Undoing Style in Cascading Style Sheets

Leonard Punt
leonardpunt@gmail.com

August 17, 2015, 32 pages

Supervisors: Sjoerd Visscher and Vadim Zaytsev
Host organisation: Q42, http://q42.com

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering
http://www.software-engineering-amsterdam.nl

Contents

Abstract

Cascading Style Sheets (CSS) is a widely used language in today's web applications for defining the presentation semantics of web documents. Despite the relatively simple syntax of CSS, the language has a number of complex features, like inheritance, cascading, and specificity, which make CSS code challenging to understand and maintain. One of the consequences is that it is not uncommon for CSS code to contain code smells. 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.

In this thesis we define a number of code smells for CSS. Furthermore we focus on one code smell, namely *undoing style*. Our definition of undoing style is: a property that is first set to a value `A`, next the property 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`. 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 a subset of these 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.

Chapter 1

Introduction

Cascading Style Sheets (CSS) [Cona] is a language used for defining the presentation semantics of web documents, like positioning, sizes, colors and fonts. CSS is widely used – over 95% of web developers use CSS, and over 90% consider it a web standard [Net10], next to that it is used in 90% of the websites [Sur].

Despite the relatively simple syntax of CSS, CSS code is not easily understood and maintained [MM12]. The language has a number of complex features, like inheritance, cascading, and specificity [Lie05, Cona]. Next to that, established design principles and tool support are missing [MTM14]. Therefore, one of the consequences is that it is not uncommon for CSS code to contain code smells [Gha14]. 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 [FB99].

In recent study, Mazinanian et al. [MTM14] 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 [Gha14] showed that CSS smells are widespread in today's websites; 99.8% of the websites (i.e., 499 out 500) analyzed in their 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*. Because of several reasons, scope being the main one, there will not be refactoring opportunities proposed for every defined CSS code smell. The reasons why we choose undoing style over other smells are described in section 2.4.

1.1 Research questions

In this work, we answer the following research questions:

RQ1. What code smells can be defined for CSS?

RQ2. What is the extent of undoing style in CSS?

RQ3. What are refactoring opportunities for undoing style that can be applied?

RQ4. How to use refactoring opportunities for undoing style that can be applied?

RQ5. Do the proposed refactorings preserve semantics?

1.2 Research method

In order to answer the first research question a literature study is conducted. The results of this literature study can be found in section 2.4.

The second, third, fourth and fifth research questions are answered by designing a tool that is capable of: (1) detecting the code smell undoing style in CSS, (2) detecting refactoring opportunities in the detected undoing style smells, (3) applying the refactoring opportunities, and (4) validating

whether the refactorings preserve the semantics. A prototype of this tool is implemented and used to validate the results.

The details on the detection of undoing style can be found in section 3.2. More details on detecting and applying refactoring opportunities can be found in section 3.3. More details on validating whether refactorings preserve semantics can be found in section 3.4.

The tool developed to answer the second, third, fourth and fifth research question is evaluated by conducting a case study. In this case study 41 real-world web applications are tested against our tool. More details on the design, results and discussion of the results of this case study can be found in chapter 4.

In this case study we used the empirical data that is used in the study conducted by Mazinanian et al. [MTM14]. We resolved some issues in this dataset, these are described in section 4.1.3. Furthermore, we added two web applications developed by the host company to the subjects.

Additionally to the case study, we provide a proof outline to argue about the correctness of the applied refactoring. This proof outline can be found in section 4.4. The formal proof is left as future work.

1.3 Outline

The rest of this thesis is organized as follows. The next chapter describes the CSS language. In chapter 3 we present all our learnings about detecting and refactoring the undoing style smell. Chapter 4 contains the design, results and discussion of the results of our case study. The thesis closes with future work and conclusions.

Chapter 2

The CSS Language

CSS [Cona] is a language used for defining the presentation semantics (i.e. style) of web documents. It is often used along with HTML, which is a language used for defining the markup of web documents. CSS enables separation of concerns by disconnecting the style from the markup. Furthermore, CSS enables reuse of styling code. Because the style is separated from the markup, the same style sheet can be applied to different markups. In this thesis we use the level 3 specification of CSS, CSS3.

Style sheets can either be internal or external. Internal means the style sheet is embedded inside the HTML document. External means the style sheet is an external file, which is then linked to HTML documents.

A style sheet consists of a list of rules. Each rule consists of a selector and a list of style declarations. Each style declaration consist of a property and a value. A grammar for style sheets is listing in 2.1. An example of a rule is shown in listing 2.2.

```
1 StyleSheet ::= [Rule]
2 Rule ::= Selector [Declaration]
3 Selector ::= String
4 Declaration ::= Property Value
5 Property ::= String
6 Value ::= String | Integer
```

Listing 2.1: CSS grammar.

```
1 p {
2   color: red;
3   text-align: center;
4 }
```

Listing 2.2: A CSS rule.

In this example `p` is the selector. A selector is used to define to which elements in HTML document this style should be applied. In the case of this example, the style will be applied to all `<p>` elements.

The second and third line are the style declarations. Style declarations are used to define the style. The first part of a style declaration, `color` and `text-align`, denotes the property. The second part of a style declaration, `red` and `center`, denotes the value.

So in case of this example, all `<p>` elements in the HTML document will have a red color and their text will be aligned to the center.

2.1 Selectors

CSS allows various selectors. In the previous example we have used an *element selector*, which is simply the name of an HTML element. Using an element selector results in the style being applied to all elements of that type.

The second type of selector is the *ID selector*. It is possible to assign an unique identifier (ID) to an HTML element. For example, `<section id="most_viewed_videos">` assigns the ID `most_viewed_videos` to the element `section`. This identifier can then be used as an ID selector. In this case the ID selector will be `#most_viewed_videos`. Using an ID selector allows us to declare the style for one element only.

A third selector is the *class selector*. Next to assigning an ID to an HTML element, it is also possible to assign a class. The difference between an ID and a class is that an ID can only be assigned to one element, whereas a class can be assigned to multiple elements. This allows us to declare the style for multiple elements only once, thus enabling reuse. An example of a class selector is: `.intro`.

A class selector can also be used in combination with an element selector or ID selector. This allows us to declare the style for all elements that both match a element or ID selector and a class selector. An example of an element selector in combination with a class selector is: `p.intro`. An example of an ID selector in combination with a class selector is: `#most_viewed_videos.intro`.

The CSS language allows to group different selectors. Grouping is done by listing the selectors after each other, separated by a comma. Grouping is useful if the same style should be applied to multiple elements. For example, if we want both the `h1` and `h2` elements to be blue, we could write `h1,h2 { color: blue; }`.

