Modular CSS in practice

Modular and scalable CSS and CSS pre-processors enable web developers to write more efficient, performant and reusable code

Second Bachelor Thesis

Media Technology degree course
St. Pölten University of Applied Sciences

Completed by:
Lisa Gringl
mt101028

Under the supervision of: Dipl.Ing. Klaus Temper

St.Pölten, 03.06.2013

Declaration

The attached research paper is my own, original work undertaken in partial fulfilment of my degree. I have made no use of sources, materials or assistance other than those which have been openly and fully acknowledged in the text.

If any part of another person's work has been quoted, this either appears in inverted commas or (if beyond a few lines) is indented. Any direct quotation or source of ideas has been identified in the text by author, date, and page number(s) immediately after such an item, and full details are provided in a reference list at the end of the text.

I understand that any breach of the fair practice regulations may result in a mark of zero for this research paper and that it could also involve other repercussions.

.....	.....
St.Pölten, on	Signature Author

Abstract

This thesis deals with modular and scalable Cascading Style Sheets in addition with CSS pre-processors. It gives an introduction in the meaning and the similarities of Object Orientated CSS and scalable/modular CSS Architecture. Besides a theoretical introduction to the CSS pre-processors LESS and Sass/Compass this works explains their components and functions by taking Sass/Scss as example. Development environments for coding have been tested on support, handling and other issues of Sass and LESS and showed that CSS pre-processors get more and more adopted by the coding tools developer. Additionally there is a review of some utility programs for writing and compiling modular CSS with CSS pre-processors. To cover the performance issue of web pages a test web project has been created with three different CSS architectures: plain CSS, modular CSS and modular CSS including Scss which comprises the most comprehensive part of the thesis. This work tests the performance with the Google Chrome Network developer tool and the affectivity of modular CSS by analysing these three projects. The paper demonstrates that modular CSS and modular CSS including CSS pre-processors can have a positive effect on performance and file size compared to plain CSS.

Kurzfassung

Diese wissenschaftliche Arbeit beschäftigt sich mit der Modularität der Stylesheetsprache CSS im Zusammenhang mit CSS Vorverarbeitungsprozessoren. Es wird eine Einführung in die Bedeutung von Objektorientierten CSS, sowie in modulare und skalierbare CSS Architektur vorgestellt. Neben einer theoretischen Erläuterung in die CSS Vorverarbeitungsprozessoren LESS und Sass/Compass, sollen die Komponenten und die Funktionen dieser Prozessoren am Beispiel von Sass/Scss gezeigt werden. Es wurden Entwicklungsumgebungen für das Schreiben von CSS Code sowie Hilfsprogramme für das Kompilieren der Vorverarbeitungsprozessoren, anhand von verschiedenen Kriterien analysiert. Dabei stellte sich heraus, dass immer mehr Entwicklungsumgebungen sich auf die Integration von Vorverarbeitungsprozessoren spezialisieren. Um herauszufinden welchen Einfluss modulare CSS Architektur auf die Performance einer Website hat, wurde ein Webprojekte mit drei unterschiedlicher CSS Architekturen erstellt: Standard CSS, modulares CSS und modulares CSS mit Vorverarbeitungsprozessoren. Diese Arbeit zeigt, dass modulares CSS sowie modulares CSS mit Vorverarbeitungsprozessoren einen positiven Einfluss auf die Performance und die Dateigröße haben.

Table of Contents

1 Introduction

There are new technologies and trends in web design and development on a day-to-day basis. The more new technologies are being invented the more is possible and thus the more detail implementation is behind modern and large websites. New technologies and possibilities lead to more lines of code. To maintain these additional lines new and adjusted workflows for the development and design sections are needed. A rising trend is to write modular CSS code for websites. This kind of code is easier to maintain and to reuse later. This trend was first taken up by Nicole Sullivan (*Nicole Sullivan (stubbornella) Object Oriented CSS at 2009 Fronteers*, 2012) in 2009 with the introduction of OOCSS (= object orientated CSS).

To get a clearer code for working in an bigger development team or maintaining an older project later, modular CSS structures the code in different parts. In order to lend additionally the advantages of dynamic code to modular CSS there are some pre-processor languages which make coding easier. Especially through responsive web design the lines of code have doubled in nearly every project. With the pre-processor languages it is possible again to reduce the lines of code to a breakdown. This work specialized on Sass/Scss and LESS, because these are the most used pre processor languages.(“Poll Results: Popularity of CSS Preprocessors | CSS-Tricks,” n.d.)

This work sets out to answer the following questions:
How suitable is modular CSS for daily use? How does modular CSS affect performance in real world scenarios?

The hypothesis of this work is:
Modular CSS code in addition with CSS pre-processors provides a better performance than “plain“ CSS code.

To answer the questions and prove or falsify the hypothesis the line of action will be to rebuild an old non-modular website with the same design in modular CSS and afterwards a pre-processor language (Scss) will be added to the modular code. The three different files will be tested on size, length and performance.

Furthermore development environments and utility programs for modular CSS and CSS pre-processor will be reviewed and tested on suitability for daily use.

