers have a preference to parameterless *mixins* over *extend*, since in 28% of all websites they exclusively used parameterless *mixins*, while in only 9% of all websites they exclusively used the *extend* construct. Therefore, we may conclude that developers mostly tried to avoid the caveats associated with the *extend* construct, while accepting the duplication in the generated CSS code resulting from the use of parameterless *mixins*. Nevertheless, we showed that it is possible to safely refactor declaration-level duplications in the generated CSS code. In a similar way, parameterless *mixins* could be replaced with the *extend* construct.



(a) Sass (b) SCSS (c) Combined

Figure 32: Percentage of websites using *extend* or parameterless *mixins*

RQ4 Conclusions: Developers tend to prefer using parameterless *mixins* over the *extend* construct, possibly because *mixins* do not affect the presentation semantics of the target documents. As a result, any **migration/refactoring technique** should give higher priority to opportunities introducing parameterless *mixins*, especially when the alternative solution using the *extend* construct cannot guarantee that the presentation of the target documents will be preserved. The **preprocessor compilers** can be enhanced to warn developers about potential styling bugs caused by the incautious use of the *extend* construct.

5.5 Threats to Validity

For minimizing the threats to the *external* validity of this study, we selected two CSS preprocessors which are known to be the most widely used by web developers [Coy12], namely Less and Sass. Additionally, we used the two dialects that Sass preprocessor supports (Sass and SCSS). Moreover, to make the results of the study as generalizable as possible, we examined 150 websites from a wide range of application domains.

To avoid *selection bias*, we included in the list of subjects the top-50 websites for each preprocessor language/dialect, as returned by the Google search engine. As a result, we were not involved in any kind of selection process.

To support the reliability of the study, we have made available the artifacts, which are necessary for replicating the experiment. These include the preprocessor files that we collected, the code we implemented for parsing Less and Sass/SCSS files and querying their ASTs, the CSV files resulting from querying the ASTs, and the R scripts that we developed for the statistical analysis [MT16c].

5.6 Chapter Summary

In this study, we examined the preprocessor codebase of 150 websites to investigate the usage patterns of four language features, namely variables, *nesting*, *mixins* and *extend* constructs. We found out that developers frequently use all these features whenever possible, and gained some valuable knowledge which certainly can help us in devising migration/refactoring techniques and providing feedback to the preprocessor language designers. In summary the take-home messages of the study are:

1. Developers have a clear preference for global variables (89.28% of the variable declarations

have a global scope), and especially variables storing color values (45.98% of the variable declarations have a color value).

2. Developers widely use the *nesting* feature (78% of the style rules are nested), even in very shallow *nesting* hierarchies consisting of one or two levels.

3. Developers tend to reuse *mixins* (63% of the *mixins* are called two or more times). They also tend to create *mixins* with a small number of parameters (68% of the *mixins* have either one or no parameters), and a relatively small size (80% of the *mixins* include 5 or less declarations). Finally, 42% of the *mixins* are used for styling vendor-specific properties.

4. While both parameterless *mixins* and the *extend* construct can be used to eliminate declaration-level duplication in the preprocessor code, developers tend to prefer using parameterless *mixins* to avoid the caveats associated with the *extend* construct.

The gained knowledge in this chapter is valuable for future research, however, we acknowledge the need for a qualitative user study with real-world developers, for triangulating the results of our quantitative study. Unfortunately, as the websites which have been investigated in this study were collected using a web search engine (Google), we did not have access to the developers of the analyzed preprocessor files.

As mentioned before, the lessons learned in this study provide us insights for devising techniques to automatically migrate existing CSS code to preprocessor code. In the next chapter, we propose a technique that detects declaration-level duplication and extracts *mixins* to eliminate them.

Chapter 6

Migrating CSS to Preprocessors by Introducing *mixins*

6.1 Introduction

Despite the gradual adoption of CSS preprocessors in the web development community, there is still a large portion of front-end developers and web designers using solely "vanilla" CSS. As mentioned before, an online poll with nearly 13,000 responses from web developers [Coy12] revealed that 54% of them are using CSS preprocessors. However, the remaining 46% develop only in "vanilla" CSS, probably because they are not aware of preprocessors and the maintainability improvements that can be gained, or because they are dealing with legacy CSS that is not easy to be migrated to CSS preprocessors. Therefore, there is a large community of web developers that could potentially benefit from tools helping them in automatically migrating "vanilla" CSS code to a preprocessor of their preference. Indeed, there are several stories about such migrations in the industry [Ble15, Na15].

A representative migration story happened in 2014 at *Etsy*[1]. Etsy had more than 400,000 lines of legacy CSS code spread over 2000 files developed over the course of 10 years. The developers at Etsy created an in-house tool for migrating the entire CSS code base to Sass automatically [Na15]: as every CSS file is also a Sass file (since Sass is a superset language for CSS), the migration tool renamed all `*.css` files to `*.sass`, and only took care of *syntactical errors* existing in CSS files. This is because Sass compiler catches all the mistakes in the code, unlike web browsers which are lenient on the mistakes. For example, having `background_color` instead of `background-color` in the CSS code is ignored in web browsers and does not stop the CSS code from being interpreted, while the Sass compiler would normally throw an error if it encounters such mistake.

[1] A widely popular peer-to-peer e-commerce website (`https://www.etsy.com`)

Etsy's migration tool essentially missed the opportunities of using many of the Sass features (e.g., *nesting*, *variables*, and *mixins*). For example, developers needed to manually introduce *mixins* in the code base, and automatic CSS to preprocessor migration tools could effectively save this manual effort.

This is the first work to investigate the automatic extraction of duplicated style declarations in CSS into *mixins* in CSS preprocessors, enabling the *reuse* of existing CSS code. Using *mixins* can also improve the readability of Style Sheets by assigning descriptive names to them. According to the results of the previous chapter, developers introduce *mixins* mostly for reusing code in Style Sheets (63% of the *mixins* were called more than once in the code base of each project), but also for breaking long style rules in smaller code fragments, or simply improving code readability. The proposed approach for abstracting duplicated style declarations to *mixins* is one of the fundamental requirements for developing a full-fledged recommendation system that can help developers to migrate existing CSS code to preprocessors.

As a matter of fact, in this study we found several cases where professional developers that *already use* CSS preprocessors also under-utilize them. In other words, there are several *missed* opportunities in existing preprocessor code bases. As we will see, the techniques proposed in this chapter can also be useful for helping developers to utilize *mixins* more effectively.

This work makes the following contributions:

- We propose a method for detecting opportunities to automatically extract *mixins* from existing CSS code. The approach is preprocessor-agnostic, i.e., it is applicable for all CSS preprocessors supporting the notion of *mixins*;
- We propose a method for assuring that the *presentation semantics* of the CSS code are preserved after migration;
- We conduct an empirical study with real websites and Style Sheet libraries using preprocessors to verify the correctness and effectiveness of our approach.

Note: Earlier version of the work done in this chapter has been published in the proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016) [MT16b].

6.2 Abstraction Mechanisms in CSS Preprocessors

Remember that the two main features of CSS preprocessor languages that can be used for eliminating duplicated style declarations in Style Sheets (additional to using style rules with class and grouping selectors in pure CSS) are the *extend* constructs and *mixins*.

Extend enables the reuse of style declarations across style rules. The *extending* style rule inherits all style declarations of the *extended* style rule, and can optionally *override* some of the inherited style declarations in order to change their style values, similar to inheritance in the object-oriented paradigm.

Mixin is a function-like construct containing a set of style declarations, optionally with parameterized style values. A *mixin* is usually called inside the body of a style rule or another *mixin* by passing style values as arguments for its parameters. A *mixin* parameter may have a default value, allowing to omit the corresponding argument.

In the case of exactly duplicated style declarations across different style rules, one may use either the *extend* construct or a parameterless *mixin* for eliminating duplication. However, we showed that developers prefer to use parameterless *mixins* over the *extend* construct (28% of the 100 analyzed websites exclusively used parameterless *mixins*, while only 9% used the *extend* construct), probably due to the *styling bugs* that may be caused by the incautious use of the *extend* construct. Additionally, *mixins* provide a more powerful reuse mechanism by allowing to parameterize the values of style properties. Therefore, in this work, we propose an approach for detecting *mixin* opportunities in CSS code that can help developers to safely migrate to a preprocessor of their preference.

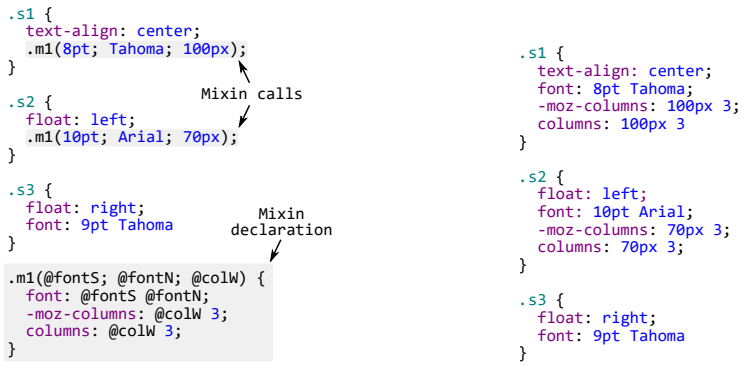
6.3 Automatic Extraction of Mixins

Our approach for detecting *mixin* migration opportunities is based on eliminating duplication at the level of style declarations, and consists of four main steps, which are explained in the following subsections.

6.3.1 Grouping Declarations for Extraction

The first step of our approach is to find sets of style rules sharing one or more style declarations styling the same properties. The tuple $\langle S, P \rangle$, where S is a set of style rules sharing a set of style properties P is considered as a *mixin migration opportunity*.

Consider, for example, the preprocessor code snippet in Less syntax shown in Figure 33a, that contains three style rules, namely `.s1`, `.s2` and `.s3`, and a *mixin* declaration `.m1`, which is called in style rules `.s1` and `.s2`. When the piece of preprocessor code shown in Figure 33a is transpiled to CSS, the code shown in Figure 33b is generated. As it can be observed, the *mixin* calls are replaced with the style declarations of the called *mixin*, and the parameterized values are replaced with the arguments passed in the corresponding *mixin* call.



(a) Preprocessor Code

(b) Generated CSS

Figure 33: Mixin example

Now, suppose that we want to reverse-engineer the CSS code of Figure 33b to the preprocessor code of Figure 33a. We would need to group the declarations corresponding to `font`, `columns` and `-moz-columns` properties from style rules `.s1` and `.s2` and extract these declarations into *mixin* `.m1` after parameterizing the property values being different. In this case, $S_1 = \{\texttt{.s1}, \texttt{.s2}\}$ and $P_1 = \{\texttt{font}, \texttt{columns}, \texttt{-moz-columns}\}$. There are, however, other *mixin* migration opportunities in the code fragment of Figure 33b; e.g., $S_2 = \{\texttt{.s1}, \texttt{.s2}, \texttt{.s3}\}$ and $P_2 = \{\texttt{font}\}$.

It should be emphasized that for a given property $p \in P$ the grouped style declarations should have the same *importance*. Recall that using the `!important` rule makes a declaration to have precedence over other declarations assigning a value to the same property. As a result, the importance of the involved declarations should be kept intact when forming a *mixin*, to make sure that the presentation of target documents will be preserved after migration. Consequently, we avoid the grouping of declarations having a different importance (i.e., either all or none of the grouped declarations may use the `!important` rule).

A naive approach for finding the repeated properties would be to exhaustively generate all possible combinations of style properties and then examine if they appear together in two or more style rules. However, this would certainly lead to a combinatorial explosion, due to the large percentage of duplication in CSS files. We saw that, on average, 66% of the style declarations in Style Sheets are duplicated at least once (i.e., they share the same property and value with another style declaration). For detecting *mixin* migration opportunities, the constraint that property values should be identical or equivalent is not necessary, since *mixins* allow the parameterization of differences in the property values. This results to an even larger percentage of declarations that can be potentially grouped to form a *mixin*, making the exhaustive generation inapplicable.

Similar to the approach taken in Chapter 4, we managed to overcome this issue by treating the initial problem as a *frequent itemset mining* problem. Again, we treat a CSS file as a transactional

database, and each style rule as a transaction. In contrast to the approach used in Chapter 4, however, we treat each *property* corresponding to a style declaration as an item. In other words, we use a more relaxed version of the algorithm that does not require the style declarations that form an item to be equivalent.

We set the minimum support count of the algorithm to two again (i.e., the smallest possible value). We selected this value because we have empirically shown that the median number of times a *mixin* is called in real-world CSS preprocessor code is two (Chapter 5).

Once again, the FP-Growth algorithm [HPY00] was adopted, because it is considered efficient and scalable. Applying the FP-Growth algorithm to the CSS code of Figure 33b will result in the output shown in Table 9.

Table 9: Frequent itemsets of style properties

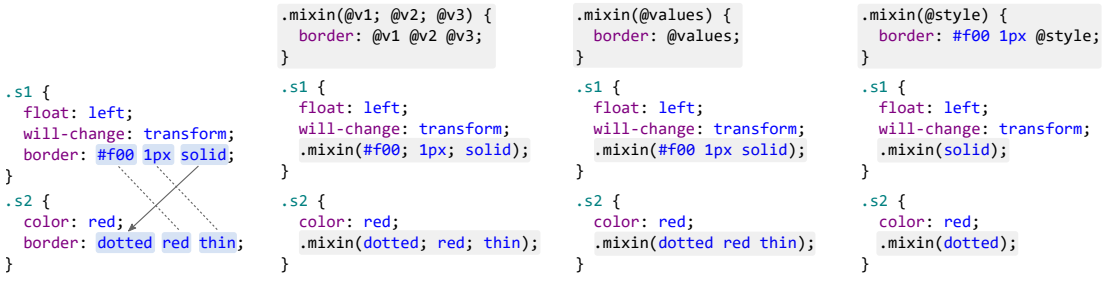
$\|P\|$	S	P
1	{.s1, .s2}	{-moz-columns}
	{.s1, .s2}	{columns}
	{.s2, .s3}	{float}
	{.s1, .s2, .s3}	{font}
2	{.s1, .s2}	{columns, -moz-columns}
	{.s1, .s2}	{columns, font}
	{.s1, .s2}	{-moz-columns, font}
	{.s2, .s3}	{font, float}
3	{.s1, .s2}	{font, columns, -moz-columns}

Each row (itemset) in Table 9 constitutes a separate *mixin* migration opportunity. The *mixin* `.m1` in Figure 33a corresponds to the last itemset of the table, and it *subsumes* the first three itemsets with $|P| = 2$, which in turn *subsume* the two first itemsets with $|P| = 1$.

6.3.2 Detecting Differences in Style Values

For a given *mixin* migration opportunity $\langle S, P \rangle$, we need to check for every property $p \in P$, if the corresponding style declarations have different (i.e., non-equivalent) values. For each difference in the property values, a parameter should be introduced in the resulting *mixin*. However, the parameterization of differences can be achieved in several alternative ways.