We can also combine selectors using four different combinators. Combining can be done to achieve more specific selectors. Assuming A and B are selectors, the four combinators are:

Descendant combinator (A B) Selects all elements selected by B that are inside elements selected by A.

Child combinator (A > B) Selects all elements selected by B where the parent is an element selected by A.

General sibling combinator (A + B) Selects all elements selected by B that are placed immediately after elements selected by A.

Adjacent sibling combinator (A ∼ B) Selects every element selected by B that are preceded by an element selected by A.

Selectors can also be defined as pseudo-classes and pseudo-elements. A pseudo-class is used to define a special state of an element. For example, it can be used to style an element when a user mouses over it. A pseudo-element is used to style specified parts of an element. For example, it can be used to style the first letter, or line, of an element.

Finally, we can add attribute conditions to a selector. An attribute selector is used to select elements with a specified attribute. For example, `a[target]` selects all `<a>` elements that have the `target` attribute set to any value. Furthermore, numerous operators exist to select only elements with a specified attribute and value.

2.2 Inheritance and cascading order

Inheritance in CSS is the process that elements pass their style properties to their child elements, even though these style properties have not been explicitly defined for the children. Note that not all style properties inherit their style.

Frequently multiple rules apply to an 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.

2.2.1 Location

The style can be defined in three different locations. The first location is inline, 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 last location is external, the style sheet is an external file and is linked to the HTML document.

The rules defined inline get precedence over internal and external style sheets. An internal style sheet gets precedence over an external style sheet.

2.2.2 Specificity

If two rules are in the same location, the rule with the highest specificity gets precedence. The specificity of a selector S is a four-digit matrix: `[a,b,c,d]` where:

- a is 1 if style is defined inline, 0 otherwise.
- b is the total number of ID selectors.
- c is the total number of class selectors.
- d is the total number of element selectors.

Note that the result is a matrix, not a score. The values should be compared column by column. If one would concatenate the result and use it as a score, 10 class selectors would have the same specificity as one ID selector, which is incorrect. An ID selector has a higher specificity than any number of class selectors.

If multiple rules happen to have the same specificity, and all are declared in external style sheets, than the rule in the style sheet that is linked last gets precedence over the other rules.

If multiple rules happen to have the same specificity, and all are declared in the same style sheet, than the rule that is defined last gets precedence over the other rules.

It is possible for a selector to bypass the specificity rules by adding the `!important` annotation at the end of a style declaration. Then the style declaration is always applied.

2.3 Media rules and queries

A special type of a CSS rule is a CSS media rule. A CSS media rule contains a set of nested CSS rules, with a condition defined by a media query. Only if this condition holds, the nested CSS rules are applied.

A media query consists of a media type, for example `screen` or `print`, and at least one expression that limits the style sheets' scope. In this expression media features are used, like the width or height of the page.

Media rules and queries can be used to let the presentation of content depend on the output device. An example is listed in 2.3.

```
1 @media screen and (max-width: 300px) {
2   .test {
3     margin-left: 0px;
4   }
5 }
```

Listing 2.3: Example of CSS media rule.

2.4 Code smells in CSS (RQ1)

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 [FB99].

Code duplication is probably the most well known code smell. Code duplication is actually more prevalent in CSS code compared to procedural and object-oriented code. Main reason is the lack of variables and functions that could be used to build reusable blocks of code. In our work we do not focus on code duplication since Mazinanian et al. already proposed a way of refactoring code duplication for CSS [MTM14].

Another well known smell is *dead code*. Dead code exists in CSS in the form of *unused selectors* and *redundant property declarations*. In our work we don not focus on dead code. Refactoring dead code is just a matter of deleting the rules or properties. The interesting part is detecting dead code, which is something Mesbah et al. and Bosch et al. already studied and found solutions for [MM12, BGL14].

Undoing style is a smell studied by Gharachorlu [Gha14]. However, we think the definition Gharachorlu uses is not precise enough. Furthermore, to the best of our knowledge, refactoring undoing style is not studied yet. Therefore we chose to study undoing style more thoroughly. More details can be found in chapter 3.

Using CSS directly in HTML of JavaScript is a smell, namely *violation of separation of concerns*. We chose not to focus on this smell, since in our experience almost every developer uses external style sheets nowadays. Many websites contain CSS directly in HTML or JavaScript [NNN+12, Sur]. However, we think this is mainly code injected by frameworks. Refactoring frameworks will be hard, 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* [Gha14]. The problem with these smells is that there is no standard on what is 'too long', or 'too much', or 'high', or 'too general'. To avoid controversy, we chose not to include these smells in our tool.

Furthermore, there are numerous linters that detect errors and smells, like CCS Lint, Codacy, CSSNose and W3C CSS Validator [CSS, Conb, Cod, Gha14]. An overview of these errors and smells can be found in table 2.1. Note that Codacy includes CSS Lint and CSSNose includes both CSS Lint and W3C CSS Validator.

The problem with linters is that they check whether the code corresponds with 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.

Category	Error/smell	CSS Lint	W3C CSS Validator	Codacy	CSSNose
Possible errors	Beware of box model size	✓		✓	✓
	Require properties appropriate for `display`	✓		✓	✓
	Disallow duplicate properties	✓		✓	✓
	Disallow empty rules	✓		✓	✓
	Require use of known properties	✓		✓	✓
Compatibility	Disallow adjoining classes	✓		✓	✓
	Disallow `box-sizing`	✓		✓	✓
	Require compatible vendor prefixes	✓		✓	✓
	Require all gradient definitions	✓		✓	✓
	Disallow negative text-indent	✓		✓	✓
	Require standard property with vendor prefix	✓		✓	✓
	Require fallback colors	✓		✓	✓
	Disallow star hack	✓		✓	✓
	Disallow underscore hack	✓		✓	✓
	Bulletproof font-face	✓		✓	✓
Performance	Don't use too many web fonts	✓		✓	✓
	Disallow `@import`	✓		✓	✓
	Disallow selectors that look like regular expressions	✓		✓	✓
	Disallow universal selector	✓		✓	✓
	Disallow unqualified attribute selectors	✓		✓	✓
	Disallow units for zero values	✓		✓	✓
	Disallow overqualified elements	✓		✓	✓
	Require shorthand properties	✓		✓	✓
	Disallow duplicate background images	✓		✓	✓
Maintainability & Duplication	Disallow too many floats	✓		✓	✓
	Don't use too many font-size declarations	✓		✓	✓
	Disallow IDs in selectors	✓		✓	✓
	Disallow `!important`	✓		✓	✓
Accessibility	Disallow `outline:none`	✓		✓	✓
OOCSS	Disallow qualified headings	✓		✓	✓
	Headings should only be defined once	✓		✓	✓
Conformance	Parse Error: Syntactical issues in lines of CSS code such as missing braces		✓		✓
	Value Error: The value of a property is not recognized by the browser		✓		✓

Table 2.1: Overview of errors and smells detected by linters.