This work starts with comparing former best “plain“ CSS practises with the basics of modular CSS. The second chapter gives an overview about the accruement and the principles of modular and scalable CSS and the influence of OOCSS to this topic, together with showing up the similarities between OOCSS and the rudiments of Jonathan Snook’s „Scalable and Modular Architecture for CSS “. Furthermore CSS pre-processors and their basics will be explained, by taking the example of Sass. After explaining the different methods in theory, the practical part then will show how modular CSS and CSS pre-processors work in real projects. Once three different CSS versions have been constructed, different parameters will be tested to detect how effective these methods of writing CSS code are. The last chapter gives an overview about well-established development environments and some utility programs for working with CSS pre-processors.

2 Common state and problems of plain CSS

CSS (=Cascading Style Sheet) is a static language for describing the appearance of a web page. A big step for mark up and re-usability of CSS was the W3C recommendation of separating HTML and CSS in the Cascading Style Sheets W3C Recommendation level 1 from the 17th of December 1996. This was the first step to more modular CSS (Wium Lie, Elika J.Etemad, & Tab Atkins Jr., 2013).

Unfortunately to this day CSS has been very difficult to scale in bigger web projects. The more comprehensive the web projects are, the bigger the code and file sizes get. Furthermore, on big projects, many people might work together, and sometimes the ones who have to maintain a web project are not the same as the ones who initially created it. So the problem is that a desired modification of appearance in a finished webproject is often made by someone with other coding behaviour. The usual approach is to search for the selector in the HTML and then write a new CSS rule at the end of the CSS document. So the code will become more and more redundant.(*Nicole Sullivan (stubbornella) Object Oriented CSS at 2009 Fronteers*, 2012).

Best practices are recommended guidelines which are not really published by someone, but rather developed by some gurus from the web area and the remaining community by taking their yearlong experience into consideration. Shay Howe mentioned some of these earlier best practices for writing CSS in his talk about Front End Legos (Howe, 2013). These best practices aren't valid anymore when writing scalable and modular CSS because these earlier best practices complicated changes which is bad for modular CSS. Two of these earlier best practices he mentioned are:

- Extra classes and extra elements should be avoided
- Leverage type and descendent selectors should be used

Type selectors are selectors that describe a special HMTL element like `ul`, `div` or `p`.

```
ul {
Style property: value;
}
```

Listing 1: type selector

Descendent selectors are rows of selectors with a special origin. In the example below there is a style rule defined for a `li` element, which is contained in an `ul` and the `ul` is contained in a `div`. Only if the parentage is correct in the syntax of HTML, the CSS rule will be applied to the li-element.

```
div ul li {
          Style property: value;
}
```

Listing 2: Descendent selector

Another rule brings along another problem with CSS: High specificity beats low specificity. If declaring a CSS rule for a `li` element and also another rule for the above mentioned div `ul li` constellation in the same CSS document, the second rule is stronger than the first one because the second has a higher specification.

```
li {
   font-size: 1.0em; // low specifity
}

div ul li {
   font-size: 1.2em; //styling rule with higher specifity
}
```

Listing 3: Specification of styling rules

This leads to this finding: When extra classes or elements should be avoided and descendent selectors should be used instead, how is it then possible to ensure that the CSS will not bloat? For this reason the earlier best practices are not effective anymore, real modular CSS should be written.

Furthermore specification of CSS is often really deep and declared in combination with ID selectors. This restricts the reusability of code classes, because the styling rules then can be only applied to an element if it is directly in the same specification structure. Instead of high specification, also called nesting (e.g. `#main .me ul li`), more classes in modular CSS should be used (Howe, 2013).

```
.class .class1 .class2 {
   font-size: 1.0em; // nested rule
}
.class2 {
   font-size: 1.0em; // no-nested rule
}
```

Listing 4: CSS nesting

3 The idea of scalable and modular CSS

One of the most popular books regarding modular CSS is “Scalable and Modular Architecture for CSS“ from Jonathan Snook written in 2012. The book includes guidelines for creating scalable and modular CSS code. The idea of modular CSS was born earlier, before this book and any of the later mentioned CSS preprocessors where invented. The one person who has initialized and shaped this modular movement was Nicolle Sullivan with introducing OOCSS. This chapter will explain how the principles of OOCSS work and how these principles lead up to that kind of modular CSS, Jonathan Snook is talking about.

3.1 Introduction to OOCSS

Nicole Sullivan first presented the idea of object orientated CSS in 2009 at the conference “Web Directions North” (*Nicole Sullivan (stubbornella) Object Oriented CSS at 2009 Fronteers*, 2012). The aim of OOCSS is to make the code smaller, compacter and more efficient. The first step to make CSS more performant is to reduce the number of code lines, if this is possible. The next step is to take these fewer lines of code and make them reusable. OOCSS relies on these two steps. Furthermore there are two main principles of OOCSS which should be followed:

1. Separate structure from skin and
2. Separate container from content

Before a web project can be started it must be decided which components should be used. Nicole compares these components with lego.