As an example, consider the CSS code shown in Figure 34a that contains two style rules for selectors `.s1` and `.s2`, respectively. Both style rules style property `border`, which is a well-known



(a) Original CSS code (b) Fine-grained parameters (c) Coarse-grained parameters (d) Minimal parameters

Figure 34: Alternative ways for extracting a *mixin*

shorthand property allowing to set the values of several other CSS properties simultaneously, such as `border-width`, `border-style`, and `border-color`.

Note that, some CSS style properties do not force a specific order for the values of the properties they group. As a result, in the style rule `.s1` the `border` style declaration first assigns the `border-color` with `#f00`, then the `border-width` with `1px`, and finally the `border-style` with `solid`, while in style rule `.s2` the order is different starting first with `border-style` followed by `border-color` and `border-width`.

In this particular example, and in all the cases involving shorthand properties (and other non-shorthand properties that accept multiple values, e.g., `background-position`), there are three possible ways to parameterize the differences in property values:

1. Introduce a separate parameter for each pair of different values in the order they appear in the corresponding shorthand property declarations (Figure 34b). This approach has two main limitations. First, it may introduce parameters for properties using different kinds of values. The variables and parameters in CSS preprocessors do not have a type, and thus it is possible to have parameters accepting arguments of different value kinds. However, it will be extremely difficult for a developer to understand and reuse a *mixin* having parameters that can take values for semantically different properties (e.g., a property that should be styled with color values, and a property that should be styled with dimension values). Second, such an approach may lead to *mixins* with an unnecessarily long parameter list, which is considered a code smell [FBBO99], since it makes more difficult to call and reuse such *mixins*. We also showed that developers mostly tend to create *mixins* with zero or one parameter (68% of the *mixins* have either one or no parameters).

2. Introduce a single multi-value parameter for all individual properties grouped by the corresponding shorthand property (Figure 34c). Although this approach gives more flexibility to the developer when calling the *mixin*, it is also more error-prone, because the developer needs to

know very well the CSS documentation regarding the individual style values that are mandatory and those that can be omitted (i.e., *optional* values), or the order of the individual style values (in case of style properties where the order is important). Inexperienced CSS developers would need to spend time studying the documentation in order to properly call a *mixin* with such parameters.

3. Introduce a parameter for each pair of *matching individual properties* having non-equivalent values in the corresponding shorthand property declarations (Figure 34d). In the example shown in Figure 34a, the matching individual properties between the two style rules are represented with arrows. The dashed-line arrows indicate properties with equivalent values (e.g., the *named color* `red` and the *hexadecimal color* `#f00` are not lexically identical, but are alternative representations for the same color). The solid-line arrows indicate properties with non-equivalent values that should be parameterized. This approach has two main advantages over the other approaches. First, it introduces a minimal number of parameters compared to the first approach, when some individual properties are styled with identical or equivalent values (regardless of the order they appear in the shorthand property declarations). Second, it allows to introduce parameters with more *semantically expressive* names compared to the second approach, since the names of the matching individual properties can be used as parameter names (e.g., `@style` parameter in Figure 34d corresponding to the individual property `border-style`).

Inferring Individual Style Properties

In our approach, we adopted the last parameterization strategy discussed in the previous section due to its advantages over the other two strategies. To implement this strategy, we first need to infer the *individual style property* (ISP) corresponding to each style value that appears within the style declarations for the set of properties P declared in the set of style rules S. An ISP represents the *role* of a style value in a style declaration, and corresponds to the actual individual style property this value is being assigned to. Table 10 shows the ISPs that are assigned to the values of the `border` style declarations in the example of Figure 34a. For instance, the pair of values corresponding to colors (`#f00` and `red`) are both assigned to the same ISP, which is the individual property `border-color`.

For the style properties that can accept only a single value (e.g., `color`, `float`), the ISP assigned to their values is the same as the style property name. For shorthand properties (e.g., `border`, `background`, `columns`), we refer to the CSS specifications [Wor17] for assigning an ISP to each one of their values. In our current implementation, we have coded ISPs for 42 multi-valued and shorthand CSS properties, which account for all major CSS properties used in Style Sheets. The list of the currently-supported style properties is given in Table 11. When comparing two declarations

Table 10: Individual Style Properties (ISPs)

Declaration	Style Value	ISP
border: #f00 1px solid	#f00	"border-color"
	1px	"border-width"
	solid	"border-style"
border: dotted red thin	dotted	"border-style"
	red	"border-color"
	thin	"border-width"

for parameterizing the differences in their values, we compare each pair of values corresponding to the same ISP. We follow the same approach explained in Chapter 4 for examining whether two values are equivalent.

Optional (omitted) values

We mentioned that some properties can have *optional* values in CSS. For instance, the property `font` can accept 7 style values, while developers may omit 5 of them. In the case of omitted values, web browsers follow the CSS specifications to compute them. In some cases, they assign *initial* (i.e., default) values to the omitted values. In other cases, the omitted value is calculated based on another explicitly given value. Following the same approach, for every omitted value we actually compute a *virtual value*, and also assign the appropriate ISP to it. This allows parameterizing declarations having an *unequal* number of style values, e.g., `font: bold 10pt Tahoma` and `font: 18pt Arial`. In this example, the first declaration is styling the `font-weight` ISP with the explicitly-defined value `bold`, while the second one styles the same ISP with the default value `normal`.

Shorthand vs. individual properties

Another possible scenario is having some style rules taking advantage of shorthand properties, while other style rules are instead declaring separately individual properties (this is similar to having type III CSS duplication, as defined in Chapter 4, with the exception that style values are not considered in the comparison).

Consider the CSS example shown in Figure 35a. Style rule `.s1` contains four separate style declarations for the individual properties `margin-top`, `margin-right`, `margin-bottom`, and `margin-left`, while style rule `.s2` contains a single style declaration for the shorthand property `margin`. Note that, the last value in the `margin` declaration is omitted (i.e., there are three values instead of four). This

Table 11: List of the supported style properties

Non-shorthand properties	**Shorthand properties**
background-position	border-radius
background-size	margin
background-clip	padding
background-origin	border-width
background-image	border-style
background-repeat	border-color
background-attachment	border
border-top-left-radius	border-bottom
border-top-right-radius	border-left
border-bottom-right-radius	border-right
border-bottom-left-radius	border-top
transform	outline
transform-origin	column-rule
perspective-origin	columns
border-spacing	list-style
text-shadow	transition
box-shadow	font
font-family	background
content	
transition-property	
transition-duration	
transition-timing-function	
transition-delay	
quotes	

(a) Original CSS

(b) Extracted *mixin*

Figure 35: Mixin for shorthand/individual properties

value corresponds to the `margin-left` ISP, and based on the CSS specifications, it should take the `margin-right` value (i.e., `5px`) when it is omitted. To enable the detection of *mixin* migration opportunities in such cases, we again collapse the sets of individual property declarations that can be grouped into *virtual shorthand declarations*. In this way, it is possible to find *mixin* migration opportunities between style rules either having actual or virtual shorthand declarations, and thus the CSS example shown in Figure 35a can be migrated to use the *mixin* shown in Figure 35b.

6.3.3 Introducing a Mixin in the Style Sheet

In this section, we describe an algorithm that takes as input a *mixin* migration opportunity $MO = \langle S, P \rangle$ to generate a *mixin* declaration and update the style rules in set S to call the new *mixin* passing the appropriate arguments.

In Algorithm 2, we use the following helper functions. Function `getISPsWithNonEquivalentValues` returns the ISPs, which are defined in the style declarations (declared inside the style rules in set `S`) corresponding to property `p` and have non-equivalent values.

Function `generateStyleDeclarationTemplate` generates a style declaration template for property `p` with a placeholder (i.e., an unknown value) for each ISP defined in the style declarations (declared inside the style rules in set `S`) corresponding to property `p`. The template will essentially contain a mapping of ISPs to style values or variables (i.e., *mixin* parameters) after the execution of the algorithm.

Finally, function `getStyleDeclaration` returns the style declaration corresponding to property `p` declared inside style rule `s`, and function `getStyleValue` returns the style value corresponding to an ISP defined in style declaration `d`.

Algorithm 2: Algorithm for introducing a *mixin*

Input : A *mixin* migration opportunity $MO = \langle S, P \rangle$

Output: A *mixin* declaration $M = \langle M_p, M_d \rangle$

A mapping of selectors to lists of *mixin* arguments

```
1  M_p ← ∅ // the ordered set of mixin parameters
2  M_d ← ∅ // the ordered set of mixin style declarations
3  foreach s ∈ S do
4      M_a(s) ← ∅ // the list of mixin arguments for s
5  end
6  foreach p ∈ P do
7      differences ← getISPsWithNonEquivalentValues(p, S)
8      template ← generateStyleDeclarationTemplate(p, S)
9      foreach ISP ∈ template.ISPs do
10         if ISP ∈ differences then
11             param ← newMixinParameter(ISP)
12             M_p ← M_p ∪ param
13             ISP ↦ param // map ISP to mixin parameter
14             foreach s ∈ S do
15                 d ← getStyleDeclaration(p, s)
16                 arg ← getStyleValue(ISP, d)
17                 M_a(s) ← M_a(s) ∪ arg
18             end
19         end
20         else
21             s ← S_0 // get the first style rule in S
22             d ← getStyleDeclaration(p, s)
23             value ← getStyleValue(ISP, d)
24             ISP ↦ value // map ISP to common value
25         end
26     end
27     M_d ← M_d ∪ template
28 end
```

Generating mixin declaration

In order to create the *mixin* declaration, the algorithm generates a style declaration template (line 8) for each property p in the set of style properties P. Function `generateStyleDeclarationTemplate` goes through all declarations styling p in the set of style rules S and finds the union of ISPs that are assigned with values. This approach can guarantee that all affected ISPs will be present in the template, even if some style declarations omit the definition of optional values. Next, for each ISP in the template, the algorithm checks if the assigned style values are equivalent or not. If the values are different, the ISP is mapped to a new *mixin* parameter, which is also added to the parameter list of the *mixin* declaration. Otherwise, the ISP is mapped to the commonly assigned value in all style rules. Finally, the resulting template after the mapping of all ISPs is added to the list of style declarations inside the body of the *mixin* declaration. It should be emphasized that the order of the style declarations inside the *mixin* follows the relative order of the style declarations in the original style rules from which they were extracted. As it will be explained in Section 6.3.4, this is essential for preserving the presentation of the target documents, in the case where some style declarations have order dependencies with each other. In such a case, an ordering that reverses the original order dependencies between the style declarations would affect the values assigned to the ISPs, thus changing the presentation.

Unifying mixin parameters

Developers tend to create small number of parameters for *mixins*, as discovered in our empirical study (Chapter 5). Consequently, exploring ways to even further minimize the number of introduced parameters for *mixins* might help in recommending *mixins* that are probably more acceptable by developers.

Consider, for instance, the real code snippet taken from the W3C.CSS framework[2] illustrated in Figure 36a. Suppose that we would like to extract a *mixin* for the duplicated style declarations in this code. As it is observed, there are 2 sets of *matching* style declarations (i.e., style declarations with the same property names) repeated across the style rules, corresponding to the `padding-top` and `padding-bottom` style properties. In each set, one style value is different across all the style rules, meaning that it requires parameterization for extracting a *mixin*. Consequently, if we create one parameter for each of these sets of matching yet different style values using Algorithm 2, the *mixin* will need two parameters (as shown in Figure 36b). Moreover, in all the call sites of the *mixin* we would also need to pass two arguments.

It is, however, observed in Figure 36a that the style values are consistently repeated across the *non-matching* style declarations. In other words, the values for the `padding-top` and `padding-bottom`

[2]A CSS framework "with built-in responsiveness" (`https://www.w3schools.com/w3css/`)

(a) Original CSS (b) *mixin* without unified parameters (c) *mixin* with unified parameters

Figure 36: Merging *mixin* parameters

style properties are always equal in all the style rules, resulting to an opportunity for *unifying* the introduced *mixin* parameters. As a general rule, whenever the style values of non-matching style declarations participating in an extract *mixin* refactoring are consistently repeated in all the style rules, we unify the corresponding parameters created for the differences between the values of the matching style declarations. Following this rule, the *mixin* extracted from the code shown in Figure 36a will look like what is depicted in Figure 36c.

Naming the mixin and mixin parameters

The names of the *mixin* parameters are generally equal to the individual style properties (i.e., ISPs) assigned to the values for which the parameters are extracted. This helps in understanding the role of the parameter in the *mixin*, resulting to better readability. For example, in Figure 36b, the name of the two parameters are `@padding-top` and `@padding-bottom` (remember that the ISP assigned to the only value of a single-valued style property is equal to its property name, which is the case for `padding-top` and `padding-bottom`).

In case of *unified parameters*, however, this rule cannot work, because the parameters being unified might belong to different ISPs. In such cases, we try to find a common term expressing these different ISPs.

As a convention, all CSS names that need more than one word are separated by hyphens (e.g., `padding-top`, `word-wrap`). Following this convention, all the assigned ISPs in our method are also hyphen-separated, if they are composed of more than one word. We use this convention to find a common name for *mixin* parameters. We run the *longest common subsequence* (i.e., LCS) algorithm

on the words constituting the ISPs corresponding to a unified parameter, treating each word as a single unit in the algorithm.

For example, suppose that we would like to unify two parameters, corresponding to the `border-left-style` and `border-right-style` ISPs. The former ISP is treated as `ABC`, and the latter as `ADC` – note also that we use the same letter for the same word to allow the LCS algorithm find the common words. In this example, the LCS is equal to `AC`, corresponding to the term `border-style`, which is selected as the name of the parameter. In Figure 36c, using this approach has led to selecting `@padding` as the name of the introduced, unified parameter.

In the cases that the LCS algorithm cannot find a common subsequence (which, for instance, happens when unifying two parameters that are not related ISP-wise, like for the `left` and `float` ISPs), the term `@argX` is selected for the parameter, where `X` is an integer, increased for each such parameter in the *mixin*.

Note that, the same approach can be employed for naming the extracted *mixins* as well. Here, we can run the LCS algorithm on the property names of the style declarations defined within the *mixin*, with a similar treatment for the constituting words.

In any case, as we will see in Chapter 7, the developer has full control over the suggested parameter names before applying the refactoring.

Adding mixin calls to style rules

Given the *mixin* migration opportunity $MO = \langle S, P \rangle$, for each one of the style rules in S, a call to the generated *mixin* should be added. Whenever the assigned style values for a given ISP are not equivalent, the algorithm goes through all style rules in S, and for each style rule s appends to the corresponding list of *mixin* arguments $M_a(s)$ the actual value assigned to the ISP by s (lines 14-18). At implementation level, in each style rule s the style declarations corresponding to the set of properties P are removed, and a *mixin* call with the argument list $M_a(s)$ is added.

6.3.4 Preserving Presentation

In refactoring, preserving the behavior of the program is very critical [Opd92]: the refactored program should have exactly the same behavior as the original program before refactoring. In a similar manner, any refactoring or migration operation applied to CSS code should preserve the *presentation* of the target documents (i.e., the style values applied to the DOM elements after refactoring should be exactly the same as before refactoring). Therefore, in the context of CSS, *program behavior* corresponds to *document presentation*, and any CSS refactoring/migration technique should make sure that document presentation is preserved.

As we discussed in detail in Chapter 4, in order to preserve the presentation of the target documents, the order dependencies between the style rules and style declarations declared in a CSS file should be preserved after refactoring.