Chapter 3

Undoing style in CSS

3.1 Definition

In our work we focus on the code smell *undoing style*. Our definition of undoing style is: a property that is first set to a value `A`, next the property 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`. We refer to this pattern as the `A-B*-A` pattern. An example of undoing style is described in section 3.1.3.

The `A-B*-A` pattern matches a several flavors of smells, for example `A-B-A`, `A-B-C-D-A` and `A-A`. We gather all these smells under the name *undoing style*, but it might be possible that some of the smells will be given other names in the future.

Although the `A-B*-A` pattern matches a several flavors of smells, we claim that there are no false positives. All the matched flavors of smells are 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 to early. This pattern might indicate that people do not get the cascading and inheritance properties of CSS.

3.1.1 Issues with Gharachorlu's definition

Our definition of undoing style is more precise than the definition Gharachorlu uses [Gha14]. Gharachorlu marks every property value that is `0` or `none` as a smell. This definition produces both false positives and false negatives.

The false positives are produced because not every `0` or `none` property value is bad. An example of a `none` property value that is perfectly fine is if a developer wants to hide an element. The correct way to do this is to set the property `display` to value `none`. However, using Gharachorlu's definition, this `none` value will be marked as a code smell. Therefore Gharachorlu's definition produces false positives.

Our definition, however, will not mark the `none` value in the previous example as a smell. We check if the property `display` is first set to `none`, then to some other value and then back to `none` again, which is not the case.

Another example of false positives produced by Gharachorlu's definition is that all reset styles are marked as smells. 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 some consider it good practice [Mey, RH].

Since the reset styles only set a value, there can only be an `A-A` pattern. That is, the reset style is the same of the browser style. Therefore, when using our definition of undoing style, not all reset styles will be marked as smells. Only the reset styles that have the same value as the browser default are marked as smells.

Furthermore, using Gharachorlu's definition, false negatives are produced because only `0` or `none` property values are taken into account. However, a style can be undone by by any valid value for that

property. For example, if the developer first sets the `margin` to `25px`, next to `50px`, and thereafter back to `25px`, the style is undone as well, and thus should this occurrence be marked as a code smell. However, using Gharachorlu's definition, this smell will not be detected. Therefore Gharachorlu's definition produces false negatives.

Using our definition, these false negatives are mitigated. We do not only check for `0` or `none` values, but for every valid value for a property. If we detect an `A-B*-A` pattern, regardless of the values for `A` and `B`, we mark it as a smell.

3.1.2 Implicit values

Note that in order to detect all occurrences of this smell, the implicit values have to be taken into account as well. 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, than the implicit value is the initial value for non-inheriting properties and the inherited value for inheriting properties.

An 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` to `10px`, and next to `0`, it should be marked as undoing style, since the implicit value for the `margin` is `0`.

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.

3.1.3 Example

An example of an undoing style is shown in listings 3.4 and 3.5. This example is a simplified version a smell we found on one of the websites the host company 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 to 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` is not necessary.

```
1  .input-field > * {
2      border: 0 none;
3      float: left;
4  }
5
6  .text-field input {
7      float: none;
8      padding-left: 48px;
9      width: 100%;
10 }
```

Listing 3.4: Example of undoing style - CSS.

```
<div class="input-field text-field">
  <label for="field1">...</label>
  <input type="text" name="field1" id="field1" class="rel">
</div>
<div class="input-field text-field">
  <label for="field2">...</label>
  <input type="text" name="field2" id="field2" class="rel">
</div>
<div class="input-field text-field">
  <label for="field3">...</label>
  <input type="text" name="field3" id="field3" class="rel">
</div>
```

Listing 3.5: Example of undoing style - HTML.

3.1.4 Exception

There is an exception to the definition of undoing style, namely pseudo-classes. A pseudo-class is used to define a special state of an element. 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 to the first. Next a CSS rule can be created, using the class selector and the appropriate style. However, more convenient is 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-class `:first-child`.

The pseudo-classes in which this pattern is allowed are `:first-child`, `:last-child`, `nth-child` and `nth-last-child`.

3.2 Detection

In order to detect the smell undoing style, we need information from both the CSS and the HTML. With this information 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 to figure out their location in order to know 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 `A-B-C-B-A`, in this pattern contains two `A-B*-A` patterns, namely `A-B-C-B-A` and `B-C-B`. However, because `B-C-B` is embedded in `A-B-C-B-A`, we ignore `B-C-B`.

Patterns are allowed to overlap, as long as they do not overlap completely. Take the pattern `A-B-A-B` for example. This patterns contains two valid `A-B*-A` patterns: `A-B-A` and `B-A-B`.

All the detected undoing style smells are refactoring opportunities. The developer should decide whether to apply a refactoring or not. In section we will outline a refactoring for the undoing style smells that adhere to certain preconditions.

The algorithm used for detecting undoing style smell can be found in listing 3.6.

