The A?B*A Pattern: Undoing Style in CSS and Refactoring Opportunities it Presents

Leonard Punt
University of Amsterdam, The Netherlands
Q42, The Netherlands

Sjoerd Visscher
Q42, The Netherlands

Vadim Zaytsev
University of Amsterdam, The Netherlands
Raincode, Belgium

***Abstract*—Cascading Style Sheets (CSS) is a language widely used in contemporary web applications for defining the presentation semantics of web documents. Despite its relatively simple syntax, the language has a number of complex features like inheritance, cascading and specificity, which make CSS code challenging to understand and maintain. It has been noted in prior research that CSS code is prone to contain code smells which indicate design weaknesses and maintainability issues.**

In this paper we focus on one of those code smells called undoing style. It happens when a property is set to a value A, then overridden to another value B, possibly multiple times, and then set back to the original value of A. We refer to this pattern as the A?B*A pattern. We propose a technique that detects undoing style in CSS code and recommends refactoring opportunities to eliminate instances of undoing style while preserving the semantics of the web application.

We evaluate our technique on 41 real-world web applications, and outline a proof of correctness for our refactoring. Our findings show that undoing style is quite prominent in CSS code. Additionally, there are many refactorings that can be applied while hardly introducing any errors.

I. Introduction

Cascading Style Sheets (CSS) [2], [5], [17] is a language used for defining the presentation semantics of web documents, like positioning, sizes, colours and fonts. CSS is widely used — 96% of web developers use CSS and over 90% consider it a web standard [28], and it is used on 95% of the websites [39].

Despite the relatively simple syntax of the language, CSS code is not easily understood and maintained [22]. The language has a number of complex features, like inheritance, cascading and specificity [2], [5], [18]. On top of that, established design principles and tool support are missing [21]. Therefore, one of the consequences is that it is not uncommon for CSS code to contain code smells [12]. A code smell is a pattern of code that indicates a weakness in the design. Such a weakness may cause issues in code understanding and maintenance in the long term [10]. In a recent study, Mazinanian et al. [21] found that on average 66% of the style declarations are repeated at least once in a CSS file. Furthermore an 8% size reduction can be achieved by exploring their detected refactoring opportunities. More recently, Gharachorlu [12] showed that CSS smells are widespread in today's websites; 99.8% of the websites (i.e., 499 out 500) analysed in that study, contain at least one type of CSS code smells.

The goal of this work is to detect and come up with semantic preserving refactoring opportunities for the CSS code smell *undoing style*. We have deliberately chosen to scope the project to focus on one smell and investigate it in all details it deserves, rather than providing limited refactoring opportunities for each smell.

The rest of the paper is organised as follows. We briefly introduce CSS and explain its sophistication in section II. In section III we report previous findings on the code smells found in CSS, identify their shortcomings and define the problem we intend to solve. The A?B*A pattern mentioned in the abstract, is defined and elaborated in section IV, followed by the algorithm of its detection in section V. A detailed realistic example is contained in section VI. The experiments we ran as validation are included in section VII, with the discussion of the results found in section VIII. We sketch the proof of correctness of our approach in section IX, revisit related work in section X and conclude the paper with section XI.

II. The CSS Language

An example of a CSS sheet would be a specification that declares that all paragraphs (matching `<p>` tags) should have their inner text centered and printed in red, if hardware permits:

```
p {
    color: red;
    text-align: center;
}
```

The complete grammar of CSS3 [3] is still under development, contains many top and bottom nonterminals, mixes several notations and utterly fails to satisfy any quality requirements of proper grammar engineering [19]. We present a manually derived simplified version of it:

```
StyleSheet  ::= Rule* ;
Rule        ::= Selector "{" Declaration* "}"
              | "@charset" String
              | "@font-face" "{" Declaration* "}"
              | "@import" (URL | String) MediaQueryList?
              | "@media" MediaQueryList "{" Rule* "}"
              | "@page" "{" Declaration* "}" ;
Declaration ::= Property ":" Value "!important"? ";"? ;
```

As we can see, a style sheet is nothing more than a collection of rules, and each rule binds a selector to a collection of property-value declarations. CSS has changed its definition of what constitutes a selector from CSS2 [5] to CSS3 [3], but in contemporary terms a selector is a comma-separated group of combinator-separated sequences of simple selectors [9] (such as `h1, div > h2.header, h3 + img[src][alt]`). We consider simple selectors in more detail, slowly going from the earliest version of CSS to the latest, adding minimal clarifications whenever necessary.

CSS 1 [17]:

- **Type selector** (`X`) selects all `<X>` elements
- **Link pseudo-classes** (`X:link` and `X:visited`)
- **First pseudos** (`X:first-child`, `X::first-line`, `X::first-letter`)
- **Class selector** (`X.C`) selects all `<X>` elements which belong to the `C` class (possibly among other classes)
- **ID selector** (`X#N`) selects one element with the ID `N`
- **Descendant combinator** (`X Y`) selects all elements selected by `Y` that are inside elements selected by `X`.

CSS 2 [5]:

- **Universal selector** (`*`) selects all elements
- **Attribute selectors** (`X[A]`, `X[A=N]`, `X[A~=N]` and `X[A|=N]`) select all elements that have an attribute, or have the exact value of the attribute, or have a value included in a space- or dash-separated list.
- **Action pseudo-classes** (`X:active`, `X:hover`, `X:focus`)
- **Sibling pseudo-elements** (`X::before`, `X::after`)
- **Language pseudo-class** (`X:lang(L)`)
- **Child combinator** (`X > Y`) selects all elements selected by `Y` where the parent is an element selected by `X`.
- **Adjacent sibling combinator** (`X + Y`) selects all elements selected by `Y` that are placed immediately after elements selected by `X`.

CSS 3 [2]:

- **Structural pseudo-classes** (`X:root`, `X:empty`, `X:nth-child(N)`, `X:nth-last-child(N)`, `X:nth-of-type(N)`, `X:nth-last-of-type(N)`, `X:last-child`, `X:first-of-type`, `X:last-of-type`, `X:only-child`, `X:only-of-type`)
- **Attribute selectors** (`X[A*=N]`, `X[A^=N]` and `X[A$=N]`) implement substring selection, starts-with and ends-with comparators
- **Target pseudo-class** (`X:target`)
- **Element state pseudo-classes** (`X:enabled`, `X:disabled`, `X:checked`)
- **Negation pseudo-class** (`X:not(C)`)
- **General sibling combinator** (`X ~ Y`) selects every element selected by `Y` that is preceded by an element selected by `X`.