Within the context of *mixin* migration, the introduction of a new *mixin* does not affect the original order of the style rules. Therefore, it is not possible to break the existing inter-style rule order dependencies by introducing a *mixin*. However, there might exist *intra-style rule* order dependencies between style declarations in the style rules where the *mixin* calls will be added. Consider, for instance, the CSS code shown in Figure 37c. The `border-bottom` declaration in style rule `.a` overrides the style values, which are defined by the `border` declaration. The `border` declaration is a shorthand declaration defining the border for all individual sides of an element (`top`, `right`, `bottom`, `left`). Next, the developer overrides with `none` the border styling for the bottom side of the element. CSS developers tend to override more generic style properties (e.g,. `border`) with more specific ones (e.g., `border-bottom`), because this approach requires less code than defining separately all specific style properties (e.g., `border-top`, `border-right`, `border-bottom`, `border-left`).

```
<html>
  ...
  <body>
    ...
    <span class="a">
      test
    </span>
    ...
  </body>
</html>
```

(a) Sample HTML

```
.a {
  ...
  border-bottom: none;
  ...
  .mixin1();
}
.mixin1() {
  border: solid 3px red;
  padding: 1px;
}
```

(b) Misplaced *mixin* call

```
.a {
  ...
  border: solid 3px red;
  border-bottom: none;
  padding: 1px;
  ...
}
```

(c) Original CSS

```
.a {
  ...
  border-bottom: none;
  ...
  border: solid 3px red;
  padding: 1px;
}
```

(d) Generated CSS

(e) Original presentation

(f) Broken presentation

Figure 37: Intra-style rule order dependencies

Applying style rule `.a` to the sample HTML code shown in Figure 37a will result to a web document rendered as shown in Figure 37e. Let us assume that we extract a *mixin* containing the

`border` declaration, and we place the *mixin* call at the end of `.a`, as shown in Figure 37b. The preprocessor will then generate the CSS code shown in Figure 37d, where the `border` declaration is placed *after* the `border-bottom` declaration. This will invert the original overriding relation between the two declarations, resulting to the undesired presentation shown in Figure 37f (i.e., a styling bug). Therefore, placing the *mixin* call in an incorrect position can actually change the presentation of the target documents.

We define an *intra-style rule order dependency* from style declaration d_i to d_j (both declared in the same style rule) due to individual property *isp*, denoted as $\langle d_i \rangle \xrightarrow{isp} \langle d_j \rangle$, iff:

a) declarations d_i and d_j set a value to individual property *isp* and have the same importance (i.e., both or none of the declarations use the `!important` rule),

b) declaration d_i precedes d_j in the style rule.

To ensure that the presentation of the target documents will be preserved, we define the following *preconditions*:

> *Precondition 1*: The addition of a *mixin* call in a style rule should preserve all order dependencies among the style declarations of the rule.

Similar to the approach that we took for finding the right position of the new style rule when refactoring duplicated code within CSS (Chapter 4), the problem of finding an appropriate position for calling the extracted mixin m inside the body of a style rule can be also expressed as a *Constraint Satisfaction Problem* (CSP) defined as:

Variables The positions of the style declarations involved in order dependencies including m.

Domains The domain for each variable is the set of values $\{1, 2, ..., N - M + 1\}$, where N is the number of style declarations in the original style rule, and M is the number of style declarations extracted from the style rule to m.

Constraints Assuming that m contains style declarations assigning values to the set of individual properties $ISPs$, an order constraint is created in the form of $pos(d_i) < pos(d_j)$ for every order dependency $\langle d_i \rangle \xrightarrow{isp} \langle d_j \rangle$ where $isp \in ISPs$

In the example of Figure 37c, there is one order dependency:

$$\texttt{border: solid 3px red} \xrightarrow{\texttt{border-bottom-style}} \texttt{border-bottom: none}$$

resulting to the following constraint:

$$pos(\texttt{border}) < pos(\texttt{border-bottom})$$

Based on this constraint, the call to `.mixin1` should be placed at any position before the `border-bottom` declaration to preserve the presentation of the target document in Figure 37a. If there are multiple conflicting order dependencies between the *mixin* call and declarations of the style rule, it might be necessary to reorder some style declarations in order to comply with the solution returned by the solver. On the other hand, if the CSP is unsatisfiable (i.e., no solution is found) the corresponding *mixin* migration opportunity is excluded as non-presentation-preserving.

> *Precondition 2*: The ordering of the style declarations inside a *mixin* should preserve their original order dependencies in the style rules from which they are extracted.

This precondition is checked by extracting the original order dependencies between the style declarations inside mixin m from each style rule where m will be called. Assuming that m contains style declarations assigning values to the set of individual properties $ISPs$, if there exist two style rules s_i and s_j, where an order dependency for the same $isp \in ISPs$ is reverse, i.e., $\langle s_i, d_k \rangle \xrightarrow{isp} \langle s_i, d_l \rangle$ vs. $\langle s_j, d_l \rangle \xrightarrow{isp} \langle s_j, d_k \rangle$, then there is an order dependency *conflict* between s_i and s_j, and the corresponding *mixin* migration opportunity is excluded as non-presentation-preserving.

6.4 Evaluation

To assess the correctness and usefulness of the proposed technique, we designed a study aiming to answer the following research questions:

RQ1: Does the proposed technique always detect *mixin* migration opportunities that preserve the presentation of the web documents?

RQ2: Is the proposed technique able to find and extract *mixins* that developers have already introduced in existing CSS preprocessor projects?

6.4.1 Experiment Design

Selection of Subjects

To be able to answer the aforementioned research questions, we need to create a dataset of CSS files, which actually contain opportunities for introducing *mixins* by grouping style declarations duplicated among different style rules.

We relied on the dataset of our previous study (Chapter 5), which was used to investigate the practices of CSS preprocessor developers by analyzing the code base of websites using two different preprocessors, namely Less and Sass. More specifically, out of 50 websites in which style sheets were developed using Less, we selected the preprocessor code base of 21 websites, in which at least one *mixin* declaration was called at least two times, since a *mixin* called more than once in the preprocessor code will result in duplicated style declarations in the generated CSS code. Our approach should be able to reproduce the original *mixins* declared in the preprocessor files, and possibly recommend other *mixin* opportunities that the developers might have missed.

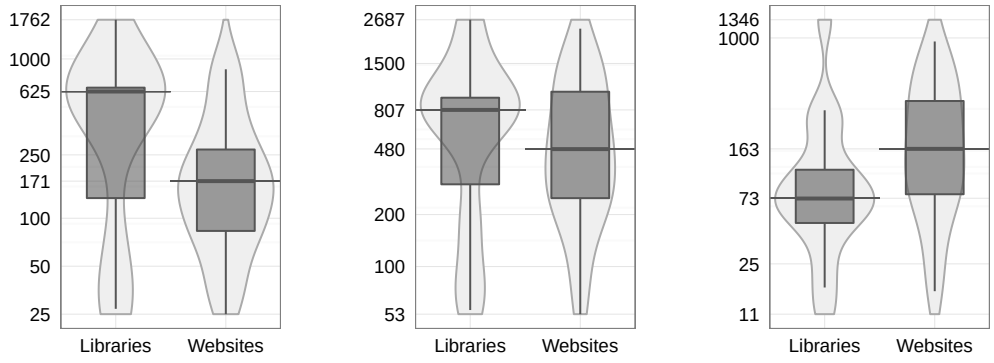
We further extended this dataset with the CSS code generated from the preprocessor source code of eight popular Style Sheet libraries. We expect that the selected libraries apply the best practices regarding *mixin* reuse, since they are developed by very experienced developers. The complete list of the selected websites and libraries, along with the number of Less files and CSS files (resulting from transpiling Less files), the number of developer-defined *mixins*, the total number of style rules and declarations (representing the size of the analyzed CSS files), and the number of migration opportunities detected by our approach for each subject are shown in Table 12. The collected data, in addition to the implemented tools are available on-line [MT16d].

To better demonstrate the size characteristics of the examined CSS files, we show the distribution of the number of style rules and declarations defined in these files in Figure 38a and Figure 38b, respectively. The scale of the box plots and the underlaid violin plots is logarithmic, and the horizontal bars correspond to the median values. The examined libraries tend to have more style rules and declarations than the examined websites. Figure 38c shows the plots for the number of *mixin* migration opportunities detected by our approach per CSS file in the dataset. As it can be observed, the median number of opportunities is 73 and 163, for the libraries and websites, respectively. In order to control for the CSS file size, and perform a fair comparison between the number of *mixin* migration opportunities in libraries and websites, we further normalized the number of opportunities detected in each CSS file by the number of style declarations defined in it. The normalized medians are 0.13 and 0.35 for the libraries and websites, respectively. This result shows that although the CSS code generated by libraries is larger in size than the code generated by the examined websites, the libraries tend to have less duplicated style declarations (and thus less *mixin* migration opportunities) than the examined websites. We can consider this as an indication that the preprocessor code of libraries is better designed.

Table 12: Overview of the collected data

	Name	#Less files	#CSS files	#Actual mixins†	#Style Rules	#Declarations	#Detected Opportunities
Websites	aisandbox.com	3	2	6	113	443	188
	auroraplatform.com	2	1	1	83	247	100
	bcemsvt.org	17	2	2	163	327	70
	brentleemusic.com	13	1	13	861	2344	944
	campnewmoon.ca	2	1	3	218	527	162
	chainedelespoir.org	28	1	8	290	1081	528
	chunshuitang.com.tw	1	1	1	176	511	165
	colintoh.com	9	2	4	59	174	48
	first-last-always.com	16	1	6	339	1116	638
	florahanitijo.com	4	1	11	189	822	273
	greatlakeshybrids.com	1	1	3	104	373	168
	hotel-knoblauch.de	1	1	2	199	446	119
	intertelecom.ua	1	1	6	393	1329	708
	jutta-hof.de	2	1	3	171	560	245
	kko.com	1	1	1	98	255	84
	med.uio.no	80	1	6	762	1622	436
	naeaapp.com	14	1	8	828	1507	382
	neofuturists.org	12	3	11	522	1397	593
	paulsprangers.com	5	1	3	125	337	64
	schwimmschule-spawala.de	2	1	8	171	560	537
	summit.webrazzi.com	1	1	1	84	261	96
Libraries	base	20	1	2	489	736	114
	essence	119	7	11	4571	5939	615
	flatui	60	1	15	1139	2631	1346
	formstone	36	5	4	145	387	120
	kube	16	1	6	374	807	206
	schema	22	1	12	536	1524	284
	skeleton	1	1	2	95	222	45
	turret	83	1	34	1762	2687	307
Total		**572**	**44**	**193**	**15059**	**27811**	**9585**

† includes only the *mixins*, which are called at least two times



(a) #Style Rules (b) #Declarations (c) #Opportunities

Figure 38: Characteristics of the analyzed CSS files

6.4.2 Results

RQ1: Testing Presentation Preservation

Motivation: The goal of RQ1 is to ensure that our technique would never recommend the introduction of a *mixin* that could change the presentation of the target web documents (i.e., cause a styling bug).

Method: Let us assume that we have two CSS files C and C' (the latter resulting from transpiling the preprocessor code after the introduction of a new *mixin* in C) and we apply them on the same target documents. The target documents will have the same presentation iff:

1. There is a one-to-one correspondence (i.e., mapping) between the style rules defined in C and C'.
2. The relative order of the mapped style rules is the same in both C and C' (i.e., cascading is preserved).
3. Each pair of mapped style rules from C and C':
   (a) selects the same DOM elements in the target documents (i.e., specificity is preserved).
   (b) defines exactly the same set of individual style properties.
   (c) assigns equal or equivalent style values to each defined individual property.

Conditions 1, 2, and 3a are met by the nature of the transformation, because the introduction of a *mixin* does not add or remove style rules (i.e., one-to-one correspondence is met), does not change the order of the style rules (i.e., cascading is preserved), and does change the selectors specified in the style rules (i.e., specificity is preserved), respectively.

The only conditions that could be violated from an erroneous introduction of a *mixin* are 3b and 3c. Condition 3b could be violated, if the introduced *mixin* contains declarations for more, less, or different individual style properties than those that were removed/extracted from the style rule calling the *mixin*. Condition 3c could be violated if the parameterization of the differences in the style values is not correct, or if the *mixin* call is not placed in the appropriate position inside a style rule, or if the style declarations are not ordered correctly inside the *mixin* (Section 6.3.4).

Therefore, we developed a method to test conditions 3b and 3c that takes as input a CSS file C and a *mixin* migration opportunity MO, applies MO on file C to generate the corresponding CSS preprocessor code CP, then transpiles CP to obtain CSS file C', and examines the assertion:

> For every pair of matching style rules (s, s') defined in C and C', respectively, `style-map`$(s) \equiv$ `style-map`(s').

Two style rules are considered as matching if they have an identical selector. Function `style-map` takes as input a style rule and extracts a map in which the keys are the individual style properties (ISPs) defined in the style rule, and each key is mapped to the *final* style value assigned to the corresponding ISP, after all possible overrides. Two style maps are *equivalent* ($\equiv$) if their key sets are equal, and the style values corresponding to each key are equal or equivalent. Two style values are considered *equivalent*, when they are lexically different, but constitute alternative representations for the same style value (e.g., `red` $\equiv$ `#F00`).

Results: In total, we detected and applied 9,585 *mixin* migration opportunities and automatically tested them using the aforementioned method. It should be emphasized that a large portion of these opportunities overlap with each other (i.e., they affect common style rules and declarations), and thus it is not possible to apply them sequentially, since the application of an opportunity will make infeasible the opportunities it overlaps with. Therefore, we applied each one of them separately on the original CSS files.

We observed several cases where our testing method found styling bugs, which were due to our faulty implementation of style value inferencing. As an example, we found cases in which failing to assign correct ISPs to style values led to their incorrect parameterization and, consequently, the resulting preprocessor code was transpiled to a CSS file with different styling semantics than the original CSS file. For instance, when a shorthand property is assigned with the value `none`, only one of the ISPs is actually assigned with `none`, while the remaining ISPs are assigned with *default* values. Our implementation was not inferring correctly the default values, and this caused problems in preserving the presentation of the target documents.

Among the detected opportunities, there were 1227 cases, for which precondition #1 had to be examined, because there were order dependencies between style declarations extracted in the *mixin* and declarations remaining in the style rules where the *mixin* would be called. In one case, finding a satisfiable solution for positioning the *mixin* call was not feasible, and thus the migration was not performed. Moreover, there were 1190 cases, for which precondition #2 had to be examined. In all these cases, the original order of the declarations inside the extracted *mixin* could be preserved.

Overall, none of the issues found using our testing method was due to a flaw in the approach we proposed for detecting and extracting *mixins*. All issues were caused by implementation bugs that were eventually fixed, resulting in 100% of the tests being passed. Consequently, we can conclude that the *mixin* migration opportunities proposed by our approach are actually presentation-preserving.

RQ2: Detecting Mixins Defined by Developers

Motivation: The goal of RQ2 is to investigate whether our technique is able to recommend *mixins* that a human expert (i.e., a developer with expertise in the use of preprocessors) would introduce.