3.2.1 Static analysis versus dynamic analysis

Although it might be possible to perform this analysis statically, we chose to do dynamic analysis.

```
/* 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 3.6: Algorithm to detect the undoing style smell.

The problem with static analysis is that nowadays every realistic web application is dynamic in some way. If we perform static analysis on these dynamic web applications, we will only capture a very limited part of the web application.

However, if we execute the HTML document in a browser (and thus perform dynamic analysis), the client-side scripts will be executed. Furthermore, the browser will give us access to the Document Object Model (DOM), which is programming interface for the parsed HTML documents. Through this interface we can request the elements of the HTML documents. Furthermore we can get access to the objects that represent the parsed style sheets (internal and external) that are used on that document.

There is a shortcoming in our analysis. The interface the DOM provides allows scripts to dynamically access and update the content, structure and style of documents. This can happen at any time. Right now we are not able to capture these updates and check if they introduce or mitigate an undoing style smell.

Our tool is, however, capable of detecting the smell on multiple DOM instances. So if a developer provides all possible DOM states, our analysis will be sound. The only shortcoming is that the tool itself doesn't capture all the possible DOM states, this is left as future work.

3.3 Refactoring

Refactoring code is the act of restructuring or rewriting existing code, while preserving the semantics. A refactoring does not change the behavior of the code, but it improves its qualities, like maintainability [FB99].

3.3.1 Detecting a refactoring opportunity

Before we are able to determine whether an undoing style smell can be refactored we need to figure out a couple of things. First, we need to know which CSS rules contain the `A-B*-A` pattern. We call the first `A` the initial rule, this is the rule that comes lowest in the cascading order. The `B*` are the enclosed rules and the final `A` the reset rule, this is the rule which style is currently applied. Second, we need to know to which nodes these rules apply.

Once we have this information we're able to check if the `A-B*-A` pattern adheres to two preconditions. If the detection adheres to these to preconditions, it is a refactoring opportunity. The preconditions are needed in order to preserve the semantics when applying the refactoring opportunity. More information about why the preconditions are needed can be found in the outline for the proof of correctness, in section 4.4.

The first precondition is that the most specific part of the selector of the enclosed rule should be an ID or a class selector. The second precondition is that the elements that the reset rule applies to, should be a subset of the elements the initial rule applies to.

3.3.2 Applying a refactoring opportunity

Three steps need to be taken when applying a refactoring opportunity.

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. First, it needs to be defined directly beneath the enclosed rule. Second, 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 precondition.

The second step is to take the difference between all nodes of the enclosed rule and the nodes of the reset rule. Next we add the new class, that is the most specific part of the selector of the new rule, 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, the whole rule can be deleted 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.

3.3.3 Example

An example of a refactored undoing style is shown in listings 3.9 and 3.10. The original code is shown in listings 3.7 and 3.8.

In this example the initial rule is the first rule, the enclosed rule is the second rule, and the reset rule is the third 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.

As we can check, both preconditions hold. The most specific selector part of the enclosed rule is a class selector. The elements that apply to the reset rule is a subset of the element that apply to the initial rule.

If we look at the refactoring code, we see that the rule with selector `.refactoring-1` 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 `.refactoring-1` 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.

```
a {
  text-decoration-line: none;
}

.cms .link {
  text-decoration-line: underline;
}

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

Listing 3.7: Example of refactored undoing style - CSS.

3.3.4 Exception

Note that our tool ignores media rules and thus does not refactor undoing style smells that occur inside media rules. The reason for this is that some of these occurrences are impractical to refactor. An example is shown in listing 3.11.

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

Listing 3.8: Example of undoing style - HTML.

```css
a {
  text-decoration-line: none;
}

.cms .refactoring-1 {
  text-decoration-line: underline;
}
```

Listing 3.9: Example of refactored undoing style - CSS.

```html
<div class="cms">
  <a href="#" class="link refactoring-1">...</a>
  <a href="#" class="link more">...</a>
</div>
```

Listing 3.10: Example of refactored undoing style - HTML.

```css
.test {
  margin-left: 50px;
}

@media screen and (max-width: 300px) {
  .test {
    margin-left: 0px;
  }
}
```

Listing 3.11: Example of CSS media rule.

In order to refactor the undoing style listed in 3.11, the algorithm described in this section does not work. It will create a new rule with the enclosed style, which is placed outside the media rule, and remove the rule inside the media rule. The new class will not be added to any node, since both selectors apply to the same nodes. As a consequence, all nodes will get the implicit style `0`, which is incorrect.

It is possible to rewrite the example listed in 3.11. The first step is to create a new media rule with a media query that is the inverse of the media query of the existing rule. The next step is to move the rule with the enclosed style inside the new media rule. Then it is possible to remove the reset rule. However, a rewriting like this most likely increases the complexity, and thus reduces the maintainability and readability. That is why we do not consider it a refactoring opportunity and decided to ignore media rules.

3.4 Preserving semantics

As mentioned in the section 3.3, a refactoring should preserve the semantics. How do we claim that semantics are preserved? The tricky part is that the DOM is dynamic, at any given moment a client is able to apply an update. This ability to apply client-side DOM updates is used by many web applications.

We are simply not able to check that for any given update to the DOM, a refactoring preserves the semantics. Therefore we ask the developer to elaborate all possible DOM states. If the developer provides all possible DOM states, we claim that our refactoring will preserve the semantics. An outline for the proof of correctness can be found in section 4.4.

These DOM states are used to check if the semantics are preserved as well. We request the computed styles of each element, both before and after the refactoring. The computed styles of an element are the values of all the CSS properties after applying the active style sheets. The computed styles should not have changed after the refactoring is applied. If that is the case, we claim the semantics are preserved.

Currently, the tool itself doesn't capture all the possible DOM states, this is left as future work. This functionality can be implemented in several ways. One option is to use a crawler. This crawler should persist every DOM state it encounters. Another option is to make use of the test suite of the web application, if it exists. All the DOM states the front-end tests visit should be persisted. A third is to use the visual style guide of a web application, if it exists. The visual style guide should contain many or even all DOM states that are possible. Finally, it is possible to generate DOM states from the source code. However, in many cases that would require parsing of the client-side scripting code and the template files, from which the HTML documents are generated.

3.5 Qualities

In the section 3.3 we talked about refactorings improving qualities of the code. Examples of qualities are understandability, maintainability and reusability.

In order to measure these qualities, several metrics can be used. For example *lines of code*, or *number of rule blocks*. However, the correlation between these metrics and the qualities is unclear.

A more sophisticated metric is the *abstractness factor*. The abstractness factor is an indicator of the degree of the separation of content and presentation. A high abstractness factor represents a high maintainability and reusability of both the style sheet and the HTML document [KN10].

Our goal is less lines of code. However, because of grouped selectors and shorthand properties, a refactoring might actually result in more lines of code. For example, if there is an undoing style pattern in a shorthand property, the shorthand property has to be deleted and all properties have to be declared individually. Therefore we're talking about refactoring opportunities. The developer has to decide whether refactoring the code pays off or not.

Chapter 4

Evaluation

To assess the efficacy of our approach, we conducted a case study addressing the research questions RQ2, RQ3, RQ4 and RQ5. RQ1 is already addressed in section 2.4. The research questions are repeated below:

RQ2. What is the extent of undoing style in CSS?

RQ3. What are refactoring opportunities for undoing style that can be applied?

RQ4. How to use refactoring opportunities for undoing style that can be applied?

RQ5. Do the proposed refactorings preserve semantics?

The tool and the empirical data are available online.[1]

4.1 Experiment design

4.1.1 Selection of subjects

In total, our case study contains 41 subjects. In order to select representative real-world web applications, we used the empirical data that is used in the study conducted by Mazinanian et al. [MTM14]. This data set includes 38 randomly selected, and author selected online web applications. It includes a subset of the top-100 visited web sites based on Alexa ranking. Furthermore web applications developed by companies considered leaders in web technologies, such as Facebook, Yahoo!, Google, and Microsoft, are added. This data set is available online [Maz].

Besides the 38 subjects from Mazinanian et al. their study, two web applications developed by the host organisation have been studied as well.