Having so many different options and multiple ways to combine them clearly presents a maintenance challenge. In practice it leads to codebases bloated with duplicated rules adapted slightly for each use [21], [22]. To complicate matters further, all the properties specified by the style sheet rules, are being assigned to elements of a hierarchical structure, commonly inherited downwards and overwritten by more specific rules.

It is commonplace to find multiple rules apply to one element, either because they are explicitly defined in a style sheet, or because of inheritance, or both. When a property is defined multiple times for an element, the cascading order decides which value is applied. The cascading order is based on the rules' **location** and **specificity**.

The style can be defined in three different locations. The location with the highest preference is inline, when the style is defined on the HTML element itself using the `style` attribute. The second location is internal, the style sheet is embedded inside the HTML document. The least preferred location is external, the style sheet is an external file and is linked to the HTML document.

A selector's specificity [9] is a three-digit number in a high enough base, where the least significant digit is the number of type selectors and pseudo-elements, the second least significant digit is the number of class selectors, attribute selectors and pseudo-classes, and the most significant digit is the number of ID selectors. For example, `ol li` has a specificity score of 002, while `div.main ol li.red` has a score of 023. The universal selector `*` has a default specificity of 000. The most bizarre thing about this standard is that doubling some selectors will have no impact on the matching itself but will change their specificity. So yes, `div#X#X#X#X#X` has a score of 501.

A special `!important` modifier can be used to supersede any specificity calculations and essentially break cascading: any rule marked with `!important` is considered to be more specific than any rule without it. Using `!important` is commonly considered bad practice in web development due to the debugging difficulties it introduces [13].

III. Code Smells in CSS

A *code smell* is a pattern of code that indicates a weakness in the design and a possible cause for future comprehension problems [10]. *Code duplication* is probably the most well-known code smell, and easily portable from programming languages to CSS. Code duplication is actually more prevalent in CSS code compared to procedural and object-oriented code. The main reason is the lack of variables and functions that could be used to build reusable blocks of code. Ways to deal with code duplications have already been proposed [21], and the future versions of CSS may include variables after all.

Another well-known smell is *dead code*, which maps in the CSS technical space to unused selectors and redundant property declarations. In our work we do not focus on dead code, but usually it relies on either dynamic cross-language slicing [25], or on collecting all DOM states by crawling the pages and deleting unused rules or properties [6], [22].

Using CSS directly in HTML or JavaScript is a smell, namely *violation of separation of concerns*. We chose not to focus on this smell, since in our industrial experience

Error / smell	[8]	[40]	[12]	[13]
Parse error		✓	✓	
Value type error		✓	✓	
Box model size	✓		✓	
Proper `display` properties	✓		✓	
Duplicate properties	✓		✓	✓
Empty rules	✓		✓	✓
Unknown properties	✓		✓	
Adjoining classes	✓		✓	
Box-sizing	✓		✓	
Incompatible vendor prefixes	✓		✓	
Insufficient gradient definitions	✓		✓	✓
Negative text-indent	✓		✓	✓
No vendor prefix properties	✓		✓	✓
No fallback colours	✓		✓	✓
Star hack	✓		✓	✓
Underscore hack	✓		✓	✓
Error-prone font-face	✓		✓	
Too many web fonts	✓		✓	
Use of `@import`	✓		✓	✓
Regex-looking selectors	✓		✓	
The universal selector	✓		✓	✓
Unqualified attribute selectors	✓		✓	✓
Units for zero values	✓		✓	✓
Overqualified elements	✓		✓	✓
No shorthand properties	✓		✓	✓
Duplicate background images	✓		✓	✓
Too many floats	✓		✓	
Too many `font-size` declarations	✓		✓	
IDs in selectors	✓		✓	✓
`!important`	✓		✓	✓
`outline:none`	✓		✓	✓
Qualified headings	✓		✓	✓
Multiple definitions of headings	✓		✓	✓

TABLE I
Smells and errors detected in CSS style sheets by linters and checkers: CSS Lint, W3C CSS Validator, CSS Nose and CssCoco.

almost every developer uses external style sheets nowadays. Many websites do contain CSS directly in HTML or JavaScript [26], [39], but this is mainly code injected by frameworks. Refactoring frameworks could be a project on itself, challenging since numerous frameworks, or specific versions of frameworks, are no longer maintained or not open source.

Next there are smells like *too long* rules, *too much* cascading, *high specificity* values and *too general* selectors [12]. The problem with these smells is that there is no standard widely accepted definition of "too much" or "too high". To avoid controversy, we chose not to include these smells in any of our tools (reported here and in [13]).

There exist numerous linters that detect errors and smells, like CCS Lint [8] (and others based on it, like Codacy [7]), CSSNose [12], CssCoco [13] and the W3C CSS Validator [40]. An overview of these errors and smells can be found in Table I, where they are grouped by categories: conformance, possible errors, compatibility, performance, maintainability, duplication, accessibility. CSSNose is the most comprehensive one since it includes both CSS Lint and W3C CSS Validator and does not exclude the controversial smells. The problem with linters is that they check whether the code conforms to certain style guidelines. However, from our experience, many developers do not agree with the style guidelines used. That is why we chose not to include these smells in our tool.

Finally, there is *undoing style*, previously studied [12] but, as we claim, insufficiently so. Furthermore, to the best of our knowledge, refactoring undoing style has not been studied yet.

CSSNose marks every property value that is `0` or `none` as having an "undoing style" smell [12]. This definition produces both false positives and false negatives. The **false positives** are produced because a `none` value for property `display` is one of the standard ways for web developers to hide an element. Additionally, all reset styles are marked as smells by CSSNose. In practice reset styles are used to remove the inconsistencies in presentation defaults between browsers, since all browsers have presentation defaults, but no browsers have the same defaults. These reset styles are commonly used and considered good practice [23], [31]. **False negatives** are produced by CSSNose because only `0` or `none` property values are taken into account. However, a style can be *undone* by any valid value for that property. For example, if the developer first sets the margin to `25px`, next to `50px`, and then back to `25px`, the style is undone as well, and this occurrence should be marked as a code smell.

IV. The A?B*A Pattern

We propose the following definition of undoing style, which closely follows the intuitive perception of it: *if a property is first set to a value A or has an implicit value A, then it is possibly overridden and set to a different value B, possibly multiple times, and subsequently overridden again and set back to the original value A*, then we say that this property follows an A?B*A pattern and smells like undoing style. Our definition does not suffer from the false positives problem described above: setting `display:none` has no A?B*A in it; and if overwriting browsers' defaults is not considered harmful by developers, we can differentiate between *A?A* (undoing the default) and *AA* (undoing self-provided value) patterns. It also does not suffer from the problem with the false negatives, since the A?B*A pattern is checked for all values and properties, not just for `0` or `none`.