Method: To evaluate this research question we first built an *oracle* of human-written *mixins* by extracting all *mixins* in our preprocessor dataset being called at least two times. These *mixins* are suitable for testing our approach, because they introduce duplicated style declarations in the style rules where they are called after transpiling the preprocessor code to generate CSS code. The *mixins* called only once in the preprocessor code cannot be detected by our approach, because they do not introduce duplicated style declarations. Next, we transpiled each preprocessor file and applied our technique to detect all *mixin* migration opportunities in the resulting CSS files.

A *mixin* m, created by applying the migration opportunity mo detected by our approach, matches with a *mixin* m' in the oracle, iff:

1. the set of ISPs styled by m is equal to or is a superset of the ISPs styled by m',
2. m is called in at least all the style rules where m' is called.

The first condition ensures that m styles the same set of properties as m'. This condition is relaxed, so that m could style more ISPs than m'. This relaxation is necessary to deal with cases where the preprocessor developer missed the opportunity to include additional style properties being duplicated in the style rules from which m' was extracted. The second condition ensures that m is called in the same style rules where m' was called. This condition is also relaxed, so that m could be called in more style rules than m'. This relaxation is necessary to deal with cases where the preprocessor developer missed the opportunity to reuse m' in additional style rules. If m' is matched by applying the first relaxed condition, then m' is not a *closed* frequent itemset, since there is at least one superset with the same frequency.

Results: Our approach was able to recover 189 out of the 193 *mixins* in the oracle. In particular, 3 *mixin* migration opportunities detected by our approach were exact matches, 17 contained additional properties (i.e., supersets with the same frequency), 29 were called by additional style rules (i.e., same sets with higher frequency), and 214 contained additional properties and were called by additional style rules (i.e., supersets with higher frequency). The large percentage of inexactly matched *mixin* migration opportunities (260/263 = 98.8%) actually shows that in most of the cases developers under-utilize *mixins*.

We further manually investigated the 4 oracle *mixins* that our approach could not detect. In general, these *mixins* fall into two categories:

Mixins using property interpolation: *Interpolation* is an advanced preprocessor feature allowing, e.g., to use variables to form property names. It is useful for styling different properties that have the same sub-properties (e.g., `margin-top` and `padding-top` can be interpolated as `@{property}-top`), or can take the same values. Figure 39a shows an example of an *interpolated* property name inside a *mixin*, and the resulting CSS code is depicted in Figure 39b. Our technique cannot detect such *mixin* opportunities, since it does not support the parameterization of differences in property names.

```
.inherit(@property) { @{property}: inherit; }
.s1 { .inherit(margin); }
.s2 { .inherit(padding); }
```

(a) Preprocessor Code

```
.s1 { margin: inherit; }
.s2 { padding: inherit; }
```

(b) Generated CSS

Figure 39: Example of interpolated property name

Use of `!important` in arguments: As mentioned before, in our approach we avoid the grouping of properties having a different importance. However, the `!important` rule can be actually used in the arguments of a *mixin* call to parameterize properties having a different value and importance, as shown in the example of Figure 40a, resulting to the CSS code shown in Figure 40b. Our approach does not support for the moment such advanced parameterization of differences.

```
.m(@var){ color: @var; }
.s1 { .m(~'red !important'); }
.s2 { .m(blue); }
```

(a) Preprocessor Code

```
.s1 { color: red !important; }
.s2 { color: blue; }
```

(b) Generated CSS

Figure 40: Example of `!important` use in arguments

Table 13: Threshold-based filtering of opportunities

	Filter	**#Opportunities**	**#Recovered**	**Recall (%)**
I	None	9585	189	97.9
II	#Declarations ≤ 7	8686	180	93.3
III	#Parameters ≤ 2	4421	176	91.2
IV	II & III	4320	169	87.6

6.4.3 Limitations

The success of a recommendation system is associated with the relevance of the recommendations to its users, often measured in terms of precision and recall. Assuming that the *mixins* introduced by developers (e.g., the oracle used in RQ2) constitute the gold standard, our approach can achieve very high recall with a small number of undetected actual *mixins* (i.e., false negatives), but it generates a large number of *mixin* opportunities, and some of them might be considered irrelevant by the developers (i.e., false positives). Although, it is not possible to determine the actual number of false positives without asking the developers' opinion about the recommendations, it is certain that the developers would like to inspect the smallest possible list of *mixin* opportunities that contains most of the relevant ones.

Therefore, we investigated whether it is possible to reduce the number of generated *mixin* opportunities (i.e., recommendations) without jeopardizing recall. To achieve this, we filtered out the *mixin* opportunities having a number of style declarations, or parameters above certain thresholds. The threshold values were automatically derived from the box plot *upper adjacent values* for the oracle used in RQ2. All data points above the upper adjacent value of the box plots are *outliers* that correspond to abnormal *mixins* introduced by developers. Indeed, defining thresholds based on box plot outliers is a statistical approach that has been also used in metric-based rules for detecting design flaws in object-oriented systems [Mar04]. Table 13 shows the number of *mixin* opportunities obtained using different filters along with the number of recovered *mixins* from the oracle. The results show that it is possible to recover close to 90% of the oracle *mixins* with less than half of the original opportunities by applying appropriate threshold-based filters.

Note that, in addition to the aforementioned threshold-based filters, our tool (Chapter 7) provides other filtering and sorting features to further aid the developers in reducing the number of migration opportunities and selecting the most appropriate ones to apply. For instance, the developer can hide all the opportunities that involve a certain style property or a `@media` at-rule, and sort them based on the number of style rules from which a *mixin* is extracted.

6.5 Comparison with Charpentier et al.'s Approach [CFR16]

As mentioned in Chapter 3, Charpentier et al. also worked on the similar problem of automated extraction of *mixins* for CSS, in parallel to us. We gave a summary of their work in Chapter 3, and here we compare it with our approach for detecting *mixin* migration opportunities in more detail. As the authors implemented their technique in a tool called MOCSS, we call the approach with this name henceforth.

6.5.1 Summary of the Method

MOCSS uses formal concept analysis (FCA) for identifying duplicated style declarations that can be extracted as *mixins*. We mentioned that using FCA with the goal of grouping duplicated declarations first appeared in a technical report by Federman and Cook in 2010 [Dav10], and thus, is not a novel idea. The difference of the two approaches is that in MOCSS, for a set of style declarations to be considered as one FCA *attribute*, they just need to have the same property names, while in the Federman and Cook's approach, both style properties and values should be textually equal. This is necessary as the former approach introduces *mixin* parameters whenever the style values are different, while the latter is designed to refactor duplicated style declarations into style rules having grouping selectors. Other than this difference, the two approaches work similarly, i.e., MOCSS also uses a traversal of the concept lattice to extract *mixins*.

Charpentier et al. report the scalability of MOCSS, in addition to the results of a user study with four participants, and conclude that their approach is able to suggest relevant *mixins* for extraction within a few milliseconds.

6.5.2 Comparison

The opportunities and the human involvement in applying them

Firstly, MOCSS does not allow the developers to choose what *mixin* migration opportunity to apply, or in what order they should be applied. In other words, the output of MOCSS is *only one solution* and not a set of opportunities, i.e., it is a set of *mixins* that are applied one after another. The only freedom that is given to the developer is a set of thresholds, e.g., what is the minimum number of times that a *mixin* should be called to be considered for extraction. In any case, we argue that this single solution is not always the *best* one.

As discussed, Federman and Cook state that the concept lattice created from CSS file can be traversed in several different ways, which in turn leads to different refactorings, each of which having advantages/disadvantages. The same can happen for MOCSS.

For example, one traversal of the concept lattice starts by extracting duplicated style declarations that are repeated in the largest number of style rules, and continues until there is no more extraction possible. In such cases, the resulting constructs (being either *mixins* in MOCSS, or style rules with grouping selectors in the Federman and Cook's approach) can include as few as one style declaration. Conversely, another traversal starts from the longest set of duplicated declarations, which can be repeated in as few as two style rules.

Both these traversals can create *mixins* that are desired (or undesired) by the developer, depending on, e.g., the type of the style declarations that are extracted. For instance, the developer might want to extract a *mixin* with only one style declaration for the `animation` property that is repeated in several style rules. This extraction can happen with the first mentioned type of traversal. At the same time, there could be a set of style declarations that are repeated only a few times, but are coherent enough to form a *mixin* (let's say, a *mixin* for grouping `font`, `text-align`, and `word-wrap` in five style rules). Such *mixin* can be extracted from the second traversal; however, it is possible that a larger (and yet, less coherent) set of style declarations have already been extracted, since the traversal aims at maximizing the size of the *mixin* being created (for example, the traversal can first extract a *mixin* for `font`, `text-align`, `word-wrap`, and `color` from two style rules, making the extraction of the mentioned *mixin* that is desired by the developer impossible).

The approach also does not give the user the freedom for choosing names for the introduced *mixins* and the *mixin* parameters. In general, there is no interaction between the user and the approach, and the changes are done globally in the code. We already know from the literature that such global refactorings are less frequent [BMZ+05].

We argue that the developer should be always considered in the loop for any kind of transformation. Our approach, therefore, is designed as a building block of a *recommendation system*, which rather than just giving one solution, attempts to offer the developer several solutions, while helping her in filtering out the undesired ones. The current implementation, however, is not a *complete* recommendation system, as we will discuss in Chapter 8.

Efficacy

In our study, we computed the recall of the approach by mapping the generated *mixin* migration opportunities to the *mixins* that developers manually created. The study by Charpentier el al., however, did not compute the recall of MOCSS. We mentioned that, in theory, both FCA and the frequent pattern mining algorithm are equivalent (as FCA is also used to generate association rules [Smi09]). However, since MOCSS only provides a single solution (i.e., one set of *mixins* that can be refactored together from the CSS file), we hypothesized that it might miss to identify some of the manually-introduced *mixins*.

Therefore, we run MOCSS on the same dataset that we used in our study, to compare our results with it. The experiment is run in the same way we evaluated our approach: a set of manually-developed LESS files are first transpiled to CSS, and then MOCSS is used to generate a preprocessor file from the transpiled CSS files. The introduced *mixins* in the generated preprocessor code and the *mixins* in the original LESS files are then compared to see whether MOCSS is able to recover the original *mixins* manually introduced by the developers. The details of the comparison are given in Section 6.4.2 (RQ2).

Note that, we used similar configuration thresholds in both tools for a fair comparison:

The minimum number of calls for *mixins* is set to **two**, so that the style declarations that shared in as low as two style rules are also extracted,

The minimum number of declarations in *mixins* is set to **one**, so that *mixin* having single style declarations that are repeated across several style rules are also extracted,

The maximum number of parameters defined in *mixins* set equal to ∞, so that the *mixins* can group style declarations with any number of differences across style values.

Results: We found that for the whole dataset, MOCSS introduces 4,057 *mixins* in the migrated preprocessor files. Out of those 1,887 *mixins* actually include style declarations, and the rest (i.e., 2,170) are extra *mixins* that just delegate to other *mixins*.

MOCSS introduces these extra *mixins* in the resulting preprocessor code with the justification that they are needed to keep the ordering of the style rules intact. These extra *mixins*, however, are not normally required: we discussed in this section that the order of style rules are not affected by introducing *mixins*. This is indeed one of the advantages of using *mixins* over the *extend* construct (of course, with the cost of creating duplicated style declarations in the generated code). Instead, the order of the declarations defined inside the style rules could be affected by the location of the introduced *mixin* calls.

Out of the 1,887 non-extra *mixins*, only 70 match the manually-developed *mixins* in the original preprocessor files, i.e., 36% recall. This can be explained by the way MOCSS uses the concept lattice to generate *mixins*.

For instance, suppose that the style properties `text-align` and `font` are repeated across multiple style declarations. Instead of creating one *mixin* for merging the two style properties, MOCSS might generate two *mixins* (i.e., one for each of the properties), and call both of them in the corresponding style rules. The advantage of this approach is that each of these *mixins* can be reused in other style rules where only one of the `text-align` or `font` properties appear (and not both of them). The disadvantage, however, is that it might generate *mixins* that are not desired for the developer. This is indeed supported by the low recall of the tool.

MOCSS also cannot recover the *mixins* that use interpolated style properties (the example of Figure 39), since it also works with the assumption that the style properties are constant strings and will not vary. In contrast, it can recover *mixins* with style values having different use of the `!important` keyword (the example of Figure 40). Since MOCSS does not do any parameterization on the style values, the style declarations `color: red !important` and `color: blue` can be merged. As mentioned, our current implementation does not support this kind of parameterization, but it can be supported later. Notwithstanding, such cases are exceptional and we did not find many of those in the dataset.

Remember that we propose to the developer 9,585 *mixins* for the whole dataset, and we can apply threshold-based filters on the opportunities to decrease their number to as low as 4,320 while keeping the recall still pretty high, i.e., 87.6%. This reduced number of opportunities in our approach is close to what MOCSS actually introduces in the code (which are mostly extra, or undesired by the developers). Again, this shows why it is crucial to let the developers investigate several opportunities, while helping them to choose and apply the ones that are more desirable, rather than providing one set of *mixins* as the single possible solution to the migration problem.

The proposed *mixins*

There are several differences in the *mixins* introduced by MOCSS and our approach, namely:

1. From the alternative ways for *mixin* introducing parameters discussed in Section 6.3.2, MOCSS uses the one that introduces *mixins* with one parameter for all the passed values. We presented the maintainability issues that arising from using a single parameter for *mixins*. The approach that we took, on the other hand, needs more sophisticated analysis of style values and their assigned individual style properties.

2. MOCSS does not unify *mixin* parameters, while we discussed our *mixin* unification technique and its benefits in Section 6.3.2.

3. MOCSS does not consider style values that are equivalent (e.g., for extracting two style declarations `color: blue` and `color: #00F`, a parameter will be created, while we avoid this by marking the two style declarations as equivalent). In addition, equivalent shorthand and individual style declarations are not handled.

4. As mentioned, MOCSS generates extra *mixins* into the generated preprocessor code. Such extra *mixins* will actually hamper the maintainability of preprocessor code base, due to the unnecessary indirection that they create.

Safety of the refactorings

MOCSS neither discusses nor checks any *preconditions*, i.e., the proposed transformations are not necessarily presentation-preserving. Even though both MOCSS and our technique follow the same workflow for testing whether the generated preprocessor codes have the same behavior as the original one – i.e., I) generating preprocessor code, II) compiling the code back to CSS, III) comparing the original CSS file with the generated CSS file in II – MOCSS does not take into account the existence of intra-style rule order dependencies and therefore does not check them when doing the comparison. As a result, it incorrectly reports a preprocessor code to be behavior preserving while it might not be.

To attest this, we fed MOCSS with manually-created test cases similar to what is depicted in Figure 37c, where intra-style rule order dependencies forced a certain ordering for style declarations, and the approach failed to produce a correct preprocessor code. We showed that in the dataset there were more than 1200 *mixins* for which this safety check and reordering of the style rules were necessary.

Scalability

In Figure 41, we have compared the time (in milliseconds) that our approach takes to detect *mixin* migration opportunities with that of MOCSS. This includes the time taken only for *creating* solutions, not for applying the actual refactorings to the code. We configured the two tools with similar configurations, with the values that mentioned before.