The 41th subject is a new version of the subject 'Gmail' from in the original data set. More information on this can be found in section 4.1.3.

The complete list of the selected systems is shown in table 4.1.

4.1.2 Extraction of CSS styles and DOM states

Mazinanian et al. used the dynamic analysis features of Crawljax [MvDL12] to dynamically capture different DOM states of a web application. These DOM states are persisted to HTML files.

The HTML files contain inline and internal style sheets, together with links to external style sheets. In order to extract the external style sheets Mazinanian et al. developed an external CSS file extractor plug-in for Crawljax.

Note that the references to the external style sheets in the HTML documents need to be updated, because the extracted external style sheets are in a different location.

[1] http://leonardpunt.github.io/masterproject/

Facebook	Pinterest
YouTube	Reddit
Twitter	Tumblr.com
YahooMail	Wordpress.org
Outlook.com	Vimeo.com
Gmail	Igloo
Github	Phormer
Amazon.ca	BeckerElectric
Ebay	Equus
About.com	ProToolsExpress
Alibaba	UniqueVanities
Apple.com	ICSE12
BBC	EmployeeSolutions
CNN	SyncCreative
Craiglist	GlobalTVBC
Imgur	Lenovo
Microsoft	MEC
MSN	Staples
Paypal	MSNWeather
9292.nl	Rijksmuseum.nl

Table 4.1: Selected subjects.

4.1.3 Issues with dataset

There are some problems with Mazinanian et al. their dataset. First there are two sites with an incorrect name, 'Apple.ca' is actually 'Apple.com' and 'MountainEquip' is 'MEC'. In our study we renamed these two sites to their correct name.

Next, the DOM state that is captured for the subject 'Gmail' is not usable. The problem is that a cookie is missing, therefore a visitor will be redirected to a non-existing page. We mitigated this problem by removing the redirect from the source code. Besides that we captured the intended state ourselves as well. These two subjects are named 'Gmail original' and 'Gmail fixed'.

Furthermore several external style sheets are captured incorrectly, resulting in an empty style sheet. We chose to include these style sheets as is in our study, in order to stay as close to the original dataset as possible. We did investigate why the style sheets were empty, our findings are listed below:

- 'MEC', the file 282a12.css. This is a CSS file in a `<script>` tag, probably an error in the document.
- 'About.com', the files `18b91843bb4bcb07c2ba68a01bbb8a02b9eb4c50.css` and `54c660b14dd08ca6b408f07de1f5080d251a4ef2.css`. The original URLs for these files do return style sheets, so probably an error occurred while fetching these style sheets.
- 'Apple.com', the file 91cab95bff78fd3800625c0789cd87c0e4180299.css. When we tried to retrieve this file, a File Not Found error was returned. However, we do not know if this is also the error that occurred when the original dataset was collected.
- 'GlobalTVBC', the file 61dc696007ca3c1aeb54a2d0bab8ea932de60e21.css. When we tried to retrieve this file, a Forbidden error was returned. However, we do not know if this is also the error that occurred when the original dataset was collected.
- 'Alibaba', the files d8e76b82abbae61a4a89fb4000324c09b6413719.css and f20bbdc283941382159c1e12d655f54f1bdc68c2.css. The original URLs for these files do return style sheets, so probably an error occurred while fetching these style sheets.

- 'SyncCreative', the file 0e6fedfab56593cd6d0ea0bd8dee80454585b2af.css. When we tried to retrieve this file, a Forbidden error was returned. However, we do not know if this is also the error that occurred when the original dataset was collected.

Finally, some unused style sheets have been captured. We have excluded these style sheets from our study. The files are:

- 'ProToolsExpress' the files: 167d8fb47eb42d1f908ba5d4141a34f4333b18c9.css, a0f24af3ff23a278289a1f50a5d6b6598a76415a.css, b5f14f865786216f67ae8a41ab1c1774aa955334.css, c7a368297aab3abe7d74e0ae421fc38dd18a048f.css and e54053a51b1eb8ae62e9bc76ad9351ea4f1d4c89.css. These files are not linked in any document.
- 'SyncCreative' the file style.css. Since this file is a duplicate of b4ad21b4c1ba99451234f5e3da9a501a50dac0fe.css.

4.1.4 Detection and refactoring of undoing style

In order to detect and refactor the instances of undoing style in the set of persisted DOM states S collected from a web application, we:

1. Fix the references to external style sheets in S.
2. Detect the undoing style smells in S, as described in section 3.2.
3. Refactor a subset of the detected smells in S, as described in section 3.3.
4. Check if the semantics are preserved in S, as described in section 3.4.

4.2 Results

4.2.1 Extent of undoing style in CSS (RQ2)

The result of our empirical study shows that undoing style is prevalent in CSS code. Figure 4.1 displays a box plot with the ratios between the number of detections and the number of rules in the analyzed style sheets. The median value for the ratio is 16% while the average is 25%.

The outliers are displayed as orange dots. The most extreme outlier has a ratio of 242%. This is caused by a client-side script that copies an inline style sheet. Therefore many elements contain an A-A pattern.

Figure 4.2 displays a box plot with the number of undoings styles detected in the analyzed CSS code. On average, we were able to detect 380 occurrences of undoing style. The outliers on this boxplot are subjects with a lot of CSS rules, which explains the higher number of undoing style smells.

The percentages of the different types of the `A-B*-A` pattern are displayed in figure 4.3. As can be observed, 49% of the detections are of type `A-B-A`, while 36% of the detections are of type `A-A`. Furthermore, the existence of type `A-B(4)-A` or longer is rare. The longest pattern we found was `A-B(70)-A`.

Additionally we looked at the initial CSS rule, the rule that initiates the `A-B*-A` pattern. We found that in 63% of the detections the initiator is an implicit value and in 18% an element selector. More details can be found in figure 4.4. 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.

4.2.2 Applied refactorings (RQ3 and RQ4)

Figure 4.5 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 applied all refactoring opportunities, the number of refactorings is equal to the number of refactoring opportunities.

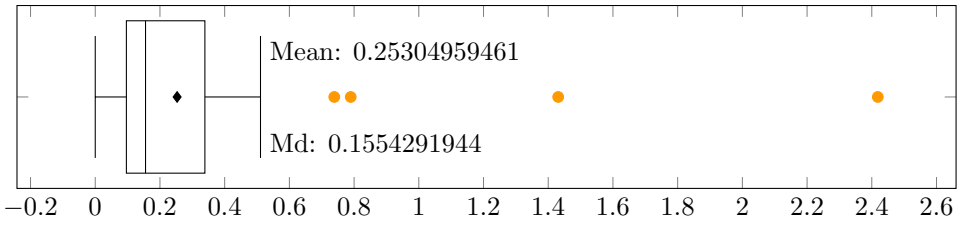

Figure 4.1: Ratio of detected undoing style smells to CSS rules.