The A?B*A pattern matches several flavours of smells, such as *ABA*, *ABCDA* and *AA*. For simplicity we allow ourselves to refer to all of them as "undoing style", but precisely speaking some of them should be given different names, so the name "the A?B*A pattern" is more exact.

We claim that there are no false positives covered by the A?B*A pattern, and all the matched flavours of smells are in fact harmful. The problem with this pattern is twofold. First, more CSS is written in order to achieve less styling. Second, the very nature of CSS is that styles will cascade and inherit from styles defined previously. New rules should only add properties to styles defined before, not undo them. If a style is undoing a cascaded or an inherited style, the style that is cascaded or inherited is applied too early. This pattern may indicate that some developers do not fully comprehend the cascading and inheritance properties of CSS.

In order to detect all occurrences of the A?B*A pattern, the implicit values have to be taken into account. There are three types of implicit values: default values, initial values and inherited values. *Default* values are defined in user agent style sheets, which are default CSS styles defined by the user agent (e.g. browser). *Initial* values are values that are defined as the initial value for a property in the W3C specification. *Inherited* values are values that are inherited. The implicit value of a property is the default value, if it is defined. If there is no default value, then the implicit value is the initial value for non-inheriting properties and the inherited value for inheriting properties. For example, `margin` is a non-inheriting property and has `0` as initial value. If there is no default style, the implicit value for `margin` is `0`. So if a developer sets the value of the property margin first to `10px` and then to `0`, it should be marked as undoing style, since the implicit value for the margin was `0` anyway. Another example: `color` is an inheriting property. If there is no default style, we need to get the inherited style in order to determine the implicit style for `color`. The inherited style can be retrieved by getting the parent's value for the property.

Pseudos form an exception to the definition and are allowed to undo style. A pseudo-element defines a special static state of an element and a pseudo-class is used for dynamic styles. Usually it is more convenient to reset the style in this special state than to add the style to all elements, except to the element in that state. An example is if all items of a list have to have some style, except the first. One way to achieve this is by assigning a class to every element in the list, except the first one, since many pseudos have no natural inverse. Next a CSS rule can be created, using the class selector and the appropriate style. However, it is more convenient to assign the class to the list element itself, instead to all its child elements. Next the style can be reset for the first element using the pseudo-elements `:first-child`. The pseudo-elements in which this pattern is allowed are `:first-child`, `:last-child`, `:nth-child` and `:nth-last-child`.

An example with the A?B*A pattern is shown below.

```
.input-field > *  { border: 0 none; float: left; }
.text-field input {
    float: none;
    padding-left: 48px;
    width: 100%; }
```

```
<div class="input-field text-field">
  <label for="fld1">...</label>
  <input type="text" name="fld1" id="fld1" class="rel">
</div>
<div class="input-field text-field">
  <label for="fld2">...</label>
  <input type="text" name="fld2" id="fld2" class="rel">
</div>
<div class="input-field text-field">
  <label for="fld3">...</label>
  <input type="text" name="fld3" id="fld3" class="rel">
</div>
```

It is a simplified version of a smell we found on one of the websites the company of the first two authors developed. The first rule sets the `float` property to `left` for all elements inside an element with class `input-field`. The second rule sets the `float` property to `none` for all `input` elements inside an element with class `text-field`. The second rule overrides the first rule, since it has a higher specificity. The implicit value for `float` is `none`, this implicit value is overridden by both rules. These rules contain the A?B*A pattern, where the first A? is the implicit style, B is the first rule and the second A is the second rule. This code snippet is a nice example of a style that is applied too early. In this case it is better to attach the style `float:left` only to the `label` elements inside an element with class `input-field`. Then the reset to `none` would not be necessary.

V. Detection

In order to detect the undoing style smell, we need both the CSS (normalised such that `0` and `0px` or `#000` and `#000000` match) and the HTML. With the information from both we can figure out which style properties match a certain HTML element. Then we can easily check whether a style property is defined multiple times for this HTML element. Next we need to know the cascading order of properties. This is needed in order to determine which property overrides the other ones. In order to determine the cascading order, we first check if a property is defined as `!important`, if not, we calculate and compare the specificity of the selectors. If the selectors have the same specificity, we need their location information to decide which style gets precedence over the other.

Once we know if a property is defined multiple times for an HTML element, and we know the cascading order, we can check if there are A?B*A patterns for that property and HTML element. First we try to find the longest A?B*A pattern possible. All other patterns that are embedded in the longer pattern can be safely ignored, since they will be removed if the longest pattern is refactored. An example is the pattern $ABCBA$, which contains four A?B*A patterns: $ABCBA$, BCB, $ABBA$ and ACA. However, we prefer $ABCBA$ and let the rest be subsumed. Patterns are allowed to overlap, as long as they do not overlap completely (so $ABAB$ does contain two valid A?B*A occurrences).

All the detected undoing style smells are refactoring opportunities. The developer should decide whether to apply a refactoring or not. In section IX we will formally outline a refactoring for the undoing style smells that adhere to certain preconditions. The algorithm used for detecting the A?B*A pattern, can be found on Listing 1. An executable tool is described as a proper artefact [29] and is publicly available. For obvious reasons we had to choose for *dynamic analysis* and not static analysis — realistic contemporary web applications are always dynamic in some way. However, our tool still has limitations since it is run on the client side (in the browser) and is therefore limited to whatever is accessible from the browser: parsed style sheets (internal and external), Document Object Model (DOM) elements, etc., but not document templates. We leave it up to the developer who uses our tool, to provide multiple DOM instances which is required when undoing style manifests itself in DOM updates — they are usually collected with a crawler or leveraged from the test suite, depending on the project.

For practical reasons, our tool ignores `@media` rules: they add a circumstantial element to the mix which piles a level of complexity on the task of optimising CSS style sheets (e.g., undoing style for a very specific screen size can be more maintainable than explicitly introducing several `@media` clauses); treating `@media` clauses similarly to the rest requires tweaking of the algorithm but more importantly yields more complex rules after refactoring.

Any A?B*A pattern occurrence is considered a *refactoring opportunity* [21] if it meets two preconditions: (1) the most specific part of the selector of the enclosed rule should be an ID or a class selector; (2) the elements that the reset rule applies to, should be a subset of the elements the initial rule applies to. The reason for having these particular preconditions will become apparent from section IX where we sketch a proof for preserving the semantics of CSS if the refactoring takes place.