As it can be observed, while the median of this time is 130.6 milliseconds, for our approach it is only 42.8 milliseconds. Thus, on the one hand, our approach is on average 4 times faster than MOCSS. On the other hand, there was a file in the dataset on which our method took around 2 minutes to finish (the CSS file of the `brentleemusic.com` website). Investigating this file showed that it actually included a huge number of duplicated declarations around different style rules, leading to a large number of opportunities to compute. MOCSS, however, quickly computed a single solution for this file (in 983 milliseconds). Interestingly, when we investigated the file generated by MOCSS for this case, we observed more than 15 duplicated style declarations in 3 *mixins* and 3 style rules, which included individual vendor-specific properties for the `transition` property (e.g., `transition-timing-function`, `transition-duration`, `transition-property`, with 5 different vendor-prefix definitions). We are unaware of the underlying reasons for this behavior, but this essentially shows that the tool cannot serve as a reliable migrating technique for removing duplicated style declarations in CSS code bases.

The longest time that MOCSS spent on a file to compute a solution was 3133.5 milliseconds, on the CSS file generated for the Flat-UI framework. For our approach, this file took only 709

(a) CSSDev (Our approach) (b) MOCSS [CFR16]

Figure 41: Comparison of the time taken for detecting migration opportunities

milliseconds to finish. Our approach is faster because the library is professionally coded and has less duplicated code, leading to fewer migration opportunities.

We should acknowledge that this comparison is not completely fair. First of all, MOCSS computes only one solution for the extraction of *mixins*, while we generate several possible solutions. Second, looking at the implementation of MOCSS, we understood that they extensively used the quite-newly-introduced `Stream` API of Java 8 along with lambda expressions, which are considered to be slower than their traditional counterparts (e.g., `Iterator`s and enhanced `for`s [Kuk13, Zhi15]).

User Study

Charpentier el al. conducted a user study with four developers to assess the acceptance of the proposed migrations by MOCSS. This is a plus for them, as for now, we lack a user study. Notwithstanding, we believe such a user study done in a genuinely controlled way requires a very careful design with more participants. Consequently, it is planned for the future work.

6.6 Threats to Validity

While the proposed approach for detecting *mixin* opportunities is preprocessor-agnostic, our actual implementation for the introduction of *mixins* currently supports only the Less preprocessor. This is because the source code transformations required to introduce a *mixin* are specific to the *abstract syntactic structure* of the targeted preprocessor. We selected to support Less, because it is slightly more popular among the developers [Coy12]; however, the implemented tool is extensible enough to support any preprocessor. In addition, we have already shown (Chapter 5) that code written in Sass (another popular preprocessor) has very similar characteristics with code written in Less. Thus, examining projects developed in Less does not limit the generalizability of our study. Nevertheless, we used Less files collected from a wide range of web systems, including libraries and websites, to

mitigate the threat to the external validity of our study.

The ultimate approach for testing presentation preservation would be to compare the target documents, before and after applying migration transformations, as they are rendered in the browser. However, a visual comparison would be time-consuming and error-prone. Additionally, the state-of-the-art automatic techniques are computationally intensive, e.g., differentiating screen captures of web pages using image processing methods [RCVO10, MH15, RCPO13]. We instead considered all possible presentation changes a *mixin* can impose on a Style Sheet, and developed a lightweight static analysis method, based on preconditions derived from CSS specifications. This approach was able to reveal several styling bugs due to our faulty implementation, showing that the method is promising in testing whether presentation is preserved.

6.7 Chapter Summary

In summary, the main conclusions and lessons learned are:

1. Our approach facilitates the automatic migration of CSS code to preprocessors, by safely extracting duplicated style declarations from CSS code to preprocessor *mixins*.

2. Our approach is able to recover the vast majority (98%) of the *mixins* that professional developers introduced in websites and Style Sheet libraries.

3. We found that developers mostly under-utilize *mixins* (i.e., they could reuse the *mixins* in more style rules, and/or could eliminate more duplicated style declarations by extracting them into the *mixins*).

4. By applying appropriate threshold-based filters, it is possible to drastically reduce the number of detected *mixin* opportunities without affecting significantly the recall.

In the next chapter, we briefly demonstrate CSSDev, an Eclipse plug-in that contains our implementation of the proposed approaches in Sections 4 and 6.

Chapter 7

CSSDev: A tool suite for the analysis and refactoring of CSS

7.1 Introduction

In the previous sections, we stated several reasons for developing and maintaining CSS being a challenging task, 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. We found that more than 60% of style declarations are duplicated in real-world CSS files,
- The lack of best practices has led to low quality CSS code suffering from various CSS-specific smells [Gha14], 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 [RCVO10, RCPO13]).

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 (and possibly making changes to the corresponding HTML, JavaScript, or any piece of code that generates HTML),

2. Running the web application in one (or, most probably, 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, like Less [SF10] or Sass [Cat06]. 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., 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 chapters (i.e., Chapters 4 and 6), we proposed approaches for refactoring duplicated code in CSS in a presentation-preserving manner, by grouping duplicated style declarations into new selectors, or by migrating CSS code to a preprocessor language by extracting function-like constructs (i.e., *mixins*) from duplicated style declarations. 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 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 this chapter.

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

Note: Earlier version of the work done in this chapter has been published in the Proceedings of the 39th International Conference on Software Engineering (ICSE 2017), Tool Demonstration Session [MT17].

7.2 Tool Design

CSSDev consists of the following main modules:

7.2.1 CSS Model Generator Module

This module is responsible for generating a lightweight, hierarchical model of CSS, as described in Chapter 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, i.e., the *individual style properties* (or ISP) associated with each style value, as described in Chapter 6. For example, in the style declaration `border: dotted 1em \#F0F`, for the value `dotted`, the model stores that it is a CSS keyword that defines the *style* of the border that appears around an element, i.e., ISP(`dotted`) = `"style"`.

The model 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 [Wor17]. 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, mainly for two reasons:

- CSS specifications change rapidly and a parser might become obsolete.
- CSS parsers that are completely W3C-complaint try to skip the invalid portions of CSS code and continue parsing the CSS file from the first valid CSS rule/declaration that is found. Parsers in the web browsers are from this category.

  On the other hand, there are parsers that fail fast on the invalid input. Based on the usage, the CSS analysis tool should be flexible enough for changing parsers.

  For instance, when analyzing real-world CSS files (e.g., when doing empirical studies on CSS code), a CSS parser from the first category should be used, as these CSS files might contain invalid CSS that is not caught and fixed due to the lenience of web browsers in parsing CSS code. In contrast, when developing CSS (or CSS preprocessor) code, for applying source code analysis or refactoring it is preferred to use a fast-failing parser, to spot the mistakes early and avoid potential presentation bugs.

Our model provides the necessary abstractions to make easy the replacement of CSS parsers with different capabilities. In any case following the *dependency inversion* principle is a good practice in software development.

7.2.2 Duplication Module

This module is responsible for efficiently detecting different types of duplicated style declarations in CSS, and identifying opportunities for refactoring. This is where the *frequent pattern mining* algorithms (mainly the FP-Growth algorithm, while we also provide an implementation of the Apriori algorithm, which is not recommended as it is not as scalable as FP-Growth) are implemented. The duplication detection is done on the CSS model.

7.2.3 Crawler Module

This module is responsible for crawling HTML target documents for which the CSS file under analysis is used. As mentioned, we use Crawljax [MvDL12], a tool for crawling dynamic web applications relying on JavaScript to handle user interactions.

7.2.4 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 *presentation-preserving* to apply, i.e., the resulting CSS file produces the same presentation after refactoring.

7.2.5 Preprocessor Module

This module deals with CSS *preprocessor languages*. It allows to safely migrate existing CSS code to preprocessors by extracting duplicated style declarations to function-like constructs, i.e., it contains the implementation of the approach described in Chapter 6.

Several preprocessors have been introduced in the industry, and they have been extensively adopted by developers. The implementation of this module is mostly preprocessor-agnostic, i.e., it can generate transformations for virtually any CSS preprocessor language.

7.2.6 Refactoring Module

When developing automatic transformations for more mature programming languages (e.g., Java and C++), there are quite descent implementations for *AST rewrite* engines, which are responsible for translating AST-level modifications into low level textual edits. For CSS, however, we were not able to find such implementation. Consequently, we developed the necessary logic for fulfilling

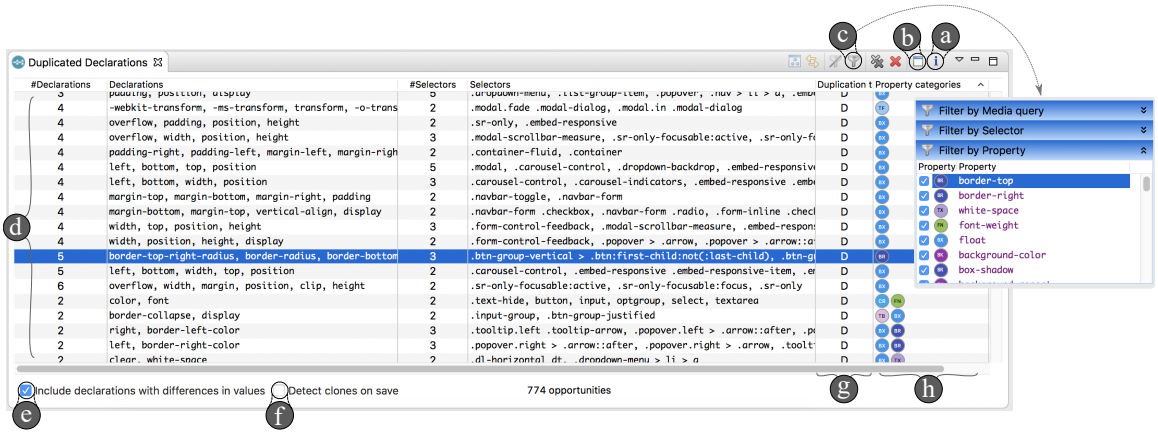


Figure 42: Duplication view

the task of extracting textual edits from model-level transformations in the Refactoring Module of CSSDev.

7.3 Tool Features

7.3.1 Clone Detection

CSSDev provides the functionality for detecting three types of equivalent style declarations within CSS code (type I, II and III duplication, discussed in detail in Chapter 4), in addition to the duplicated style declarations with differences in style values which can be extracted using CSS preprocessor *mixins* (discussed in Chapter 6).

The main plugin's view is shown in Figure 42. The user can initiate duplication detection by selecting a CSS file in the workspace, and clicking on the "Detect" command in the view (Figure 42(a)). This can also be done whenever the user saves the CSS file (Figure 42(f)). In either case, the duplicated style declarations are listed in a table, where each of the rows is an opportunity for refactoring (Figure 42(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 42(g)).

Moreover, for each refactoring opportunity, the user can see the unique *style property categories* to which the involved style declarations belong (Figure 42(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 (Chapter 5). Indeed, the plugin's view allows the developer to sort the detected opportunities 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 42ⓒ). The developer may also show/hide the opportunities that contain duplicated declarations having differences in their style values (Figure 42ⓔ).

7.3.2 Extracting Order Dependencies

Normally, the relative order of declarations in a CSS file does not matter, unless there exist *order dependencies* between different style rules, which force certain constraints in the style rule positions within the CSS file. As discussed, order dependencies exist with respect to some target documents, and a refactoring that changes the order of style declarations 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 [BM13].

CSSDev uses an automatic crawler, Crawljax [MvDL12], 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 (Figure 43), 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

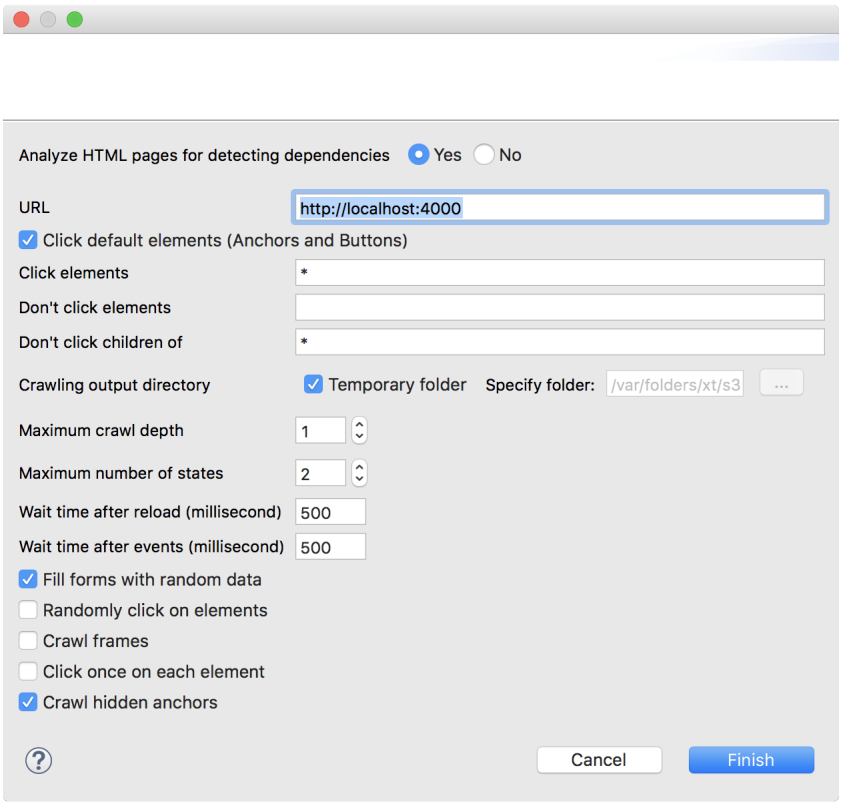


Figure 43: Crawler settings

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 presentation-preserving refactorings.

The developer can also visually investigate the existing dependencies extracted after crawling the target documents (Figure 44). In the graph shown in the view, the nodes stand for the style declarations, whereas the edges correspond to the dependencies between the style rules and style declarations defined within them. There are four types of edges, namely:

1. **Cascading overriding dependencies**: Dependencies due to the *order* of style rules having selectors with the same specificity, that select the same elements in some target document state, and share one or more style properties,

2. **Specificity overriding dependencies**: Dependencies between the style rules with different specificities that style a set of common style properties for the same elements in some target document state. Remember that, in such cases, the style declarations belonging to the style rule having the selector with lower specificity will be overridden.