> *“You put together the single stones to get a lego house.” (*Nicole Sullivan (stubbornella) Object Oriented CSS at 2009 Fronteers*, 2012)*

Single stones are needed to start. This means that first a component library must be created, which contains content objects like headings and lists and extended

media objects. The aim is to write the specification once and then reuse it as often as possible. This approach has a positive effect on performance and quality of code.

The components are the basic rules for the look and feel of the website. If an exception is needed, an existing object has to be extended with additional parts.

```
<div class=”media media_ext”>
        <video> </video>
</div>
```

Listing 5: OOCSS class extenstion

The Video object belongs to the media component but has additional features which are defined at the class media_ext. The principle of this way of development is never repeating but extending. The modules must be unique, i.e, if two modules look too similar on one page you have to choose one. Furthermore, all objects should be flexible in width and height so that you can use all objects in a grid and the components expand to the outer container. In addition to that, it is very important that you do not specify the element which the class is into:

```
div.myerror { /* should be avoided */
   width: 300px;
   color:red;
}

.myerror{/ * should be favored*/
   width: 300px;
   color: red;
}
```

Listing 6: Specification in OOCSS

So it should be avoided that the class can’t be reused in other elements and the code remains as open and flexible as possible.

If all components are ready, the skin can be created. The skin specifies how the modules look like in a particular case.

The main goals of OOCSS are that mark up in CSS will become predictable and the code will become reusable for better performance and less redundancy.

3.2 The principles of a scalable and modular architecture

To produce modular CSS code, it should be separated in different parts to ensure easy maintenance. Jonathan Snook's core idea is to separate the CSS in the following 5 parts: every part has its own coding guidelines (Snook, 2012, p. 4):

1.Base
2.Layout
3.Module
4.State
5.Theme

3.2.1 The 5 main CSS parts of modular CSS

The base category only includes single element selectors, no classes, no IDs. It's only for setting default styling rules to the elements. The use of `!important` should be avoided completely in the base category. Additionally a self-written CSS reset should be included in this category, but reset frameworks like YUI CSS Reset ("CSS Reset - YUI Library," n.d.) should be abandoned because they are often bloated (Snook, 2012, pp. 8–9).

The layout part includes only styles for layout definition. Layout elements are major elements like footer, header or article. These major elements include minor elements – the so-called modules like sidebars, button, login boxes. The layout elements itself can also be categorized in major a minor parts. A major part is an element that is almost always only once used in the HTML code, e.g header or footer. A minor element, e.g. a div container, which contains some modules, is more often used. Layout elements are the only type of elements where IDs as selectors should be used (Snook, 2012, pp. 10–12).

The module part is the most important one. Modules are more discrete parts of the page. Every module works as an independent part of the CSS which can be put in every part of the document when needed. IDs should be avoided and only class names, with low specifity, should be used. If you use an element selector, then only use one that includes semantics. `Div` and `span` elements, for example, are elements without sematic. If the modules should get extra properties, sub-classes should be made. (Snook, 2012, pp. 19–24).

State rules include CSS declarations for state changes with JavaScript only. A state style overrides all other properties and values. A module and a state rule is very similar, but nevertheless different, because state styles can be applied to

layout and module parts. The state part is the only one where the use of `!important` isn't forbidden. The reason behind is that the state rules often has to override other styling rules in complex system, nevertheless the `!important` attribute should only be used if it is a absolutely necessary (Snook, 2012, pp. 25–27).

Theme rules describe the look and feel of the web page, the layout and the modules. Putting colors and images in an own file makes it easy to reuse and change the theme later. In a theme default values of every of the last 4 parts mentioned above can be overridden (Snook, 2012, pp. 28–29).

3.2.2 Naming convention

Often developers working on a project don't think about the fact that someone else maybe maintaining the code later. This ends up with confusing or meaningless class names. In case of modular CSS it is very important that classes have understandable names. The names should stand in an understandable relationship with the HTML content. So if the HTML content is broken down in categories, which is recommended for modular CSS and which will be explained in the OOCSS chapter, the classes should have some prefixes. For example, state classes should have the prefix is (is-hidden, is-active) and layout rules the prefix l (l-wrapper,l-footer). Naming conventions are very important to find the styling rules again later in a project (Snook, 2012, p. 7).

Nicolle Sullivan introduced the idea of a CSS skin, separated from the rest of the components. Jonathan Snook took this idea of a skin and separated it again into the Theme and the Layout rules. Furthermore all the guidelines for modules are very similar to the components approach from Nicolle Sullivan. Even the extended method from Nicolle Sullivan looks the same as Jonathans Snook's version.

Besides these two main authors in this CSS architecture field, there are many others who write, blog and research in this field. Therefore this work tries to cover the best guidelines for modular CSS from various sources. There is no "right way" to write modular CSS. Developers can only try to get a good overview of the different viewpoints and techniques and apply the appropriate guidelines to each particular project. The practical part of this work will try to incorporate all described research findings.