Acting on a refactoring opportunity involves three steps. The **first step** is to move the property from the enclosed CSS rule to a new CSS rule. Note that if the property is defined multiple times in a rule, only the last definition should be moved. All the other definitions should be removed. In order to preserve semantics, there are two requirements for the new rule: (1) it needs to be defined directly beneath the enclosed rule; (2) it has to have the same or a one class point higher specificity than the enclosed rule's selector. These two requirements are needed because the location and the specificity could have influence on the cascading order. By defining the new rule directly beneath the original rule and keeping the specificity the same or changing it as little as possible, we know the new rule overrides and is overridden by the same rules as the original rule. The only exception is that the new rule now overrides the original rule, but this is not an issue since the property of the new rule does not exist in the original rule anymore, since it is moved. The selector of this new rule is almost the same as the selector of the original rule: we only update the most specific part. If the most specific part is a class selector, we replace this class selector by a new class selector. If the most specific part is an ID selector, we append a class selector to the ID selector. In both cases we satisfy the requirement that the new rule has to have the same or a one class point higher specificity. Note that the most specific part cannot be an element selector, because of the preconditions.

The **second step** is to take the difference between all nodes of the enclosed rule and the nodes of the reset rule. The new class (the most specific part of the new rule's selector) is added to these elements. The **third step** is to remove the undoing style property from the reset rule. If the rule is empty after removing the property, it is removed from the style sheet. Note that we cannot remove the usages from the HTML documents, since they might be used by client-side scripts. Because we moved a property to a new rule, we need to update all detections that have a reference to that enclosed rule and property somewhere in their A?B*A pattern. That is the final step.

After the refactoring the original and the updated programs are checked for all DOM states provided by the developer (see above) and requests computed styles (values for all the properties) for each element. If these computed styles have not changed, then the semantics of CSS has been preserved under transformation.

VI. Example

Consider the following CSS and HTML code:

```
a { text-decoration-line: none; }
.cms .link { text-decoration-line: underline; }
.cms .link.more { text-decoration-line: none; }
```

```
<div class="cms">
  <a href="#" class="link">...</a>
  <a href="#" class="link more">...</a>
</div>
```

The example has ABAB with underline as A and no decoration as B, with the implicit A-rule being the browser's default for `<a>`. We will focus on the BAB part with the initial rule is the first B-rule, the enclosed rule is the second A-rule, and the reset rule is the third A-rule. The nodes that apply to the initial rule are both `<a>` elements, these elements apply to the enclosed rule as well. The last element is the only element that applies to the reset rule. Both preconditions hold for this BAB. The most specific selector part of the enclosed rule is a class selector. The elements that apply to the reset rule, are a subset of the elements that apply to the initial rule. The refactored code:

```
a { text-decoration-line: none; }
.cms .class1 { text-decoration-line: underline; }
```

```
<div class="cms">
  <a href="#" class="link class1">...</a>
  <a href="#" class="link more">...</a>
</div>
```

We see that the rule with selector `.class1` is the new rule. This rule contains the declaration `text-decoration-line: underline`, which is moved from the enclosed rule to the new rule. The property `text-decoration-line` is removed from the reset rule. Both the enclosed and the reset rule are removed since they were empty after the refactoring was applied. The class `.class1` is added to each element that matched to the enclosed rule but not to the reset rule, which is the first `<a>` element in this example. This refactoring preserves the semantics, removes the undoing style smell and reduces the number of CSS rules.

VII. Empirical Evaluation

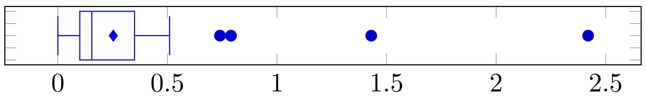
For empirical evaluation of our tool we reuse the data set from the study of Mazinanian, Tsantalis and Mesbah [21] which they graciously offer online at http://users.encs.

concordia.ca/~d_mazina/papers/FSE'14/. This data set includes 38 randomly selected as well as author selected online web applications, all highly visited and developed by web market leaders. All adjustments we have allowed ourselves, are described as a proper artefact [30]. In short:

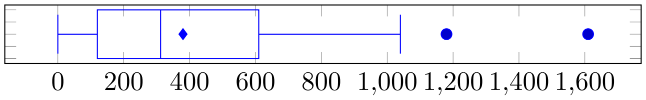
- Renamed two websites: `apple.ca` to `apple.com` and MountainEquip to MEC to fit our location and their updated names.
- Removed cookie-based redirect from the Gmail dataset since it was detecting the lack of "right" cookies on our machine and redirecting us away so no DOM states could be found.
- Added a newly fetched Gmail dataset to counter the possibility that our intervention had some negative effect on the detection. The original Gmail dataset was kept in the experiment along with it.
- Removed three style sheets that were empty in the FSE dataset: two of them return Forbidden errors (from GlobalTVBC and SyncCreative) and one File Not Found (from Apple.com). It is unknown whether the errors manifested themselves during the original collection of data as well.
- Refetched four style sheets that were empty in the FSE dataset for unknown reasons (two from About.com and two from Alibaba).
- Excluded five unused style sheets from the ProToolsExpress dataset and one duplicate style sheet from SyncCreative.
- Added two projects developed by Q42, the company with which the first two authors are affiliated: http://9292.nl and http://Rijksmuseum.nl. They are fairly dynamic and well-known within the Dutch internet.

The experiments were executed in a fairly transparent way, following the theory and design we explained in section V: we loaded the application and the DOM states, fixed references to external style sheets, detected A?B*A patterns, filtered out non-opportunities that failed to satisfy our preconditions, refactored the rest and checked for semantics preservation

A. RQ1: What is the extent of undoing style in CSS?



Plot 1. Ratio of detected undoing style smells to CSS rules.

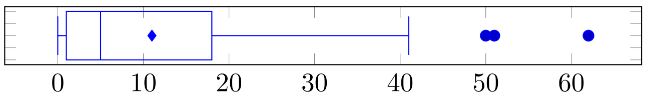


Plot 2. Total number of detected undoing style smells.

Our result show that undoing style is prevalent in CSS code. Plot 1 displays the ratios between the number of detections and the number of rules in the analysed style sheets. The median value for the ratio is 16% while the average (diamond on the plot) is 25%, dots indicate outliers. The most extreme outlier has a ratio of 242% and caused by a client-side script that copies an inline style sheet, causing many elements to exhibit the AA pattern. Plot 2 displays the number of undoing style occasions detected in the analysed CSS code. On average, we were able to detect 380 occurrences of undoing style. The outliers here are subjects with a lot of CSS rules, so the A?B*A pattern cases are proportional.

49% of the opportunities are of type ABA, 36% of type AA, 12% of type AB^2A, 3% of type AB^3A, only 0.75% of type AB^4A and all other cases together total 0.25%. Additionally we looked at the initial CSS rule, the rule that initiates the A?B*A pattern, and found that in 63% of the cases the initiator is an implicit value, in 18% an element selector, in 9% a class selector, but other options also often get to 1% or 2% each. When looking further into the implicit values, we found that in 7% of the implicit values are inherited values, the other values are either default or initial values.