3. **Importance dependencies**: A special type of dependency between the style rules when

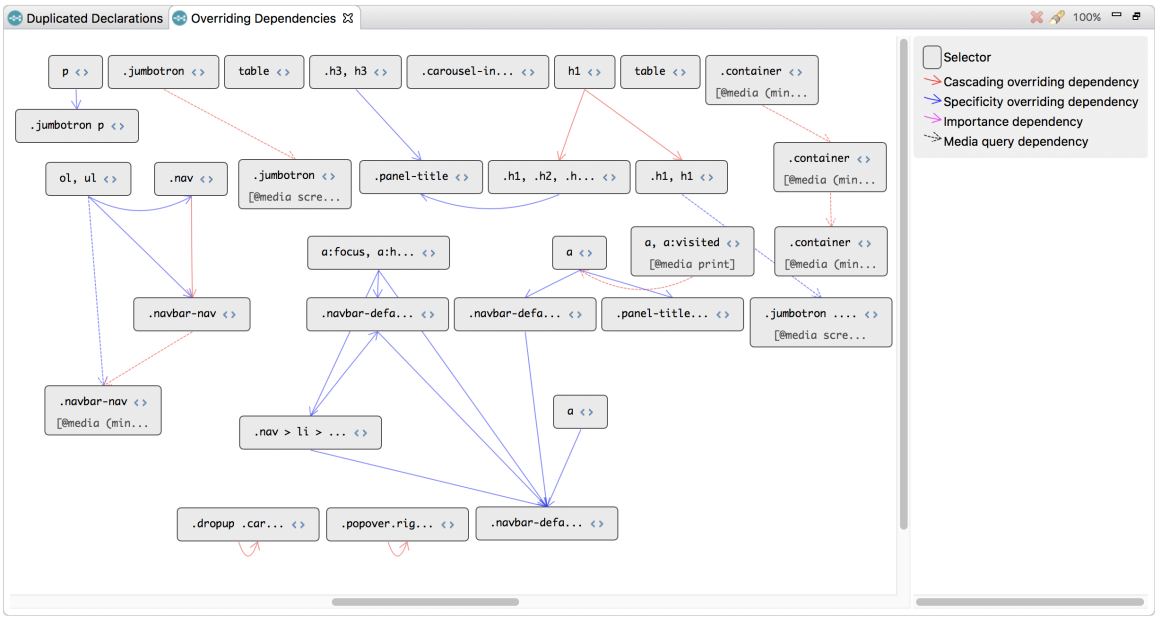


Figure 44: Overriding dependencies View

they style the same property for the same elements, but only in one of the style rules the corresponding style declaration uses the `!important` keyword (meaning that it overrides the other style declaration in the other style rule),

4. **Media query dependencies**: A dependency between the style rules defined inside a specific `@media` at-rule, and the style rules defined in (and hence, overridden by) the style rules defined for the *default* media. The default media is the media which is applied to all style rules when there is no explicit `@media` at-rule defined (one can alternatively declare this media by defining `@media all`), and it means *any media with any characteristic*.

In the dependency visualization view, it is possible to show/hide types of dependencies on demand, which can help in reading the graph when it is too crowded. The intra-style rule order dependencies, which should be considered when extracting a *mixin*, can be also identified by the nodes having self loops.

This view is particularly useful for applying *change impact analysis* on the CSS code, i.e., to trace the possible side effects of applying a change in the CSS code. The developer can see the elements that are selected by each style rule, by clicking on the <> icon in each node. The tool will in turn display all the selected DOM elements in each of the explored states in the crawling (Figure 45). The elements can be distinguished by their corresponding HTML code, and an XPath expression that locates them in the DOM tree. The tool also shows the shortest path in the UI that can be taken to reach each state (i.e., the chain of clicking specific elements).

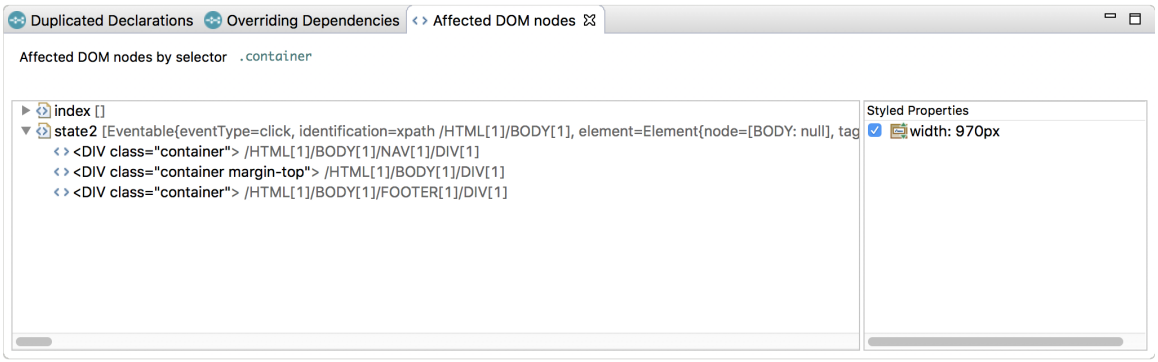


Figure 45: Affected DOM elements view

7.3.3 Clone Refactoring

Once an opportunity is selected in the Duplication view (i.e., Figure 42), 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*. Alternatively, a *parameterless mixin* can be extracted from the duplicated declarations.

In the first case, a dialog will be shown (Figure 46), giving the developer the freedom to change several options, including:

ⓐ The name of the extracted *mixin*,

ⓑ The name of each of the extracted *mixin*'s parameters,

ⓒ The selectors of the style rules from which the *mixin* should be extracted,

ⓓ 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.

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 style rule with a grouping selector, only options ⓒ and ⓓ will be available, as the two first ones are not applicable in this case (i.e., a grouping selector is automatically named by separating the individual

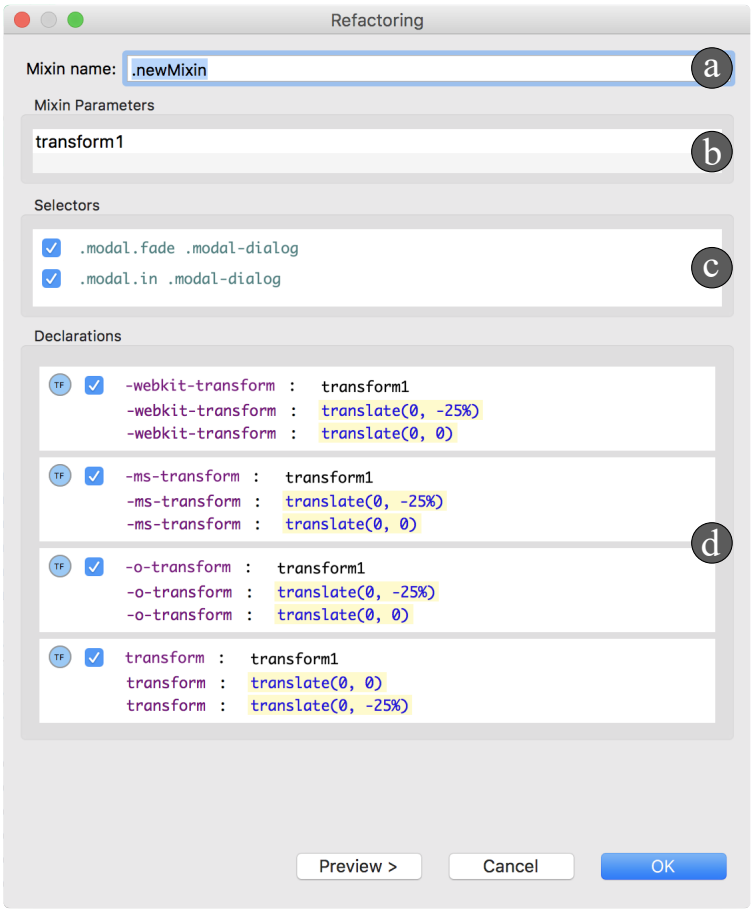


Figure 46: Refactoring options wizard

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 (Chapters 4 and 6), and generates the actual source code transformations. In some cases, CSSDev needs to reorder some of the style declarations or style rules, in order to make sure that the changes will preserve the presentation semantics of the resulting code. The developer gets a preview of all the changes (Figure 47). 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.

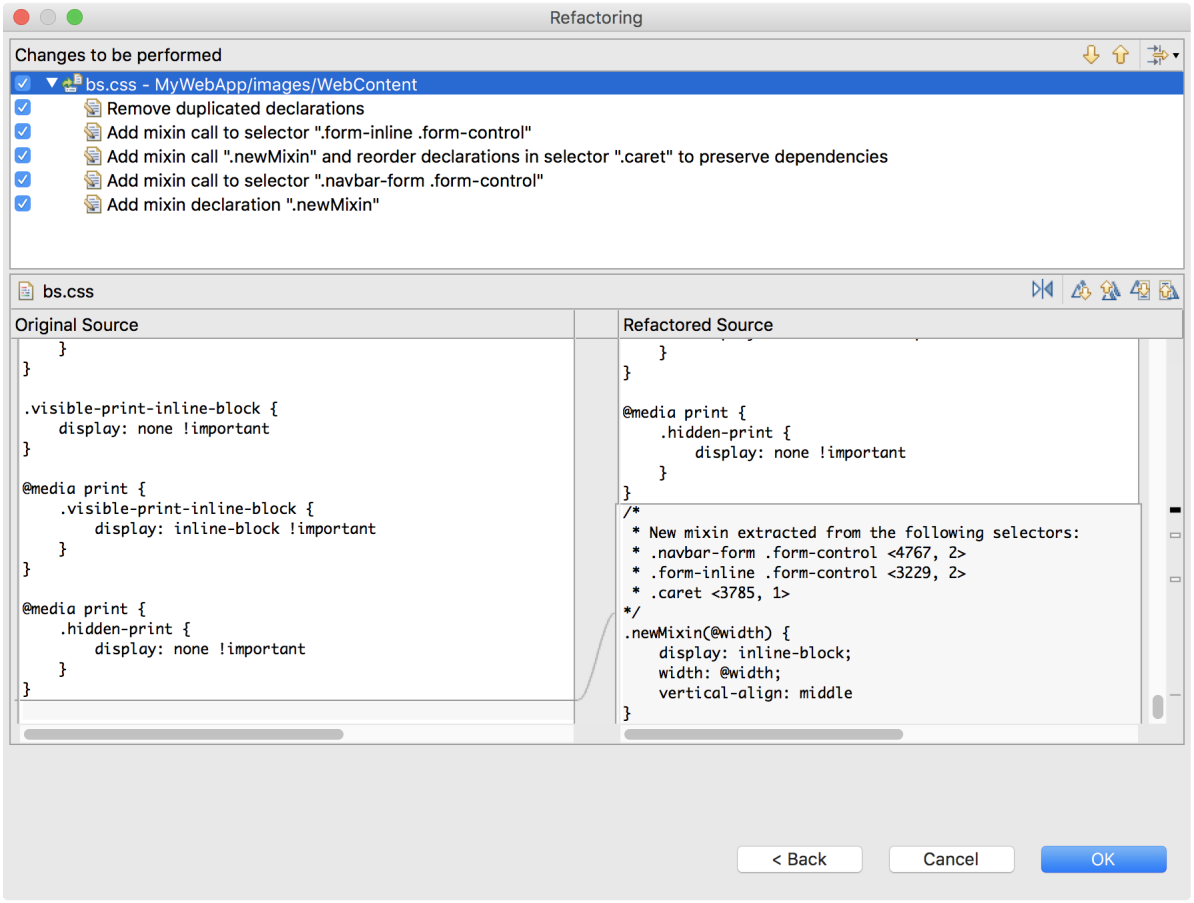


Figure 47: Refactoring preview

7.4 Chapter Summary

In this chapter, we demonstrated the code features of CSSDev, implemented as an Eclipse plugin, that allows analyzing and refactoring CSS code. The CSSDev infrastructure allows implementing even more diverse set of analysis on CSS (and CSS preprocessor code) in the future, and it is possible to create plugins for other IDEs, as the code of CSSDev does not depend on Eclipse. The source code of both CSSDev and its Eclipse plugin are available on the GitHub page of the author[2].

In the next chapter, we conclude the thesis and discuss some possible future works.

[2]http://github.com/dmazinanian

Chapter 8

Conclusions and Future Work

For the past twenty years, Cascading Style Sheets has stabilized its role in the industry as the standard styling language for web, mobile, and desktop applications. CSS is as widespread as web, extensively used by developers, constantly evolves, and code bases written using it frequently undergo maintenance. It, however, still suffers from many shortcomings, e.g., the scarcity of the empirically-validated and globally-accepted best practices, and immature tool support. This has worsened by the fact that academia has not put much research effort into it.

In this chapter, we summarize the findings of this thesis, and discuss some promising directions for the future work.

8.1 Summary of the Findings

In this thesis:

- We looked at the problem of duplicated code in CSS, and found that code duplication is extensive in CSS files: on average 66% of the style declarations are repeated at least once in the CSS code bases of the 38 analyzed real web applications.
- We devised a technique, based on *frequent pattern mining* algorithms, for reducing duplicated code in CSS, and discussed the preconditions that need to be met, to make sure that the refactorings are safe.
- We found that there is a significant number of such presentation-preserving refactoring opportunities in CSS files (62 on average), and applying them can lead to, on average, 8% size reduction in the examined CSS files.

- We conducted the first empirical study on the use of CSS preprocessors, in the code base of 150 web sites, to understand how developers take advantage of the features that these languages provide over CSS, and provided implications for researchers, tool builders, and CSS preprocessor language designers.
- We found that developers who use CSS preprocessors:
  - prefer to use global variables over the local ones (89.28% vs 10.72%),
  - especially use variables to store color values (45.98% of all variables are defined to store color values),
  - take advantage of the *nesting* feature extensively (78% of the style rules are nested), even in very shallow hierarchies (one or two levels),
  - tend to use *mixins* to avoid duplication (63% of the *mixins* are called two or more times),
  - tend to create *mixins* with a small number of parameters (68% of the *mixins* have either one or no parameters),
  - tend to keep *mixins* relatively small (80% of the *mixins* include 5 or less declarations),
  - significantly introduce *mixins* for styling vendor-specific style declaration (42% of the *mixins* are used for this purpose),
  - prefer to use parameterless *mixins* to avoid the caveats associated with the *extend* construct, while accepting the duplication that using *mixins* can create in the resulting CSS files.
- We proposed an approach, again based on the frequent pattern mining algorithms, to safely migrate existing CSS code bases to take advantage of CSS preprocessor *mixins* to avoid duplicated code, with a high recall (98%).
- We noticed that developers could under-utilize *mixins*, i.e., they could reuse the *mixins* in more style rules, and/or could eliminate more duplicated style declarations by extracting them into the *mixins*.
- We found that, while there could be a huge number of opportunities in CSS code bases, applying appropriate threshold-based filters can drastically reduce the number of these opportunities without affecting significantly the recall.
- We introduced CSSDev, a tool suite that facilitates analyzing CSS code and applying refactoring and migration activities on it.

8.2 Future Work

8.2.1 Current Limitations

We should acknowledge that the most notable shortcoming of this thesis is the fact that we did not incorporate developers' opinions in our studies.

As an example, the *maintainability improvements* that can be gained by using our refactoring/migration approaches have not been investigated. Ultimately, the most reasonable way to find this out is through a controlled user-study, which is usually very difficult to conduct.

For example, the proposed refactoring/migration techniques can blindly group style declarations that are not *semantically coherent*, leading to a code that is actually less readable. As a possible approach for alleviating this problem, we can propose a ranking mechanism that takes into account other characteristics (e.g., the aforementioned semantic coherence) rather than only size reduction, when choosing refactoring opportunities to apply. This can help developers in *prioritizing* opportunities for refactoring and filtering out the ones that will be unlikely to apply. Although we can take advantage of the knowledge obtained from our empirical study that we conducted to find out how developers use CSS preprocessor language features (Chapter 5), this knowledge should be further complemented by 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 conducting a lab study, asking the participating developers to rate the refactoring opportunities proposed by our approach, and seeking their reasoning behind the ratings. The developers will also provide feedback on the usability of the CSSDev Eclipse plug-in and suggest ways to improve it.