4 CSS pre-processors

Normally CSS code is static and CSS pre-processors make CSS code dynamic. This brings advantages for developers and makes the code more reusable. CSS pre-processors make the code easier to maintain and to modify. That leads up to an increase of the code quality (Kennedy & León, 2011, p. 261). A poll result on a popular CSS blog with about 13.000 attendees, shows that the two main used CSS pre-processors are SASS (which includes SCSS) with 18% and LESS with 23% ("Poll Results: Popularity of CSS Preprocessors | CSS-Tricks," n.d.). This chapter gives a short overview about the two main CSS pre-processors and the principles of pre-processors by taking the example of Sass, because Sass is the senior of the two languages and offers more functions as of now.

4.1 Sass/Compass

Sass stands for Syntactically Awesome Stylesheets and was invented by Hampton Catlin. The primary developers are Nathan Weizenbaum and Chris Eppstein. Sass is older than LESS, it is written in its 3rd version. In the first version of Sass, the syntax was similar as the Python one. That means the old syntax (.sass) was based on indentation and carriage returns, while now the syntax of Sass is a valid CSS file with braces and semicolons, which is called Sassy CSS (.SCSS). Sass includes a lot more functions then LESS, but both were originally built in Ruby (Kennedy & León, 2011, p. 279).

Some people are confused about the difference of Sass and Compass. Compass is no real pre-processor; it is a kind of "extension" to Sass. It is a design-agnostic open source framework that offers often used functions, mixins and variables in Sass code. If, for example, a border-radius is needed, compass offers a built-in mixin called `border-radius()` with a default value of 5px which can be used. The mixin returns the border-radius attribute with all browser pre-fixes. To get the best out of pre-processors Sass and Compass should always be combined ("Compass Core Framework | Compass Documentation," n.d.).

4.2 LESS

LESS has been invented by Alexis Sellier and is now in Version 1.4.0 Beta. The first version was like Sass written in Ruby, but now based on JavaScript. The biggest difference between Sass and LESS is that LESS can be compiled server-side with Node.js or client-side directly in the browser, but only with modern ones, like Firefox, Chrome and IE9+ ("LESS « The Dynamic Stylesheet language," n.d.). The worldwide amount of IE8 and above users is around 12%. The more modern browsers, like Chrome and Firefox are used by already 56% of the world's population (Chrome = 36,46%, Firefox 5+ 20.58) ("Top 12 Browser Versions (Partially Combined) from Oct 2012 to Mar 2013 | StatCounter Global Stats," n.d.). Furthermore, it must be mentioned that a lot of blog posts and issues at the "LESS github repository" point out and recommend compiling the LESS code into normal CSS before using it at a real life project.

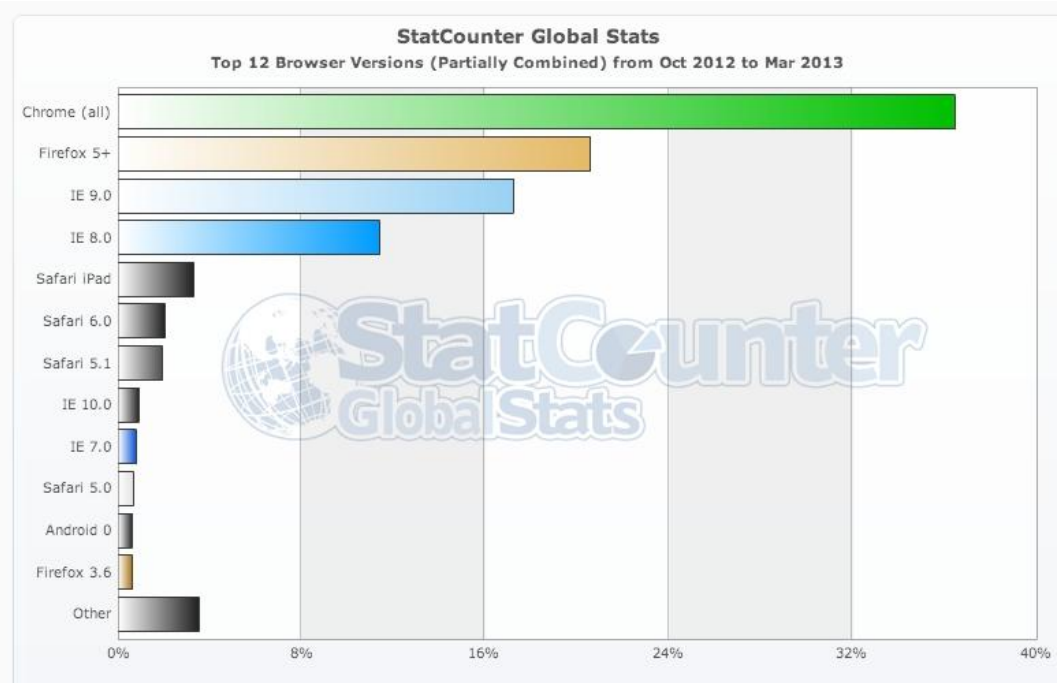

Figure 1: Current browser statistics("Top 12 Browser Versions (Partially Combined) from Oct 2012 to Mar 2013 | StatCounter Global Stats," n.d.)