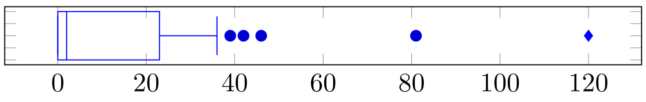
B. RQ2. What refactorings can result from detected opportunities?



Plot 3. Number of applied refactorings.

Plot 3 shows the number of refactorings we have applied on the CSS files. A refactoring is a refactoring opportunity that is applied, resulting in the removal of the smell. (Since we have acted on all refactoring opportunities, the number of refactorings is equal to the number of refactoring opportunities). As can be observed, our approach was able to apply 11 refactorings on average. The maximum number of refactorings applied on one state of an application was 62. The outliers on this boxplot are subjects with a lot of detected undoing style smells, which explains the higher number of applied refactorings. The opportunities that are not refactored can be used as input when developing more advanced refactorings.

C. RQ3. Do the proposed refactorings preserve semantics?



Plot 4. Number of changes to semantics. Outliers not shown on the plot: 227, 256, 345, 400, 627, 792, 1134, 3869.

A semantic change is a change for the value of a property at a node caused by a refactoring. These changes are detected by checking that the computed styles of all elements have not changed after the refactoring is applied. Note, however, that due to the dynamic nature of many web applications, client-side scripts might update the properties of a node during the refactoring. So it is quite likely that changes are detected that are *not* introduced by the refactoring. In total 8789 semantic changes were detected in our experiments. For all these changes, only 38 are introduced because a refactoring was applied, the rest were classified as false positives.

Plot 4 shows that the median is 2 semantic changes and the mean is 120. Indeed, the data contains extreme outliers, 44% of the all the semantics changes are in *one* subject. All the errors in that subject are caused by updates from a client-side script. The reasons we know most of the errors are not introduced as a result of applying a refactoring, are:

- There are no refactoring opportunities for the subject.
- The CSS properties that contained errors are not CSS properties that were refactored.
- The selectors of the refactored CSS rules do not match with the nodes that contain errors.
- The reported errors are false negatives.
- A client-side script includes the original style sheet, which results in both the refactored and the original style sheets being included.

From the 38 errors that are introduced by a refactoring, 36 are *because an external style sheet was not fetched, but was loaded*. This external style sheet interfered with a second external style sheet, which was fetched. Because undoing style is only detected in fetched style sheets, the pattern was only detected in the second style sheet, ignoring the rules from the first style sheet. Therefore a too short A?B*A pattern was detected and refactored. The other two errors were due to a specificity calculation bug in our tool manifesting when both rules had the `!important` annotation — they disappeared when the bug was fixed.

VIII. Discussion

Our experiment shows that undoing style is prevalent in the CSS code of today's web applications. The pattern ABA is the most prevalent one, followed by the pattern AA. Most of the undoing style patterns start with an implicit value or an element selector. The number of patterns that start with an implicit value, as well as the types of popular patterns, indicate that developers either do not know or do not trust the implicit values, and end up overriding them.

The number of patterns that start with an element selector indicates that developers apply styles too broadly. Our study shows that in many cases these styles have to be undone. We argue that it is bad practice to apply styles too broadly, because the very nature of CSS is that styles will cascade and inherit from styles defined previously. New rules should only add properties to styles defined before, not undo them. The results of the evaluation of our tool also show that refactorings are applicable to a subset of the A?B*A pattern, while mostly preserving the semantics.

To be sure whether a refactoring can be applied safely, we need to know all possible DOM states of the web application. In our current implementation, we did not provide a functionality to capture all possible DOM states and expect the user to provide them. There are several options to fix this limitation. One is to add a crawler that captures DOM states, such as based on Cilla/Crawljax [38]. Another option is to attach our tool to an existing test suite of the web application. A third option is to make use of the visual style guide of a web application, if it exists. Finally, it should also be feasible to generate the possible DOM states from the source code of the application.

Other limitations of our tool are: *client-side DOM updates* which cause quite some false positives since we do not take these into account when checking whether the semantics are preserved; limiting scope to files that are served from the same origin which corresponds to the so called *same-origin policy* that all modern browsers adhere to [32], [37], [41]; implementing equality literally and not semantically (we use normalised values served by the browser and do not implement their adjustable equivalence, so `black` and `#000000` are treated as different).

With respect to threats to validity, we inherit them from the study by Mazinanian et al. since we rely on (mostly) the same dataset [21] which in turn extended the dataset by Mesbah and Mirshokraie [22]. There are some threats to internal validity: the DOM states collected from each web application may be insufficient to decide whether a detection is refactorable or not, since for some dynamic web applications the number of DOM states is practically infinite. Missing DOM states could also make some of the applied refactorings to be semantic revising for this particular set of unvisited DOM states. The attempts to avoid selection bias included selecting 14 subjects from the list of websites analysed one case study [22], 24 subjects from another one [21] and two developed by an affiliated company.

IX. Refactoring Correctness

In this section we are going to outline the proof of correctness of our refactoring. A solid, fully formalised proof is outside the scope and format, but can be considered future work. We are going to use set theory and logic to prove that the refactoring described in section V preserves all semantics.

The refactoring comprises two steps: (1) a CSS property is moved to a new rule, (2) a CSS property is removed from an existing rule. We are going to prove that both steps are correct, that is, in both steps all semantics are preserved. We argue that the correctness of the refactoring is implied by the correctness of the two steps.

Given an ABA pattern for a style property s, let us define the following:

- $\mathcal{R}$ is the set of all CSS rules.
- I is the set of all HTML elements the initial CSS rule applies to.
- E is the set of all HTML elements the enclosed CSS rule applies to.
- R is the set of all HTML elements the reset CSS rule applies to.

$\mathcal{R}$ is a totally ordered set (specified by the order of rules in the file). The properties of the relation are described as the cascading order. The cascading order is used to determine for each style property, which CSS rule is applied to an HTML element. The rule that is applied is the CSS rule that is highest in $\mathcal{R}$. In order for an ABA pattern to be refactorable, two preconditions have to hold:

- The most specific part of the selector of the enclosed rule should be an ID or a class selector.
- $R \subset I$

The first precondition is important, because in order to create a new selector from the enclosed rule's selector, we need to add or replace a class selector. Adding a class selector has influence on the cascading order. As explained in section II, the specificity of a selector is a three-digit number with a high enough base. The specificity is one of the properties that is used to determine the cascading order. If we need to create such a new selector, there are three possibilities: the most specific part of the rule could be an ID selector, a class selector or an element selector.