One other possible way to find out what affects the refactorings is to conduct a study on the CSS code collected from projects hosted on repository hosting services (e.g., GitHub), submit the patches to them where duplicated code is refactored using any of our techniques, and further ask the developers to explain the reasons for accepting/rejecting the patches.

After a set of characteristics that affect the acceptance/rejection of the refactorings are found, we can look into machine learning approaches or train statistical models for ranking opportunities. Eventually, the fact that we have provided CSSDev with an interactive user interface allows incorporating this ranking mechanism in an effective way.

In addition, developer's opinions can also be sought to complement our empirical study on the use of CSS preprocessors. For instance, we found cases where developers did not nest style rules where nesting was applicable. The reason for this (and for several other similar behaviors) can be only revealed by asking developers. Again, GitHub developers are an appropriate target for this study: we can, for instance, follow the *firehouse interview research method* [MHZBN15, STV16, MKTD17],

where developers will be asked right after they push a change that contains an *unknown* or *unexpected* behavior to the repository, to explain the underlying reasons behind it.

Another (yet more specific and smaller) problem that we can see in this thesis is the approach that we took for assigning correct individual style properties (i.e., ISPs) to the style values. We have done this by manually studying CSS specifications, and hard-coding the rules for assigning ISPs. There could be a way to take advantage of the existing tests given by W3C to infer the ISPs. More ambitiously, we could investigate the possibility of extracting these rules from the text of the CSS specifications, as there are usually multiple examples for the possible values that each style property can accept and how they should be interpreted by the web browsers, in the specifications.

8.2.2 Other Possible Opportunities for Future Research

There is a long list of opportunities for future research on CSS; we provide some of them here.

- One very important direction for future research is to study how CSS code bases are maintained. As said, previous work shows that CSS code undergoes frequent changes [GZ16]. But we don't know *how the changes look like*. For instance, are the changes mostly bug-fixes, or addition of new features? To what extent developers add new code to CSS that *overrides* the existing code? Is this premise true that CSS is a *write-only programming language*? (in other words, both for fixing bugs and adding new features, CSS developers tend to add new code, making the existing code obsolete. This can be considered as a *misuse* for the cascading feature of CSS). Are there refactoring activities other than eliminating duplication in CSS? How does the amount of duplicated code and other code smells change over time in CSS? In general, how much do developers care about CSS code quality and try to improve it, or are there other characteristics of CSS code (e.g., its performance) that matter more to the developers?

- More interestingly, given that using CSS preprocessors is now a trend in the industry, is there a significant difference in the answers to the mentioned questions, when developers prefer to use CSS preprocessors over *vanilla* CSS? If the code written in a CSS preprocessor language is more maintainable, do we see migration activities other than introducing *mixins* in CSS repositories (i.e., from vanilla CSS to a preprocessor)?

- Another direction is to study the performance (and energy consumption) anti-patterns in CSS, and to provide refactorings for them. We mentioned one related work to this done by Jovanovski and Zaytsev [Jov16, JZ16] on critical CSS style rules, but to our knowledge, there is no other study on this issue.

As an example, an incautious use of some CSS properties in specific way can force the web browser to unnecessarily paint the whole view on each scroll event. Spotting such problems and suggesting fixes would be very useful for the CSS community.

- It is possible to extend the refactoring capabilities of CSSDev to incorporate eliminating the code smells that were previously studied (e.g., [Gha14]). However, it is first important to revisit the code smells and study if developers really see them as bad coding practices that can lead to future maintenance difficulties.

- As we discussed, the current techniques for testing CSS are very limited and immature. Both the frozen DOM technique and the methods based on image comparison might capture too detailed information, resulting in the incorrect failing of assertions, or flaky tests. We believe that it is possible to propose ways to automatically generate test cases for these approaches to make them more useful with less false positives. Specifically for the Frozen DOM technique, we might be able to come up with an approach for automatically extracting the reference (i.e., frozen) DOM.

  Proposing a full-fledged unit testing framework for CSS is also greatly needed in the industry. There are, however, several questions that should be answered, for example, what is a *unit* in CSS testing? How can we abstract out the details of DOM and visual presentation of web pages, and how should the developer define oracles?

- CSS evolves very fast, but web applications usually do not keep up with this speed in terms of adopting the new features. This is partly because not all web browsers implement the new features at the same pace and in a consistent way, so there is always the chance that a web application's user interface breaks in some web browsers, if it adopts a new feature from CSS. In any case, automatic refactoring tools can help in making the transition smooth. For instance, existing HTML pages can be refactored to take advantage of the new *Grid System* in CSS for the layout, that makes both CSS and HTML code much cleaner and simpler. This will need a sophisticated analysis to understand the role of the CSS style rules that participate in making the layout of the web application. As mentioned before, a similar approach was introduced by Mao et al. [MCD07] for refactoring HTML pages from `<table>`-based to `<div>-based` layouts.

Bibliography

[ABRC12] César Acebal, Bert Bos, María Rodríguez, and Juan Manuel Cueva. ALMcss: A Javascript Implementation of the CSS Template Layout Module. In *Proceedings of the 2012 ACM Symposium on Document Engineering (DocEng)*, pages 23–32, 2012.

[Ace10] César Fernéndez Acebal. *ALMcss: Separation between Structure and Presentation on the Web with CSS Advanced Layout.* Ph.D. Thesis, Departamento de Informática, Universidad de Oviedo, 2010.

[AMIO12] Adewole Adewumi, Sanjay Misra, and Nicholas Ikhu-Omoregbe. Complexity metrics for cascading style sheets. In *12th International Conference on Computational Science and Its Applications (ICCSA)*, pages 248–257, 2012.

[BBMS99] Greg J. Badros, Alan Borning, Kim Marriott, and Peter Stuckey. Constraint Cascading Style Sheets for the Web. In *Proceedings of the 12th Annual ACM Symposium on User Interface Software and Technology (UIST)*, pages 73–82, New York, NY, USA, 1999. ACM.

[BGL14a] Martí Bosch, Pierre Genevès, and Nabil Layaïda. Automated refactoring for size reduction of css style sheets. In *Proceedings of the 2014 ACM Symposium on Document Engineering (DocEng)*, pages 13–16, 2014.

[BGL14b] Martí Bosch, Pierre Genevès, and Nabil Layaïda. Automated and Semantics-Preserving CSS Refactoring. Technical report, HAL - Inria Open Archive, Nov. 2014.

[BGL15] Martí Bosch, Pierre Genevès, and Nabil Layaïda. Reasoning with style. In *Proceedings of the 24th International Conference on Artificial Intelligence (IJCAI)*, pages 2227–2233, 2015.

[BK01] Cornelia Boldyreff and Richard Kewish. Reverse engineering to achieve maintainable WWW sites. In *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE)*, pages 249–257, 2001.

[Ble15] Ben Bleikamp. Sass at GitHub . https://vimeo.com/86700007, 2015. Accessed: 2017-07-31.

[BM13] Zahra Behfarshad and Ali Mesbah. Hidden-Web Induced by Client-side Scripting: An Empirical Study. In *Proceedings of the 13th International Conference on Web Engineering (ICWE)*, pages 52–67, 2013.

[BMZ+05] Jim Buckley, Tom Mens, Matthias Zenger, Awais Rashid, and Günter Kniesel. Towards a Taxonomy of Software Change. *J. Softw. Maint. Evol.*, 17(5):309–332, September 2005.

[Car14] Carlo Zapponi. GitHut - Programming Languages and GitHub. http://githut.info/, 2014. Accessed: 2017-06-06.

[Cat06] Hampton Catlin. SASS: Syntactically Awesome Style Sheets. http://sass-lang.com/, 2006. Accessed: 2017-06-06.

[CD04] James R Cordy and Thomas R. Dean. Practical language-independent detection of near-miss clones. In *Proceedings of the 14th Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 1–12, 2004.

[CFR16] Alan Charpentier, Jean-Rémy Falleri, and Laurent Réveillère. Automated Extraction of Mixins in Cascading Style Sheets. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 56–66, Oct 2016.

[CLM04] Fabio Calefato, Filippo Lanubile, and Teresa Mallardo. Function clone detection in web applications: a semiautomated approach. *Journal of Web Engineering*, 3(1):3–21, 2004.

[Con11] World Wide Web Consortium. Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification – Syntax and basic data types. Technical report, World Wide Web Consortium, June 2011.

[Con12] World Wide Web Consortium. Media Queries. Technical report, World Wide Web Consortium, June 2012.

[Con13] World Wide Web Consortium. CSS Syntax Module Level 3. Technical report, World Wide Web Consortium, November 2013.

[Con15] World Wide Web Consortium. CSS Custom Properties for Cascading Variables Module Level 1. Technical report, World Wide Web Consortium, 2015.

[Con16a] World Wide Web Consortium. CSS Object Model (CSSOM), W3C Working Draft, 17 March 2016. Technical report, World Wide Web Consortium, 2016.

[Con16b] World Wide Web Consortium. CSS Values and Units Module Level 3 – Absolute lengths. Technical report, World Wide Web Consortium, September 2016.

[Coy12] Chris Coyier. Popularity of CSS Preprocessors. `http://css-tricks.com/poll-results-popularity-of-css-preprocessors/`, 2012. Accessed: 2017-06-06.

[CPO12] Shauvik Roy Choudhary, Mukul R. Prasad, and Alessandro Orso. CrossCheck: Combining Crawling and Differencing to Better Detect Cross-browser Incompatibilities in Web Applications. In *Proceedings of the Fifth International Conference on Software Testing, Verification and Validation (ICST)*, pages 171–180, Washington, DC, USA, 2012. IEEE Computer Society.

[CRTR11] Oscar Callaú, Romain Robbes, Éric Tanter, and David Röthlisberger. How Developers Use the Dynamic Features of Programming Languages: The Case of Smalltalk. In *Proceedings of the 8th Working Conference on Mining Software Repositories (MSR)*, pages 23–32, 2011.

[CtHdt13] James Cryer and the Huddle development team. PhantomCSS: Visual/CSS regression testing with PhantomJS. `https://github.com/Huddle/PhantomCSS`, 2013. Accessed: 2017-06-26.

[Dav10] David Federman and William R. Cook. Applying Formal Concept Analysis to Cascading Style Sheets. Technical report, The University of Texas at Austin, 2010.

[DFST04] Andrea De Lucia, Rita Francese, Giuseppe Scanniello, and Genoveffa Tortora. Reengineering web applications based on cloned pattern analysis. In *Proceedings of 12th IEEE International Workshop on Program Comprehension (IWPC)*, pages 132–141, 2004.

[DFST05] Andrea De Lucia, Rita Francese, Giuseppe Scanniello, and Genoveffa Tortora. Understanding cloned patterns in web applications. In *Proceedings of the 13th International Workshop on Program Comprehension (IWPC)*, pages 333–336, 2005.

[DLDP01] Giuseppe Antonio Di Lucca and Massirniliano Di Penta. Clone analysis in the web era: an approach to identify cloned web pages. In *Proceedings of the 7th IEEE Workshop on Empirical Studies of Software Maintenance (WESS)*, pages 107–113, 2001.

[DLDPF02] Giuseppe Antonio Di Lucca, Massirniliano Di Penta, and Anna Rita Fasolino. An approach to identify duplicated web pages. In *Proceedings of the 26th Annual International Computer Software and Applications Conference (COMPSAC)*, pages 481–486, 2002.

[DRNN14] Robert Dyer, Hridesh Rajan, Hoan Anh Nguyen, and Tien N. Nguyen. Mining Billions of AST Nodes to Study Actual and Potential Usage of Java Language Features. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, pages 779–790, 2014.

[DWC04] Shirley Dowdy, Stanley Wearden, and Daniel Chilko. *Statistics for research*. Wiley-Interscience, 3rd edition, 2004.

[EBN02] Michael D. Ernst, Greg J. Badros, and David Notkin. An Empirical Analysis of C Preprocessor Use. *IEEE Transactions On Software Engineering*, 28(12), 2002.

[Ede14] Daniel Eden. Move slow and fix things. `http://www.thedotpost.com/2015/12/daniel-eden-move-slow-and-fix-things`, 2014. Talk at the dotCSS conference. Accessed: 2017-06-06.

[FBBO99] Martin Fowler, Kent Beck, John Brant, and William Opdyke. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[FGLD13] Lyle Franklin, Alex Gyori, Jan Lahoda, and Danny Dig. LAMBDAFICATOR: From Imperative to Functional Programming through Automated Refactoring. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, pages 1286–1289, 2013.

[Gha14] Golnaz Gharachorlu. *Code smells in Cascading Style Sheets: an empirical study and a predictive model*. Master's thesis, University of British Columbia, 2014.

[GL10] Joseph(Yossi) Gil and Keren Lenz. The use of overloading in java programs. In *Object-Oriented Programming (ECOOP)*, volume 6183 of *Lecture Notes in Computer Science*, pages 529–551. 2010.

[GLQ12] Pierre Genevès, Nabil Layaïda, and Vincent Quint. On the analysis of cascading style sheets. In *Proceedings of the 21st International Conference on World Wide Web (WWW)*, pages 809–818, 2012.

[GMBCM13] Vahid Garousi, Ali Mesbah, Aysu Betin-Can, and Shabnam Mirshokraie. A systematic mapping study of web application testing. *Information and Software Technology*, 55(8):1374 – 1396, 2013.

[GMD+10] Mark Grechanik, Collin McMillan, Luca DeFerrari, Marco Comi, Stefano Crespi, Denys Poshyvanyk, Chen Fu, Qing Xie, and Carlo Ghezzi. An Empirical Investigation into a Large-scale Java Open Source Code Repository. In *Proceedings of the 2010 International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–10, 2010.

[Gon15] Boryana Goncharenko. *Detecting Violations of CSS Code Conventions.* Master's thesis, Universiteit van Amsterdam, Amsterdam, The Netherlands, 2015.

[Goo15] Google Inc. Google Closure Tools. `https://developers.google.com/closure`, 2015. Accessed: 2015-11-05.

[GZ15] Boryana Goncharenko and Vadim Zaytsev. Reverse Engineering CSS Coding Conventions. In *Postproceedings of 2015 Seminar on Advanced Techniques and Tools for Software Evolution (SATToSE)*, 2015.

[GZ16] Boryana Goncharenko and Vadim Zaytsev. Language design and implementation for the domain of coding conventions. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering (SLE)*, pages 90–104, 2016.

[HLO15] Matthew Hague, Anthony W. Lin, and C.-H. Luke Ong. 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)*, pages 1–19, 2015.

[HPY00] Jiawei Han, Jian Pei, and Yiwen Yin. Mining Frequent Patterns Without Candidate Generation. *SIGMOD Record*, 29(2):1–12, 2000.

[JDHW09] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. Do code clones matter? In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, pages 485–495, 2009.