4.3 Explaining principles of pre-processors by taking the example of Sass

Every CSS pre-processors consist of some core parts which distinguish in syntax, but offer almost the same functions. This work is explaining the principles of CSS pre-processors by taking the example of Sass because it was the first well-known pre-processor language and because it has constantly been further developed. The following contents refer to the Sass documentation itself ("Sass - Syntactically Awesome Stylesheets," n.d.-a) and the Pragramtic Guide to Sass written from Hampton Catlin (Catlin, 2011), the inventor of Sass.

4.3.1 Installation and compiling

To get Sass working on Windows and Linux computers, Ruby has to be installed first. On Mac OS X this is already installed as part of the system. To create Sass in Sassy CSS syntax, simply change the file extension to .scss. If Ruby is installed Sass can be easily added to the system via the command line or the terminal with this command:

```
gem install sass
```

Listing 7: Sass installation command

To simplify the working process all Sass files should be in one folder. If there is, for example, a layout.scss file in the Sass folder, this file has to be compiled in the terminal/command line before it can be opened in the browser because unfortunately there has been no direct Sass file browser support until now. To compile the file the terminal has to be navigated to the directory where the Sass file is. Afterwards the single file can be compiled with the command *sass [filename.scss] [filename.css]*. As it would be annoying to compile every Sass file after every small change, there is an inbuilt function which allows the console to track Sass code changes in a defined folder. Every time a change is tracked, the console automatically compiles the Sass files to new CSS files (Catlin, 2011, pp. 6–7).

```
sass layout.scss layout.css // compiles scss to CSS
sass –watch stylesheets/sass:stylesheets/css // tracks code changes in
the defined folder
```

Listing 8: Compiling commands in Sass

Because Sass is based on Ruby it is easy to use it with the web framework. The implementation is explained in the "Pragramtic Guide to Sass" from the Sass

inventor himself Hampton Catlin. This work specializes on the new Sassy CSS because it is the newer one and easier to use.

4.3.2 Variables

To tackle the problem of redundant code changes in different code snippets in a document Sass introduced variables for CSS. The variable must be declared and initialized once, and then it can be reused as often as wanted. If the value of the variable has to be changed, it only has to be made once at the initialization point. A variable always starts with a $-sign followed by the variable name, a ":" and then the value which should be assigned to the variable. Variables could contain colors, size, percentages and nearly every other value, which can be given to a style property. Similar to other programming languages, the variables can have two different types of scope. They can be defined globally, which means they are defined once, usually at the top of a document and then can be used everywhere in the code. The second type of variables is the scoped one which can only be used in the selectors it was initialized in and its child selectors. Furthermore a variable can be set to a default with `!default`. If there is no other value assigned to a specific variable the default value will be used.

```
$color_main: #000000 !default; // global variable set black to he de-
fault value
$container_width: em(800);

.blue {
   color: $color_main;
}

#wrapper{
   width: $container_width;
   color: $color_main; // the output is the default black, because no
other global value for this variable found
   article{
      $color_main: # 1add00; //set $color_main to green
      color: $color_main; // the output is green because there is a new
value found for $color_main
      div{
         .blue; /* text in this container will be black, because the
definition of color main not counts for an embeded class, so the
color_main variable will use the default value */
      }
   }
   div{
      $bg_color: # dbdbdb; //scoped variable
      background-color: $bg_color;
           article{
```

```
              background-color: $bg_color; //can be used in the child
selectors
           }
   }
}
```

Listing 9: Definition and scope of variables in Sass

4.3.3 Comments

It is important to make understandable comments to make it easier for other developers to work with the code. In Sass there is a possibility to write normal CSS comments and so-called “silent comments“ – this means that users don’t see the Sass comments in the compiled CSS. Furthermore there is a possibility to set the Sass output mode to compressed – this mode strips out all comments in the CSS file, even the normal multiline CSS comments. If this should be avoided, you can force the file during compilation to keep some of the comments with an exclamation mark. The different types of comments look like this (“Sass - Syntactically Awesome Stylesheets,” n.d.-b):

```
//This is a silent comment
/* This is a normal CSS comment */
/*! This is a normal CSS comment, which will not be stripped while
compiling in the compressed mode */
```

4.3.4 Nested Rules

To avoid the usual repetition of CSS, Sass uses nested rules to embed styling rules into each other. Nesting can be realized with selectors and with CSS properties. Moreover, nesting rules make code more readable because the tree organisation is clearer and more logical.

```
table.hl {
     margin: 2em 0;
        td.ln {
           text-align: right;
     } }

li {
   a{ color:red;
      &:hover{
      text-decoration:none;
      }
   }
font: {
     family: serif;
```

```
      weight: bold;
      size: 1.2em;    }
}
```

Listing 10: Nesting

Furthermore, there is the possibility to use an ampersand (=&) in nested rules for selector declaration. The ampersand works the same way as the other selectors, but it is usually used for pseudo-classes like `&:hover` or `&:visited`, which produces the following after compilation in CSS ("Sass - Syntactically Awesome Stylesheets," n.d.-b):

```
li a:hover {
   text-decoration:none;
}
```

Listing 11: Using the ampersand for nested rules

On the one hand working with nested rules is a big advantage because it saves time writing the whole specification only once, but on the other hand this easy way of embedding elements in each other quickly produces non-modular CSS code. The nested Sass code might look more compact and cleaner but the compiled CSS code is often bloated.