If the enclosed rule's selector has a class selector as most specific part, we can easily replace this class selector with a new class selector. In this case, the specificity score stays the same.

If the enclosed rule's selector has an ID selector as the most specific part, we can easily add a class selector. The middle digit in the specificity score will then be one higher, but there is no other value possible between the specificity of the enclosed CSS rule and the specificity of the new CSS rule. So the new rule will not override any CSS rules that the enclosed rule does not override.

However, if the enclosed rule's selector has an element selector as most specific part, we cannot add a class selector. This is best explained by the following example: assume we have two selectors: `p` and `div p`. The specificity of these selectors is respectively: 001 and 002. Therefore, `div p` has precedence over `p`. Now assume `p` is the selector of the enclosed CSS rule. If we have to make a new selector, it will be `p.someClass`. This new selector has specificity 011 and therefore has precedence over `div p`, which is incorrect. Ergo, the most specific part of the selector of the enclosed rule should be an ID or a class selector.

The first step in moving the CSS property is to add a new class c to every element in the set $E \setminus R$. The second step is to move the style property s from the enclosed CSS rule, to a new CSS rule that has class c as most specific part. Because of the first precondition, this new CSS rule comes directly above the enclosed CSS rule in the set $\mathcal{R}$. Now we have E' as the set of all HTML elements the new CSS rule applies to, such that $E' = E \setminus R$. Since $E = E' \cup R$, it means that if the semantics are preserved in E' and R, the semantics in E are preserved.

The semantics in E' are preserved, because the new CSS rule comes directly above the enclosed CSS rule in $\mathcal{R}$. Therefore the new CSS rule, which has property s, overrides the enclosed CSS rule. So for all HTML elements were the enclosed CSS rule was the highest rule in $\mathcal{R}$, the new CSS rule will be the highest precedence rule now. Therefore, the correct value for property s is applied. Note that the new rule does not have to be the highest precedence rule in $\mathcal{R}$ for any HTML element. However, because the new CSS rule comes directly above the enclosed CSS rule in $\mathcal{R}$, the order relation between any rule in $\mathcal{R}$ is the same for the enclosed CSS rule and the new CSS rule. In other words, the new CSS rule will not override any other rule than the enclosed CSS rule did. Therefore, the correct value for property s will be applied.

The semantics in R are preserved as well, because from the definition of the ABA pattern follows that the reset CSS rule is higher in $\mathcal{R}$ than the enclosed CSS rule. And since the new CSS rule comes directly above the enclosed CSS rule, the reset CSS rule is also higher than the new CSS rule. So the new CSS rule will not override the reset CSS rule, and thus the correct value for property s will still be applied.

Thus we have shown that moving the CSS property s to a new rule has no influence on the value of s for any HTML element. Furthermore, we did not touch any other CSS property than the CSS property s, so the value of any other property has not changed. Therefore we claim that all semantics are preserved in the first step of our refactoring.

From the second precondition and first step of the refactoring, we have $R \subset (I \setminus E')$. In other words, the enclosed CSS rule does not apply to R anymore. Furthermore, from the ABA pattern follows that the value for property p is the same in the initial CSS rule and the reset CSS rule. Therefore, property p can be removed from the reset CSS rule, because then the initial CSS rule becomes the highest rule in $\mathcal{R}$ for all elements in R. Because of the second precondition, every element in R will have an A?B*A pattern. However, the number of Bs can differ. It is important that all Bs, or enclosed CSS rules, are refactored, for all elements in R. Only then the initial CSS rule will be the highest rule in $\mathcal{R}$ for all elements in R, if the reset CSS rule is removed. We showed that removing the CSS property s from the reset rule has no influence on the value of s for any HTML element. Furthermore, we did not touch any other CSS property than the CSS property s, so the value of any other property has not changed. Therefore we claim that all semantics are preserved in the second step of our refactoring as well. Quod erat demonstrandum.

X. Related Work

There is a wide range of papers discussing refactoring in general, it would be impossible to name them all. Fowler and Beck [10] demonstrate how software practitioners can realise significant benefits to the structural integrity and performance of existing software programs using a collection of techniques. These practices are referred to as refactoring. The book is ancient by software engineering standards, but recent endeavours have demonstrated that refactoring indeed improves maintainability [16], [36], even though it is not a silver bullet and does not eliminate the need for systematic editing [14] and can even increase power consumption [27]. Single refactorings are also known to cause short term productivity and quality problems [1], [35]. Refactoring can be inferred automatically from continuous changes [24], even though in practice more than half of the refactorings is performed manually.

Several researchers have developed techniques for detecting and ranking refactoring opportunities. Bavota et al. [4] claim that an additional source of information for identifying refactoring opportunities is team development activity. This new refactoring dimension can be complemented with other approaches (such as ours) to build better refactoring recommendation tools. Silva et al. [33] recommend to extract methods that hide structural dependencies that are rarely used by the remaining statements in the original method. Steidl and Eder [34] propose a way to prioritize among a large number of quality defects. Their approach recommends to remove quality defects, exemplary code clones and long methods, which are easy to refactor and, thus, provides developers a first starting point for quality improvement. Mayer and Schroeder [20] report on an approach for automatically identifying multi-language relevant artefacts, finding references among artefacts in different languages, and refactoring them.

There are several papers discussing code smells in CSS. Mazinanian et al. [21] define three types of duplication in CSS and present a technique for detecting and refactoring those duplications. These refactorings preserve the semantics of the web application. Gharachorlu [12] proposes an automated technique to detect 26 CSS smells and errors. Based on the findings of a large empirical study, he proposes a model that is capable of predicting the total number of CSS code smells in any given website. Mesbah and Mirshokraie [22] propose a technique that automatically checks CSS code against different DOM states and their elements to infer an understanding of the runtime relationship between the two. Next it checks for unmatched and ineffective rules, overridden declaration properties, and undefined class values.

Bosch et al. [6] present a prototype of a *static* CSS semantic analyser and optimiser. The prototype is capable of automatically detecting and removing redundant property declarations and rules. They guarantee that the rendering in the browser will not be affected, for any possible document that might use the CSS. Genevès et al. [11] present a tool based on tree logics. The tool is capable of statically detecting a wide range of errors (such as empty CSS selectors and semantically equivalent selectors), as well as proving properties related to sets of documents (such as coverage of styling information), in the presence or absence of schema information. Nguyen et al. [26] introduce a tool to detect embedded code smells. The tool first detects the smells in the generated code and next locates them in the server-side code.

Keller and Nussbaumer [15] introduce a CSS quality property: abstractness factor. They argue that a high abstractness factor represents a high maintainability and reusability of the style sheet as well as the HTML document. In future work we can use this and similar metrics to check whether our refactorings not only preserve semantics, but also improve the design [10].