[JLM+09] Christopher Grant Jones, Rose Liu, Leo Meyerovich, Krste Asanović, and Rastislav Bodík. Parallelizing the web browser. In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism (HotPar)*, pages 7–7, 2009.

[JM99] Philip M. Marden Jr. and Ethan V. Munson. Today's style sheet standards: the great vision blinded. *Computer*, 32(11):123–125, Nov 1999.

[Jov16] Gorjan Jovanovski. *Critical CSS Rules: Decreasing Time to First Render by Inlining CSS Rules for Over-the-Fold Elements.* Master's thesis, Universiteit van Amsterdam, Amsterdam, The Netherlands, July 2016.

[JZ16] Gorjan Jovanovski and Vadim Zaytsev. Critical CSS Rules—Decreasing time to first render by inlining CSS rules for over-the-fold elements. In *Postproceedings of 2016 Seminar on Advanced Techniques and Tools for Software Evolution (SATToSE)*, 2016.

[KETF07] Adam Kieżun, Michael D. Ernst, Frank Tip, and Robert M. Fuhrer. Refactoring for Parameterizing Java Classes. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, pages 437–446, 2007.

[KKI02] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[KN09] Matthias Keller and Martin Nussbaumer. Cascading Style Sheets: A Novel Approach Towards Productive Styling with Today's Standards. In *Proceedings of the 18th International Conference on World Wide Web (WWW)*, pages 1161–1162, 2009.

[KN10] Matthias Keller and Martin Nussbaumer. CSS code quality: a metric for abstractness; or why humans beat machines in CSS coding. In *Proceedings of the 7th International Conference on the Quality of Information and Communications Technology (QUATIC)*, pages 116–121, 2010.

[KSR07] Raffi Khatchadourian, Jason Sawin, and Atanas Rountev. Automated Refactoring of Legacy Java Software to Enumerated Types. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, pages 224–233, 2007.

[Kuk13] Sergey Kuksenko. JDK 8: Lambda Performance study. `http://www.oracle.com/technetwork/java/jvmls2013kuksen-2014088.pdf`, 2013. Accessed: 2017-07-01.

[Lei07] Antonio Menezes Leitao. Migration of Common Lisp Programs to the Java Platform - The Linj Approach. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 243–251, 2007.

[Lie94] Håkon Wium Lie. Cascading HTML Style Sheets; Proposal published 10 Oct 1994. `http://www.w3.org/People/howcome/p/cascade.html`, 10 1994. Accessed: 2017-07-07.

[Lie05] Håkon Wium Lie. *Cascading Style Sheets*. Ph.D. Thesis, University of Oslo, Norway, 2005.

[LKL+13] Hsiang-Sheng Liang, Kuan-Hung Kuo, Po-Wei Lee, Yu-Chien Chan, Yu-Chin Lin, and Mike Y. Chen. SeeSS: Seeing What I Broke – Visualizing Change Impact of

Cascading Style Sheets (CSS). In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology (UIST)*, pages 353–356, 2013.

[LM03] Filippo Lanubile and Teresa Mallardo. Finding function clones in web applications. In *Proceedings of the 7th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 379–386, 2003.

[LW08] Angela Lozano and Michel Wermelinger. Assessing the effect of clones on changeability. In *Proceedings of the 24th IEEE International Conference on Software Maintenance (ICSM)*, pages 227–236, 2008.

[Mad13] Simon Madine. Hardy: Automated CSS Testing. `https://github.com/thingsinjars/Hardy`, 2013. Accessed: 27 June 2017.

[Mar04] Radu Marinescu. Detection strategies: metrics-based rules for detecting design flaws. In *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM)*, pages 350–359, 2004.

[MB10] Leo A. Meyerovich and Rastislav Bodik. Fast and parallel webpage layout. In *Proceedings of the 19th International Conference on World Wide Web (WWW)*, pages 711–720, 2010.

[MCAA15] Douglas H. Martin, James R. Cordy, Bram Adams, and Giulio Antoniol. Make It Simple - An Empirical Analysis of GNU Make Feature Use in Open Source Projects. In *Proceedings of the 23rd IEEE International Conference on Program Comprehension (ICPC)*, 2015.

[MCD07] Andy Y. Mao, James R. Cordy, and Thomas R. Dean. Automated conversion of table-based websites to structured stylesheets using table recognition and clone detection. *Proceedings of the 17th Conference of the Center for Advanced Studies on Collaborative Research (CASCON)*, pages 12–26, 2007.

[MH15] Sonal Mahajan and William G. J. Halfond. WebSee: A Tool for Debugging HTML Presentation Failures. In *Proceedings of the 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–8, 2015.

[MHZBN15] Emerson Murphy-Hill, Thomas Zimmermann, Christian Bird, and Nachiappan Nagappan. The design space of bug fixes and how developers navigate it. *IEEE Transactions on Software Engineering*, 41(1):65–81, Jan 2015.

[MKTD17] Davood Mazinanian, Ameya Ketkar, Nikolaos Tsantalis, and Danny Dig. Understanding the use of lambda expressions in Java. In *Proceedings of the International Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2017.

[MLBH16] Sonal Mahajan, Bailan Li, Pooyan Behnamghader, and William G. J. Halfond. Using visual symptoms for debugging presentation failures in web applications. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 191–201, April 2016.

[MM01] Johannes Martin and Hausi A. Müller. Strategies for migration from C to Java. In *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering (CSMR)*, pages 200–209, 2001.

[MM12] Ali Mesbah and Shabnam Mirshokraie. Automated analysis of CSS rules to support style maintenance. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 408–418, 2012.

[Moa13] Zach Moazeni. csscss, A CSS redundancy analyzer. `http://zmoazeni.github.io/csscss/`, 2013. Accessed: 2017-06-06.

[Moz10] Mozilla Developer Network. Web developer survey research. `https://hacks.mozilla.org/2010/11/its-all-about-web-developers/`, 2010. Accessed: 2017-07-01.

[MP11] Ali Mesbah and Mukul R. Prasad. Automated cross-browser compatibility testing. In *33rd International Conference on Software Engineering (ICSE)*, pages 561–570, May 2011.

[MRS12] Manishankar Mondal, Chanchal K. Roy, and Kevin A. Schneider. An empirical study on clone stability. *Applied Computing Review*, 12(3):20–36, 2012.

[MT16a] Davood Mazinanian and Nikolaos Tsantalis. An Empirical Study on the Use of CSS Preprocessors. In *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 168–178, 2016.

[MT16b] Davood Mazinanian and Nikolaos Tsantalis. Migrating Cascading Style Sheets to Preprocessors by Introducing Mixins. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 672–683, 2016.

[MT16c] Davood Mazinanian and Nikolaos Tsantalis. Tool and dataset for replicating the Empirical Study on the Use Of CSS Preprocessors (SANER'16). `http://dmazinanian.me/conference-papers/saner/2015/12/17/saner16.html`, 2016.

[MT16d] Davood Mazinanian and Nikolaos Tsantalis. Tool and dataset for replicating the study of Migrating CSS to Preprocessors by Introducing Mixins (ASE'16). `http://dmazinanian.me/conference-papers/ase/2016/07/07/ase16.html`, 2016.

[MT17] Davood Mazinanian and Nikolaos Tsantalis. CSSDev: Refactoring duplication in Cascading Style Sheets. In *Proceedings of the 39th International Conference on Software Engineering (ICSE) Companion*, 2017.

[MTM14a] Davood Mazinanian, Nikolaos Tsantalis, and Ali Mesbah. Discovering Refactoring Opportunities in Cascading Style Sheets. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 496–506, 2014.

[MTM14b] Davood Mazinanian, Nikolaos Tsantalis, and Ali Mesbah. Tool and dataset for replicating the study of discovering refactoring opportunities in Cascading Style Sheets (FSE'14). `http://dmazinanian.me/conference-papers/fse/2014/06/16/fse14.html`, 2014.

[MTR07] Tom Mens, Gabriele Taentzer, and Olga Runge. Analysing refactoring dependencies using graph transformation. *Software and System Modeling*, 6(3):269–285, 2007.

[MvDL12] Ali Mesbah, Arie van Deursen, and Stefan Lenselink. Crawling Ajax-based Web Applications through Dynamic Analysis of User Interface State Changes. *ACM Transactions on the Web*, 6(1):3:1–3:30, 2012.

[MZYR13] Tariq Muhammad, Minhaz F. Zibran, Yosuke Yamamoto, and Chanchal K. Roy. Near-miss clone patterns in web applications: An empirical study with industrial systems. In *Proceedings of the 26th IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, pages 1–6, 2013.

[Na15] Dan Na. Transitioning to Sass at Scale. `https://speakerdeck.com/danielna/transitioning-to-sass-at-scale-sassconf-2015-austin-tx`, 2015. Accessed: 2017-07-31.

[Nei17] Tim Neil. css-wrangler: Frozen DOM/Computed Style testing. `https://www.npmjs.com/package/css-wrangler`, 2017. Accessed: 27 June 2017.

[Opd92] William F. Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, IL, USA, 1992.

[PJ15] Christian R. Prause and Matthias Jarke. Gamification for enforcing coding conventions. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 649–660, 2015.

[PVZ16] Leonard Punt, Sjoerd Visscher, and Vadim Zaytsev. The A?B*A Pattern: Undoing Style in CSS and Refactoring Opportunities It Presents. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 67–77, 2016.

[Qua13] Andrew Quan. css-purge. `https://npmjs.org/package/css-purge/`, 2013. Accessed: 2017-06-06.

[QV04] Vincent Quint and Irône Vatton. Techniques for Authoring Complex XML Documents. In *Proceedings of the 2004 ACM Symposium on Document Engineering (DocEng)*, pages 115–123, 2004.

[QV07] Vincent Quint and Irne Vatton. Editing with style. In *Proceedings of the 2007 ACM Symposium on Document Engineering (DocEng)*, pages 151–160, 2007.

[RBS13] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. Software clone detection: A systematic review. In *Information and Software Technology*, volume 55, pages 1165–1199, 2013.

[RC07] Chanchal K. Roy and James R. Cordy. A survey on software clone detection research. Technical report, Queens University, Kingston, Canada, 2007.

[RC08] Chanchal K. Roy and James R. Cordy. NiCad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *Proceedings of the 16th IEEE International Conference on Program Comprehension (ICPC)*, pages 172–181, 2008.

[RCPO13] Shauvik Roy Choudhary, Mukul R. Prasad, and Alessandro Orso. X-PERT: Accurate Identification of Cross-browser Issues in Web Applications. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE)*, pages 702–711, 2013.

[RCVO10] Shauvik Roy Choudhary, Husayn Versee, and Alessandro Orso. WEBDIFF: Automated identification of cross-browser issues in web applications. In *Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM)*, pages 1–10, 2010.

[RJ05] Damith C. Rajapakse and Stan Jarzabek. An investigation of cloning in web applications. In *Proceedings of the 5th International Conference of Web Engineering (ICWE)*, pages 252–262, 2005.

[RJ07] Damith C. Rajapakse and Stan Jarzabek. Using server pages to unify clones in web applications: A trade-off analysis. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, pages 116–126, 2007.

[RLBV10] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Proceedings of the 31th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12, 2010.

[SBD16] M. Serdar Biçer and Banu Diri. Defect prediction for cascading style sheets. *Appl. Soft Comput.*, 49(C):1078–1084, December 2016.

[SCD03] Nikita Synytskyy, James R. Cordy, and Thomas R. Dean. Resolution of static clones in dynamic Web pages. In *Proceedings of the 5th IEEE International Workshop on Web Site Evolution (WSE)*, pages 49–56, 2003.

[SDKS14] Tõnis Saar, Marlon Dumas, Marti Kaljuve, and Nataliia Semenenko. Cross-browser testing in browserbite. In *Proceedings of the 14th International Conference on Web Engineering, (ICWE)*, pages 503–506, 2014.

[SDS13] Nataliia Semenenko, Marlon Dumas, and Tõnis Saar. Browserbite: Accurate cross-browser testing via machine learning over image features. In *Proceedings of the 29th IEEE International Conference on Software Maintenance (ICSM)*, pages 528–531, 2013.

[Ser10] Manuel Serrano. HSS: A Compiler for Cascading Style Sheets. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP)*, pages 109–118, 2010.

[SF10] Alexis Sellier and Dmitry Fadeyev. LESS - The dynamic stylesheet language. `http://lesscss.org/`, 2010. Accessed: 2017-06-06.

[Smi09] David T. Smith. *A Formal Concept Analysis Approach to Association Rule Mining: The Quicl Algorithms*. PhD thesis, Nova Southeastern University, 2009.

[Spi11] Diomidis Spinellis. elytS edoC. *IEEE Software*, 28(2):104–104, March 2011.

[STV16] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. Why we refactor? confessions of github contributors. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 858–870, 2016.

[Sul13] Nicole Sullivan. Object Oriented CSS V2.0.0. `https://github.com/stubbornella/oocss/releases/tag/v2.0.0`, July 2013. Accessed: 27 June 2017.

[TFN+12] Marco Trudel, Carlo A. Furia, Martin Nordio, Bertrand Meyer, and Manuel Oriol. C to O-O Translation: Beyond the Easy Stuff. In *Proceedings of the 19th Working Conference on Reverse Engineering (WCRE)*, pages 19–28, Oct 2012.

[TNM08] Ewan Tempero, James Noble, and Hayden Melton. How Do Java Programs Use Inheritance? An Empirical Study of Inheritance in Java Software. In *Object-Oriented Programming (ECOOP)*, volume 5142 of *Lecture Notes in Computer Science*, pages 667–691. 2008.

[TSK05] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to data mining.* Addison-Wesley, 2005.

[Uni15] United States General Services Administration. CSS coding styleguide. `https://frontend.18f.gov/#css-preprocessors`, 2015. Accessed: 2017-06-06.

[Web16] Web Technology Surveys. Usage of CSS for websites. `http://w3techs.com/technologies/details/ce-css/all/all`, 2016. Accessed: 2016-06-12.

[Wie06] Christoph Wieser. *CSS^{NG}: An Extension of the Cascading Styles Sheets Language (CSS) with Dynamic Document Rendering Features.* Diploma thesis, Institute for Informatics, LMU, 2006.

[Wor17] World Wide Web Consortium. CSS specifications. `http://www.w3.org/Style/CSS/current-work`, 2017. Accessed: 2014-11-09.

[Zhi15] Alex Zhitnitsky. Benchmark: How Misusing Streams Can Make Your Code 5 Times Slower. `http://blog.takipi.com/benchmark-how-java-8-lambdas-and-streams-can-make-your-code-5-times-slower/`, 2015. Accessed: 2017-07-14.

[ZK01] Ying Zou and K. Kontogiannis. A framework for migrating procedural code to object-oriented platforms. In *Proceedings of the Eighth Asia-Pacific Software Engineering Conference (APSEC)*, pages 390–399, 2001.

[ZKJ08] Achim Zeileis, Christian Kleiber, and Simon Jackman. Regression models for count data in R. *Journal of Statistical Software*, 27(8):1–25, 2008.

[ZWSS14] Xiaoyan Zhu, E. James Whitehead, Caitlin Sadowski, and Qinbao Song. An analysis of programming language statement frequency in C, C++, and Java source code. *Software: Practice and Experience*, 45:1479–1495, 2014.