There is a rule, mentioned by Nicolle Sullivan at Smashing Conference 2012 in Freiburg – The Inception Rule: never go more than three levels deep with nesting in SASS (Sullivan, 2012). For modular CSS the specification should be as low as possible and if a higher specification is needed, this specification shouldn't be deeper than three levels. This rule should be kept in mind if nested rules are used in Sass.

4.3.5 Mixins

Mixins are the most powerful and modular parts of Sass. They give developers the possibility to reuse full parts of the code, properties or selectors. In addition, it is possible to pass parameters to mixins. Mixins are very effective in reducing duplication of code. The bigger the project is the more advantages mixins bring into a project. To use mixins in parts of your CSS code you first have to declare the mixin and then include it in a style rule:

```
@mixin button_style($color_border,$color,$text_color) {
   border-bottom: 5px solid $color_border;
   background-color: $color;
   color: $text_color;
}
```

```
.button {
   text-decoration: none;
   text-align: center;
   margin-top: 5px;
   @include button_style($dark_grey,$grey,white);
}
```

Listing 12: Mixins

Mixins in Sass works similar to functions in other programming languages. An advantage is that Sass recognizes which mixins are really used by the HTML and only compiles those parts. The `@include` keyword starts the mixin inclusion followed by the name of the mixin. Every time when the `@include` element is applied to a CSS Style, Sass will bring the code snippets of the appropriate mixin to the right rule in the compiled CSS.

The more often projects are built with Sass the more different mixin parts can be produced to put together to an own mixin library. Moreover it is also possible to include functions in mixins to calculate, e.g. color values or other metrics. With modularity in mind it is helpful to put the mixin library in a separated SCSS file as a component to easily find and edit the mixins later.

4.3.6 Importing

According to the basic principles of modular CSS it is important to have different parts of CSS separated of each other to maintain clarity. To bring together these different parts they must be imported into the main file with @import followed by the URI of the included style sheet (Wium Lie et al., 2013, p. 3). The problem is that this might be very helpful and seems to be modular, but the @import element should be avoided in real life projects because it really slows down the page speed. If an @import directive is being used, the browser will not be able to download all stylesheets in parallel, but rather first have to download, parse and execute the “parent“ stylesheet before the browser recognizes that it has to download some imported “child“ stylesheets(“Minimize round-trip times - Make the Web Faster — Google Developers,” n.d.).

In Sass the syntax and the approach is nearly the same, but with one big advantage. If Sass is not supposed to produce corresponding CSS files when using the @import directive, you only have to provide the filenames with an underscore at the beginning. Sass then treats the stylesheets as partials and produces only one stylesheet at compiling. Furthermore, any mixins, variables or functions defined

can be used in every other stylesheet partials, only the order of the imported stylesheets is important(Catlin, 2011, p. 26).

```
@import("core/reset");//this will creates extra stylesheet

@import("core/_variables"); //partial
@import("core/_functions"); //partial
@import("core/_mixins"); //partial
@import("core/_layout"); //partial
@import("core/_ui"); //partial
@import("core/_responsive"); //partial
```

Listing 13: @import directives in Sass

4.3.7 Selector Inheritance

As mentioned in chapter 3.2, naming conventions are very important. Not only the prefixes for the modules are important, but also the logical naming of the classes or IDs. It's important to name objects based on what they do (.login_button), and not what they look like (.green_button). But in the case of using Sass you should have one element with basic attributes and use these attributes in various styling rules by keeping good semantics. With `@extend`, values and attributes get cloned from one class or ID to another. Actually the `@extend` directive isn't more than a copy of the attributes and values of a class or an ID. The big advantage is that if the button or the element should get a change, only the actual class must be changed and not every code snippet where the class is extended. It works similar to inheritance in other programming languages. The extended class gets the attributes from the parent class in addition to its own new attributes.

```
.green_button {
     background-color: $green;
     @include border_radius(5px); // using the border-radius mixin
     font-size: em(28); // using the pixel-to-em function
  }

.login_button {
  @extend .green_button;
}

.submit_button {
  @extend .green_button;
  color: $black;
}
```

Listing 14: @extend directive in Sass

This looks very similar to the mixins, but there are two main differences. Parameters can be passed to mixins, while this is not possible for parent classes. But the main difference becomes obvious after compiling the Sass code into CSS. While Mixins only add additional code to a defined CSS selector, the @extend directive not only merges all attributes and values, but rather merges the list of the selectors (Catlin, 2011, p. 36). The compiled code looks like this:

```
.green_button, .login_button, .submit_button {
   background-color: #119400;
    -moz-border-radius: 5px;
-webkit-border-radius: 5px;
        border-radius: 5px;
   font-size: 2em;
}

.submit_button {
   color: #000000; // code must be checked
}
```

Listing 15: compiled @extend code

4.3.8 Functions

There are some predefined functions included into the Sass library. The most popular and easy predefined functions are the color functions `lighten()` and `darken()`. These functions can be passed over colors and given the percentage which the color should be changed.

```
lighten($color, 20%);
darken($color, 20%);
```

Listing 16: Color functions included in the Sass library

These two functions are part of the HSL (=*hue*, *saturation*, and *lightness*) function components of the Sass library. Furthermore, there are RGB functions, opacity functions and some other color functions included in Sass. All these functions help to create color themes only with code, without the help of any other tools.