XI. Conclusion

We have summarised some of the core features of CSS and code smells that it was found to have. We have developed two techniques: (1) for detection of undoing style in CSS; (2) for refactoring detection results that conform to certain patterns. We have shown that if the detections adhere to these preconditions, the semantics will be preserved. To the best of our knowledge, our work is the first to provide refactoring opportunities with respect to undoing style in CSS. We have used the tool [29] to perform a validating experiment on 41 real-world web applications [30] and found that:

- Undoing style is omnipresent in CSS; the ratio between the number of detections and the number of rules is 25%; we were able to detect 2060 occurrences of undoing style on average. With 49%, the *ABA* pattern was the most prevalent type of undoing style.
- There are many instances that can be refactored while preserving the presentation semantics: 11 per webapp on average, with at most 62 refactorings applied on one state of an application.
- There are barely any errors introduced by our refactorings. From the 8789 detected changes to the semantics, for all subjects in total, only 38 were introduced by a refactoring and even those are explainable by the implementation details.

We have also provided a formalisation of the refactoring steps, and argued that the correctness of the refactoring is implied by the correctness of the two steps. The question of propagating refactoring impact to scripts that depend on style classes, remains future work and possibly requires dynamic slicing [25].

The main contributions of the paper are: investigating the *undoing style* CSS code smell, which we refine significantly with respect to prior work; developing an open source tool to detect refactoring opportunities conforming to our A?B*A pattern and to refactor a subset of opportunities while preserving the semantics; evaluating the proof of concept by not only deploying it in the industrial setting, but also conducting an experiment based on a (corrected version of a) previously created dataset of 41 real-world web applications to find the extent of undoing styles, the number of refactorings that can be applied and the number of errors introduced by the refactorings; and sketching a proof of correctness for our refactoring.

http://leonardpunt.github.io/masterproject is a publicly accessible website containing the tool, the dataset and the experimental data.

```
/* Params:
   - rules: all the style rules for an element, without the rules where the
            undoing style pattern is allowed (like certain pseudo-classes). */
function filterUndoingStyles(rules) {
  var possibleUndoingStyles, undoingStyles = [];

  rules.forEach(function(rule) {
    rule.forEach(function(declaration) {
      // Get all rules that have a declaration for this property
      var rulesWithDeclaration = getRulesWithDeclaration(declaration.property, rules);

      // Sort rules on cascading order
      rulesWithDeclaration.sort(function(rule1, rule2) {
        return _compareCascadingOrder(rule2, rule1);
      });

      if (rulesWithDeclaration.length > 1) {
        possibleUndoingStyles.push({ rules: rulesWithDeclaration });
      }
    });
  });

  possibleUndoingStyles.forEach(function(possibleUndoingStyle) {
    // Detect the A-B*-A patterns for this declaration
    var detections = detectABAs(overridingDeclaration);
    undoingStyles.concat(detections);
  });

  return undoingStyles;
}

var detectABAs = function(overridingDeclaration) {
  // highestA is the A with the highest specificity
  var lowestA, highestA;
  var length = overridingDeclaration.rules.length;
  var detections = [];

  for (var indexhighestA = 0; indexhighestA < length - 1; indexhighestA++) {
    highestA = overridingDeclaration.rules[indexhighestA];
    for (var indexlowestA = length - 1; indexlowestA > indexhighestA; indexlowestA--) {
      lowestA = overridingDeclaration.rules[indexlowestA];

      if (lowestA.declaration.value === highestA.declaration.value) {
        // Check if this detection is a part of a previous detection, if so: ignore
        if (!isPartOfPreviousDetection(indexlowestA, indexhighestA, detections)) {
          detections.push({
            initialRule: getInitialRule(lowestA, overridingDeclaration),
            enclosedRules: getEnclosedRules(lowestA, highestA, overridingDeclaration),
            resetRule: getResetRule(lowestA, overridingDeclaration)
          });
          break;
        }
      }
    }
  }

  return detections;
};
```

Listing 1. The algorithm to detect the A?B*A pattern, which was described in section V. An executable version is also available from http://leonardpunt.github.io/masterproject/tool.zip.

References

[1] E. Ammerlaan, W. Veninga, and A. Zaidman, "Old Habits Die Hard: Why Refactoring for Understandability Does not Give Immediate Benefits," in *SANER*, Y.-G. Guéhéneuc, B. Adams, and A. Serebrenik, Eds. IEEE, 2015, pp. 504–507.

[2] T. Atkins Jr., E. J. Etemad, and F. Rivoal, "Cascading Style Sheets (CSS) Snapshot 2015," *W3C Working Group Note*, Oct. 2015, http://www.w3.org/TR/2015/NOTE-css-2015-20151013/.

[3] T. Atkins Jr. and S. Sapin, "CSS Syntax Module Level 3," *W3C Candidate Recommendation*, Feb. 2014, https://www.w3.org/TR/2014/CR-css-syntax-3-20140220/.

[4] G. Bavota, S. Panichella, N. Tsantalis, M. D. Penta, R. Oliveto, and G. Canfora, "Recommending Refactorings Based on Team Co-maintenance Patterns," in *ASE*. ACM, 2014, pp. 337–342.

[5] B. Bos, T. Çelik, I. Hickson, H. W. Lie, C. Lilley, and I. Jacobs, "Cascading Style Sheets Level 2 Revision 2 (CSS 2.2) Specification," *W3C Editors' Draft*, Mar. 2016, http://www.w3.org/TR/2016/ED-CSS22-20160329/.

[6] M. Bosch, P. Genevès, and N. Layaïda, "Automated Refactoring for Size Reduction of CSS Style Sheets," in *Proceedings of the 14th Symposium on Document Engineering (DocEng)*, S. J. Simske and S. Rönnau, Eds. ACM, 2014, pp. 13–16.

[7] Codacy, "Patterns list," https://www.codacy.com/patterns.

[8] CSSLint, "Rules," https://github.com/CSSLint/csslint/wiki/Rules.

[9] T. Çelik, E. J. Etemad, D. Glazman, I. Hickson, P. Linss, and J. Williams, "Selectors level 3," *W3C Recommendation*, Sep. 2011, https://www.w3.org/TR/selectors/.

[10] M. Fowler and K. Beck, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[11] P. Genevès, N. Layaïda, and V. Quint, "On the Analysis of Cascading Style Sheets," in *Proceedings of the 21st World Wide Web Conference (WWW)*, A. Mille, F. L. Gandon, J. Misselis, M. Rabinovich, and S. Staab, Eds. ACM, 2012, pp. 809–818.

[12] G. Gharachorlu, "Code Smells in Cascading Style Sheets: An Empirical Study and a Predictive Model." Master's thesis, University of British Columbia, Canada, 2014.