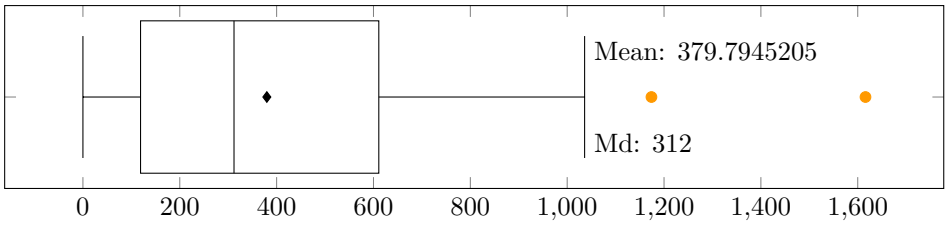

Figure 4.2: Total number of detected undoing style smells.

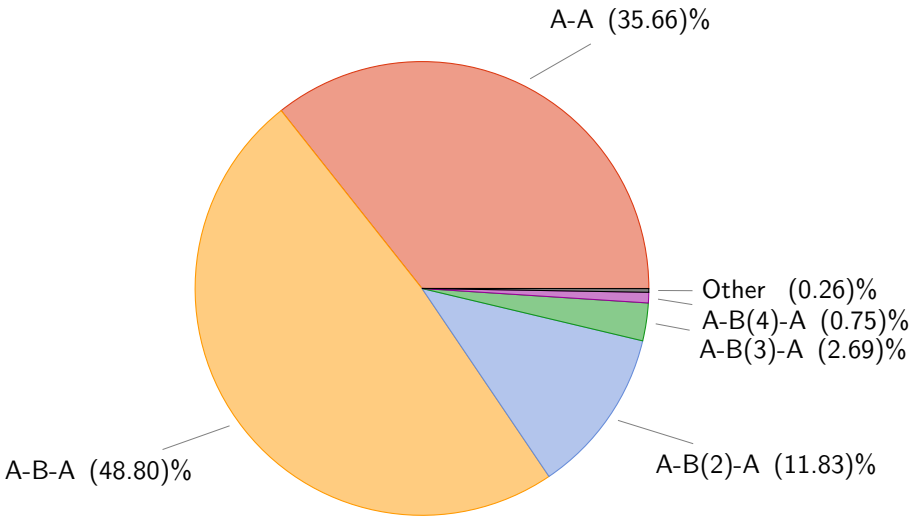

Figure 4.3: Types of detections.

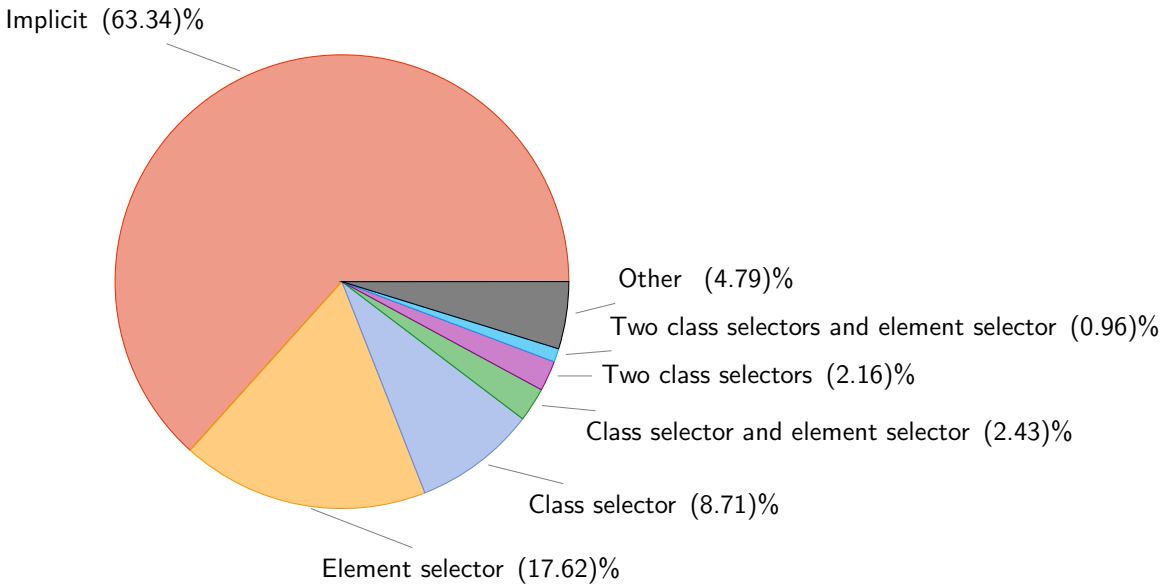

Figure 4.4: Origin or specificity of initial CSS rule.

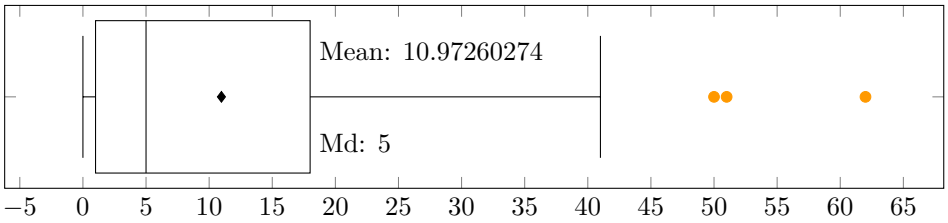

Figure 4.5: Number of applied refactorings.

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 detected undoing style smells, which explains the higher number of applied refactorings.

The detections that are not refactored can be used as input when developing more advanced refactorings.

4.2.3 Semantics changes (RQ5)

A semantic change is a change for the value of a property at a node. 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 very likely that changes are detected that are not introduced by the refactoring.

In total 8789 semantic changes are detected. For all these changes, only 38 are introduced because a refactoring was applied.

Figure 4.6 displays a box plot with all the changes to semantics. As can be observed the median is 2 and the mean is 120. So 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.

We know most of the errors are not introduced as a result of applying a refactoring, because either:

- There are no refactoring opportunities for that subject.
- The CSS properties that contained errors are not CSS properties that are refactored.