Like in other programming languages Sass also has some number functions, which help to round values or return a maximum or minimum value. The function structure is relatively simple and similar to other languages.

```
Round($value) //rounds the number
Min($xi, $x2, $x3,..) //gets the minimum value of the passed parame-
ters
Max(xi, $x2, $x3,..)) //gets the maximum value of the passed parament-
ers
```

Listing 17: Some number functions

Furthermore, there are some list, introspectation and miscellaneous functions predefined in Sass.

In addition to that, own functions can also be defined in Sass. The documentation explains how to add functions by adding Ruby methods to Sass. This work will concentrate on describing how to add functions to Sass simply with SCSS in the same document where the function then will be used.

The following function serves to translate a pixel value into the appropriate em value. A function is declared with @function followed by the function name. Every function can have some arguments passed over. To get the em of a pixel value the formula is: Divide the wished value by the base value. The function below has two parameters, the first one is the pixel value which is passed to the function when called and the second one is the base pixel value which is defined as a global variable at the beginning. When the function is called, it returns an em value to the styling rule because the result of the explained calculation is multiplied with 1em. The function isn't only for the purpose of font-size; it could be applied to every pixel value.

```
$default-font-size: 16px;

@function em($px, $base: $default-font-size) {
   @return($px / $base)*1em;
}

h1 {
   font-size: em(32);
   padding: em(10);
}

@media only screen (max-widht: em(320)){

   h1 {
   font-size: em(24);
   }

}
```

Listing 18: declaration and usage of an own function

If the size-ratio of a site should be changed, only the $default-font-size value must be changed and every value calculated with the em function changes, too. Functions bring more flexibility and modularity into SCSS. The full list of functions

and the declaration of functions by adding ruby methods is available in the Sass Documentation (“Module: Sass::Script::Functions,” n.d.).

5 Code examples in plain CSS modular CSS and SASS

This part includes a responsive website built once in plain CSS, once in modular CSS Architecture and once in modular CSS extended with SASS. To get an applicable result the author of this work rebuilt a non-modular website, created in August 2012, for the web conference and barcamp Digital Visions. The reason for this practice is to ensure an unbiased result, because of the intensive involvement in the topic of modular CSS otherwise the author couldn't guarantee to write honest plain CSS anymore. The website was created as a responsive one-pager and included various CSS3 features like box-shadows, border-radius and transitions. This chapter explains the working procedure and the findings of the implementation of modular CSS.

5.1 Procedure

The coding procedure started with building a completely new HTML file from scratch design-based on the existing non-modular site. This was essential to get a more modular HTML structure and to work with the rules of modular CSS architecture. The new structure includes some layout containers, which again include the modular re-usable components. Components have their base classes and then also posses other extended classes. The next step was coding a new CSS structure and linking it to the HTML document. After finishing the part of creating plain but modular CSS, the same code was taken as basis for the third CSS example. Based on the new HTML structure and the modular CSS, Sass and Compass were merged at the right lines of code. At the end of the experiment the different files of code were compared to each other and analysed.

5.1.1 Modular CSS

There are now six new CSS files in the CSS folder: one for the base layout (base.css), one for additional layout rules (layout.css), one for the components (module.css), one for the state rules (state.css), one for additional shaping of the

theme (theme.css) and one for the responsive changes (responsive.css). It must be mentioned that in this case the state.css remained empty because no JavaScript function has been implemented. Furthermore, the author decided to create an extra file for the responsive layout, because it is easier to maintain the different media queries. The different CSS files are included in the header section of the document to make use of parallel download in the browser.

```
<!-- including all Stylesheets -->
  <link rel="stylesheet" href="css/base.css">
  <link rel="stylesheet" href="css/module.css">
  <link rel="stylesheet" href="css/state.css">
  <link rel="stylesheet" href="css/theme.css">
  <link rel="stylesheet" href="css/layout.css">
  <link rel="stylesheet" href="css/responsive.css">
```

Listing 19: Including modular CSS in HTML

The following example shows the new structure of modular components. In the project these boxes show the included services at the conference. The single boxes are list-items which are included in an unordered list. This list is positioned in the layout container `l-wrapper_small`.

```
<div class="l-wrapper_small">
      <ul class="right">
        <li class="detail_box b_left b_left-orange">Jause und
Lunch</li>
        <li class="detail_box b_left b_left-dgrey">gemütlicher
Austausch</li>
        <li class="detail_box b_left b_left-lgrey">viele nette Leute
kennenlernen</li>
      </ul>


      <ul>
        <li class="detail_box b_left b_left-dgrey">keine Teilnahmege-
bühren</li>
        <li class="detail_box b_left b_left-lgrey">von 9.00 -
17.00</li>
        <li class="detail_box b_left b_left-orange">Getränke
inkludiert</li>
      </ul>
  </div>
```