[13] B. Goncharenko and V. Zaytsev, "Language Design and Implementation for the Domain of Coding Conventions," 2016, submitted to SLE 2016, pending notification.

[14] L. Hua, M. Kim, and K. S. McKinley, "Does Automated Refactoring Obviate Systematic Editing?" in *Proceedings of the 37th International Conference on Software Engineering (ICSE), Volume 1*. IEEE, 2015, pp. 392–402.

[15] M. Keller and M. Nussbaumer, "CSS Code Quality: A Metric for Abstractness; Or Why Humans Beat Machines in CSS Coding," in *Proceedings of the Seventh International Conference on the Quality of Information and Communications Technology (QUATIC)*, F. Brito e Abreu, J. P. Faria, and R. J. Machado, Eds. IEEE Computer Society, 2010, pp. 116–121.

[16] M. Källén, S. Holmgren, and E. Þóra Hvannberg, "Impact of Code Refactoring Using Object-Oriented Methodology on a Scientific Computing Application," in *SCAM*. IEEE Computer Society, 2014, pp. 125–134.

[17] H. W. Lie and B. Bos, "Cascading Style Sheets, Level 1," *W3C Recommendation*, Apr. 2008, http://www.w3.org/TR/2008/REC-CSS1-20080411.

[18] H. W. Lie, "Cascading Style Sheets," Ph.D. dissertation, University of Oslo, Norway, 2005.

[19] R. Lämmel and V. Zaytsev, "Recovering Grammar Relationships for the Java Language Specification," *Software Quality Journal (SQJ); Section on Source Code Analysis and Manipulation*, vol. 19, no. 2, pp. 333–378, Mar. 2011.

[20] P. Mayer and A. Schroeder, "Automated Multi-Language Artifact Binding and Rename Refactoring between Java and DSLs Used by Java Frameworks," in *Proceedings of the 28th European Conference on Object-Oriented Programming (ECOOP)*, ser. LNCS, R. Jones, Ed., vol. 8586. Springer, 2014, pp. 437–462.

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

[22] A. Mesbah and S. Mirshokraie, "Automated Analysis of CSS Rules to Support Style Maintenance," in *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, M. Glinz, G. C. Murphy, and M. Pezzè, Eds. IEEE, 2012, pp. 408–418.

[23] E. A. Meyer, "Reset reasoning," http://meyerweb.com/eric/thoughts/2007/04/18/reset-reasoning/, 2007.

[24] S. Negara, N. Chen, M. Vakilian, R. E. Johnson, and D. Dig, "A Comparative Study of Manual and Automated Refactorings," in *Proceedings of the 27th European Conference on Object-Oriented Programming (ECOOP)*, ser. LNCS, G. Castagna, Ed., vol. 7920. Springer, 2013, pp. 552–576.

[25] H. V. Nguyen, C. Kästner, and T. N. Nguyen, "Cross-language Program Slicing for Dynamic Web Applications," in *ESEC/SIGSOFT FSE*. ACM, 2015, pp. 369–380.

[26] H. V. Nguyen, H. A. Nguyen, T. T. Nguyen, A. T. Nguyen, and T. N. Nguyen, "Detection of Embedded Code Smells in Dynamic Web Applications," in *Proceedings of the 27th International Conference on Automated Software Engineering (ASE)*. ACM, 2012, pp. 282–285.

[27] J. J. Park, J.-E. Hong, and S.-H. Lee, "Investigation for Software Power Consumption of Code Refactoring Techniques," in *Proceedings of the 26th International Conference on Software Engineering and Knowledge Engineering (SEKE)*. Knowledge Systems Institute Graduate School, 2014, pp. 717–722.

[28] J. Patel, M. Wirthart *et al.*, "Web Developers and the Open Web Survey," in *Mozilla Developer Network*, 2010, https://hacks.mozilla.org/2010/11/its-all-about-web-developers.

[29] L. Punt, S. Visscher, and V. Zaytsev, "A Tool for Detecting and Refactoring the A?B*A Pattern in CSS," in *ICSME Artefact*, 2016.

[30] ——, "Experimental Data for the A?B*A Pattern in CSS: Inputs and Outputs," in *ICSME Artefact*, 2016.

[31] M. Rand-Hendriksen, "Why a CSS Reset should be at the core of your stylesheet," http://mor10.com/why-a-css-reset-should-be-at-the-core-of-your-stylesheet, 2009.

[32] J. Ruderman *et al.*, "Same-origin policy," in *Mozilla Developer Network*, 2005, https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy.

[33] D. Silva, R. Terra, and M. T. Valente, "Recommending Automated Extract Method Refactorings," in *Proceedings of the 22nd International Conference on Program Comprehension (ICPC)*, C. K. Roy, A. Begel, and L. Moonen, Eds. ACM, 2014, pp. 146–156.

[34] D. Steidl and S. Eder, "Prioritizing Maintainability Defects Based on Refactoring Recommendations," in *Proceedings of the 22nd International Conference on Program Comprehension (ICPC)*, C. K. Roy, A. Begel, and L. Moonen, Eds. ACM, 2014, pp. 168–176.

[35] G. Szoke, G. Antal, C. Nagy, R. Ferenc, and T. Gyimóthy, "Bulk Fixing Coding Issues and Its Effects on Software Quality: Is It Worth Refactoring?" in *Proceedings of the 14th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE Computer Society, 2014, pp. 95–104.

[36] G. Szoke, C. Nagy, P. Hegedüs, R. Ferenc, and T. Gyimóthy, "Do Automatic Refactorings Improve Maintainability? An Industrial Case Study," in *Proceedings of the 31st International Conference on Software Maintenance and Evolution (ICSME)*, R. Koschke, J. Krinke, and M. P. Robillard, Eds. IEEE, 2015, pp. 429–438.

[37] Tclose, Jhodges3, Bhill2, Gmaone, and Gmarkham, "Same origin policy," in *W3C Web Security Wiki*, 2009, https://www.w3.org/Security/wiki/Same_Origin_Policy.

[38] A. van Deursen, A. Mesbah, and A. Nederlof, "Crawl-Based Analysis of Web Applications: Prospects and Challenges," *Science of Computer Programming*, vol. 97, pp. 173–180, 2015.

[39] Web Technology Surveys, "Usage of CSS for websites," http://w3techs.com/technologies/details/ce-css/all/all, 2015.

[40] World Wide Web Consortium, "CSS Validation Service," http://jigsaw.w3.org/css-validator.

[41] M. Zalewski, "Browser Security Handbook, part 2," in *Google*, 2008, https://code.google.com/archive/p/browsersec/wikis/Part2.wiki.