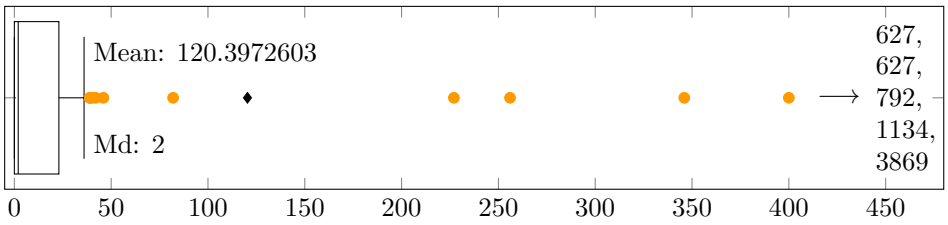

Figure 4.6: Number of changes to semantics.

- 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 are because the tool returns an incorrect cascading order if two rules have the `!important` annotation.

4.3 Discussion

4.3.1 Undoing style and applying refactorings

Our case study shows that undoing style is prevalent in the CSS code of today's web applications. The pattern `A-B-A` is most prevalent, followed by the pattern `A-A`.

Furthermore, most of the undoing style patterns start with an implicit value or an element selector.

The amount of patterns that start with an implicit value indicates that developers do not know or trust the implicit values, and therefore override them, probably unintentionally. This also explains why the `A-A` and `A-B-A` patterns are most prevalent.

The amount 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 our evaluation also shows that our tool is capable of applying refactorings for a subset of the `A-B-A` pattern, while preserving the semantics in almost all cases.

4.3.2 Limitations

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, this is left as future work. We expect the user to provide all the possible DOM states.

There are several options to fix this limitation. One is to add a crawler that captures DOM states. Another option is to attach our tool to an existing test suite of a web application. A third option is to make use of the visual style guide of a web application, if it exists. It is also possible to generate the possible DOM states from the source code.

Another limitation is client-side DOM updates. We do not take these into account when checking whether the semantics are preserved. This causes a lot of false positives.

A third limitation is that our tool is capable on only the files that are served from the same origin. This is because all modern browsers adhere to the same-origin policy [Net]. The same-origin policy restricts how a document or script loaded from one origin can interact with a resource from another origin.

A fourth limitation is that we do not always detect if two values are equal but lexically different. This is because we use the same representation for a value as a browser uses, and these representations can be lexically different for equal values.

4.3.3 Threats to validity

Since we used the dataset of Mazinanian et al., we have the same threats to validity.

Threats to internal validity

A threat is that 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 make some of the applied refactorings to be non semantic preserving for this particular set of unvisited DOM states.

Mazinanian et al. already tried to avoid selection bias, by selecting 14 subjects from the list of web sites analyzed in a related CSS study [MM12].

Threats to external validity

To mitigate the threats to the external validity Mazinanian et al. included 24 additional web sites, developed by leading companies in web technologies applying the current state-of-the-art CSS development practices. Furthermore, we added two web sites developed by the host company.

4.4 Proof

In this section we are going to outline the proof of correctness of our refactoring. The formalized proof is left as future work. We are going to use set theory and logic to prove that the refactoring described in section 3.3 preserves all semantics.

The refactoring exists of two steps, first a CSS property is moved to a new rule, second 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 `A-B-A` pattern for a style property s, then:
SR = the set of all CSS rules.
I = the set of all HTML elements the initial CSS rule applies to.
E = the set of all HTML elements the enclosed CSS rule applies to.
R = the set of all HTML elements the reset CSS rule applies to.

SR is a totally ordered set. 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 SR.

In order for an `A-B-A` pattern to be refactorable, two preconditions have to hold:

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

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 chapter 2, the specificity of a selector is a four-digit matrix. 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. Either the most specific part of the rule is 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 matrix stays the same.

If the enclosed rule's selector has an ID selector as most specific part, we can easily add a class selector. The third digit in the specificity matrix 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.

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: `[0,0,0,1]` and `[0,0,0,2]`. 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, we end up with `p.someNewClassName`. This new selector has specificity `[0,0,1,1]` and therefore precedence over `div p`, which is incorrect.

Therefore the most specific part of the selector of the enclosed rule should be an Id or a class selector.

4.4.1 Move CSS property to new CSS rule

The first step of 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 SR.

Now we have:
E' = the set of all HTML elements the the new CSS rule applies to.
$E' = E \setminus R$.

E exists out of the sets E' and R. So 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 SR. 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 SR, the new CSS rule will be the highest rule now. Therefore, the correct value for property s is applied.

Note that the new rule doesn't have to be the highest rule in SR for any HTML element. However, because the new CSS rule comes directly above the enclosed CSS rule in SR, the order relation between any rule in SR 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 `A-B-A` pattern follows that the reset CSS rule is higher in SR 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.

We showed 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.

4.4.2 Remove CSS property

Because of the second precondition and first step of the refactoring: $R \subset (I \setminus E')$. In other words the enclosed CSS rule does not apply to R anymore.

Furthermore, from the `A-B-A` 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 RS for all elements in R.

Because of the second precondition, every element in R will have an `A-B*-A` pattern. However, the

amount 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 RS for all elements in R, if the result 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.

Chapter 5

Conclusions

In this thesis we presented an overview of code smells in CSS.

We have developed two techniques: one for detection of undoing style in CSS and the other for refactoring of the 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 performed an experiment on 41 real-world web applications and found that:

1. 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 `A-B-A` pattern was the most prevalent type of undoing style.
2. There are quite a few instances that can be refactored while preserving the presentation semantics: 11 on average, and the maximum number of refactoring applied on one state of an application was 62.
3. 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.

Furthermore we formalized the two steps our refactoring consists of, and argue that the correctness of the refactoring is implied by the correctness of the two steps.

Our work makes the following main contributions:

- We presented an overview of code smells in CSS, refining prior work [Gha14].
- We developed an open source tool that is able to detect undoing style smells in CSS: `http://leonardpunt.github.io/masterproject/` that is also capable of refactoring a subset of the detections, while preserving the semantics.
- We evaluated the proof-of-concept implementation by conducting an experiment on 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.
- We sketched a proof of correctness for our refactoring.

Related and future work

There is a wide range of papers discussing refactoring in general. Fowler and Beck [FB99] demonstrate how software practitioners can realize significant benefits to the structural integrity and performance of existing software programs using a collection of techniques. These practices are referred to as refactoring.

Källén et al. [KHH14] show that both maintainability and performance increase as an effect of the refactoring, in a code base using the object-oriented methodology.

Ammerlaan et al. [AVZ15] show that refactoring code could result in a productivity penalty in the short term if the coding style becomes different from the style developers have grown attached to.

Szőke et al. [SAN+14] find that one single refactoring only makes a small change (sometimes even decreases quality), but when we do them in blocks, we can significantly increase quality, which can result not only in the local, but also in the global improvement of the code.

Negara et al. [NCV+13] implement an algorithm which infers refactorings from continuous changes. They reveal several new facts about manual and automated refactorings. For example, more than half of the refactorings were performed manually.

Several researchers have developed techniques for detecting and ranking refactoring opportunities. Bavota et al. [BPT+14] claim that an additional source of information for identifying refactoring opportunities is team development activity. This new refactoring dimension can be complemented with others to build better refactoring recommendation tools.

Silva et al. [STV14] propose a novel approach to identify and rank *Extract Method* refactoring opportunities. Their approach aims to recommend new methods that hide structural dependencies that are rarely used by the remaining statements in the original method.

Steidl and Eder [SE14] 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 [MS14] report on an approach and tool for automatically identifying multi-language relevant artifacts, finding references between artifacts in different languages, and (rename-) refactoring them.

There are several papers discussing code smells in CSS. Mazinanian et al. [MTM14] 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 [Gha14] 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 [MM12] 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. [BGL14] present a prototype of a static CSS semantical analyzer and optimizer. 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. [GLQ12] 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. [NNN+12] 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 [KN10] 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.

As part of future work, we intend to investigate more refactoring opportunities for the `A-B*-A` pattern. In addition, we would like to extend the tool to capture more DOM states.

Bibliography

[AVZ15] Erik Ammerlaan, Wim Veninga, and Andy Zaidman. Old habits die hard: Why refactoring for understandability does not give immediate benefits. In Yann-Gaël Guéhéneuc, Bram Adams, and Alexander Serebrenik, editors, *Proceedings of the 22nd International Conference on Software Analysis, Evolution and Reengineering*, pages 504–507. IEEE, 2015. doi:10.1109/SANER.2015.7081865.

[BGL14] 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*, pages 13–16. ACM, 2014.

[BPT+14] Gabriele Bavota, Sebastiano Panichella, Nikolaos Tsantalis, Massimiliano Di Penta, Rocco Oliveto, and Gerardo Canfora. Recommending refactorings based on team co-maintenance patterns. In *Proceedings of the 29th IEEE/ACM International Conference on Automated Software Engineering*, pages 337–342. ACM, 2014. doi:10.1145/2642937.2642948.

[Cod] Codacy. Patterns list. URL: https://www.codacy.com/patterns [cited 09-04-2015].

[Cona] World Wide Web Consortium. CSS specifications. URL: http://www.w3.org/Style/CSS/current-work [cited 15-04-2015].

[Conb] World Wide Web Consortium. CSS Validation Service. URL: http://jigsaw.w3.org/css-validator/ [cited 09-04-2015].

[CSS] CSSLint. Rules. URL: https://github.com/CSSLint/csslint/wiki/Rules [cited 9-4-2015].

[FB99] Martin Fowler and Kent Beck. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[Gha14] Golnaz Gharachorlu. Code smells in cascading style sheets: An empirical study and a predictive model. Master's thesis, University of British Columbia, 2014.

[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*, pages 809–818. ACM, 2012.

[KHH14] Malin Kallen, Sverker Holmgren, and Ebba Hvannberg. Impact of Code Refactoring Using Object-Oriented Methodology on a Scientific Computing Application. In *Proceedings of the 14th International Working Conference on Source Code Analysis and Manipulation*, pages 125–134. IEEE Computer Society, 2014. doi:10.1109/SCAM.2014.21.

[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 seventh International Conference on the Quality of Information and Communications Technology (QUATIC)*, pages 116–121, 2010.

[Lie05] Håkon Wium Lie. *Cascading Style Sheets*. PhD thesis, University of Oslo, Norway, 2005.

[Maz] Davood Mazinanian. Dataset for FSE'14 submission. URL: http://users.encs.concordia.ca/~d_mazina/papers/FSE'14/ [cited 24-06-2015].

[Mey] Eric A. Meyer. Reset reasoning. URL: http://meyerweb.com/eric/thoughts/2007/04/18/reset-reasoning/ [cited 26-05-2015].

[MM12] Ali Mesbah and Shabnam Mirshokraie. Automated analysis of CSS rules to support style maintenance. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 408–418. IEEE, 2012.

[MS14] Philip Mayer and Andreas Schroeder. Automated Multi-Language Artifact Binding and Rename Refactoring between Java and DSLs Used by Java Frameworks. In Richard Jones, editor, *Proceedings of the 28th European Conference on Object-Oriented Programming*, volume 8586 of *Lecture Notes in Computer Science*, pages 437–462. Springer International Publishing, 2014. doi:10.1007/978-3-662-44202-9_18.

[MTM14] Davood Mazinanian, Nikolaos Tsantalis, and Ali Mesbah. Discovering refactoring opportunities in cascading style sheets. In *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, page 11, 2014.

[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.

[NCV+13] Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph E. Johnson, and Danny Dig. A Comparative Study of Manual and Automated Refactorings. In Giuseppe Castagna, editor, *Proceedings of the 27th European Conference on Object-Oriented Programming*, volume 7920 of *Lecture Notes in Computer Science*, pages 552–576. Springer International Publishing, 2013. doi:10.1007/978-3-642-39038-8_23.

[Net] Mozilla Developer Network. Same-origin policy. URL: https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy [cited 24-07-2015].

[Net10] Mozilla Developer Network. Mozilla web developer survey research. Technical report, Mozilla, 2010.

[NNN+12] Hung Viet Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, Anh Tuan Nguyen, and Tien N. Nguyen. Detection of embedded code smells in dynamic web applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 282–285. IEEE, 2012.

[RH] Morten Rand-Hendriksen. Why a CSS reset should be at the core of your stylesheet. URL: http://mor10.com/why-a-css-reset-should-be-at-the-core-of-your-stylesheet/ [cited 26-05-2015].

[SAN+14] Gábor Szoke, Gabor Antal, Csaba Nagy, Rudolf Ferenc, and Tibor 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*, pages 95–104. IEEE Computer Society, 2014. doi:10.1109/SCAM.2014.18.

[SE14] Daniela Steidl and Sebastian Eder. Prioritizing maintainability defects based on refactoring recommendations. In Chanchal K. Roy, Andrew Begel, and Leon Moonen, editors, *Proceedings of the 22nd International Conference on Program Comprehension*, pages 168–176. ACM, 2014. doi:10.1145/2597008.2597805.

[STV14] Danilo Silva, Ricardo Terra, and Marco Tulio Valente. Recommending automated extract method refactorings. In Chanchal K. Roy, Andrew Begel, and Leon Moonen, editors, *Proceedings of the 22nd International Conference on Program Comprehension*, pages 146–156. ACM, 2014. doi:10.1145/2597008.2597141.

[Sur] Web Technology Surveys. Usage of CSS for websites. URL: http://w3techs.com/technologies/details/ce-css/all/all [cited 15-03-2015].