Listing 20: HTML in modular Code

The boxes belong to the detail_box component. This component is very abstract and can be reused for other elements, regardless of whether they are list items or not. The CSS for this module defines text-align, padding, border-radius and box-shadow - nothing that could affect outer elements.

```
//_module.scss
.detail_box {
   text-align:left;
   padding:20px;
   -webkit-border-radius:5px;
      -moz-border-radius:5px;
           border-radius:5px;
   -webkit-box-shadow: 0 8px 6px -6px rgba(0,0,0,0.2);
      -moz-box-shadow: 0 8px 6px -6px rgba(0,0,0,0.2);
           box-shadow: 0 8px 6px -6px rgba(0,0,0,0.2);
   }
```

Listing 21: Module in modular CSS

To specify the detail box for this project an additional rule in the theme.css had been created. This rule includes the theme attributes like, e.g. color, background-image, margins and width. These attributes are likely to be different in every project and therefore related to the theme.

```
//_theme.scss
.detail_box{
    background-image:url(../img/check.png);
    background-repeat:no-repeat;
    width: 255px;
    margin-bottom:10px;
    background-position:96% center;
    background-color:white;
    color:black;
}
```

Listing 22: Theme specific attributes

This was the approach during the whole project time: creating modules, creating layout containers, putting the modules in the containers and giving them the appropriate skin. Sounds easy, but sometimes it is difficult to distinguish between modules attributes and others, and to decide what element is really a module. One of these difficulties was the 8px border at the left of the detail box. Actually it is only a detail which has no real reason to be a module. But the thing is that such decisions depend on the project. The author decided to include this class in the module.css because this class without the border-color could be used with

every HTML element and isn't appropriate to another part of the CSS. So the way to decide which element or class belongs to which CSS file is often a method of elimination.

```
*/ module.css
.b_left {
     border-left: 8px solid;
     }

*/ theme.css
.b_left.b_left-orange {
        border-color: #ee5a24;
        }

.b_left.b_left-dgrey{
        border-color: #202020;
        }

.b_left.b_left-lgrey{
        border-color: #c6c1bd;
        }
```

Listing 23: Module and extensions

5.1.2 Modular CSS with Sass

After the finished modular CSS, the project was duplicated with Sass and Compass added to it. To compile the written Sass code this project used LiveReload (= Real Time Compiling utility program). For configuration of the settings a ruby configuration file had been created, named config.rb, where the directory paths, the output styles and some other settings could be adjusted. In addition to that the CSS folder was renamed to sass and an additional stylesheet folder was created. The config.rb file helped Live Reload then to compile all Scss files in the sass folder to the same named CSS files in the stylesheets folder. To get only one final style.css file, a subdirectory, named core, was created and now contains 8 CSS partials which have been brought together in the one level higher style.scss file with the `@import` rule from Sass. It must be mentioned that there are two new parts of code now – variables and mixins. They should be treated as independent parts of Scss because it is easier to find or change variables and mixins again, if they are on one place and not scattered in the code. To make use of them in the most effective way, they have to be included at the beginning of the code, so that every other part could use them.

```
// importing all scss partials in style.scss
@import 'core/variables';
@import 'core/mixins';
@import 'core/base';
@import 'core/layout';
@import 'core/module';
@import 'core/state';
@import 'core/theme';
@import 'core/responsive';
```

Listing 24: Importing partials in one main scss file

Next the theme colours of the project have been replaced with the right variables declared in the _variables.scss. Furthermore, the Compass mixins has been used to replace some CSS3 features and the necessity of vendor-prefixes. At the end, the final compiled CSS file has been linked to the HTML document.

5.2 Results

After finishing all 3 projects file size of the code has been compared to each other uncompressed, compressed and compressed with gzip. The used tool for compressing the files was the online YUI compressor (“Online YUI Compressor,” n.d.). The file sizes show that the modular CSS code is the most compact one with only 15 KB in uncompressed version. The compiled modular CSS code created with Sass (= modular CSS+) and the plain CSS code only distinguish in 572 Byte. They both have a file size of about 25KB. The compression of the code reduces the file sizes of plain CSS about 25% and modular CSS about 26%. Modular CSS+ instead is reduced to about 55%. This leads to a minimal difference of 36 Byte between modular CSS and modular CSS+ in compressed version while the plain CSS is still about 7KB bigger than the others. The further compression of the code leads to a file size of 4344 Byte for plain CSS, 2977 Byte for modular CSS and 2937 Byte for modular CSS+. Furthermore, the files show that modular CSS saves nearly 200 lines of code instead of plain CSS. The modular CSS+ is nearly 30 lines of code smaller than the simple modular version, but in the compiled mode they then have exactly the same lines of code.

These numbers show that the modular CSS+ at the end is the smallest CSS of the three versions, but only with a not appreciable difference of 40 Byte. The plain CSS instead is approximately 1.5 times bigger than the other two versions.

Table 1: Comparing CSS file sizes

	plain CSS	***modular CSS***	***6 modular CSS with Sass (modular CSS+)***
File size (uncompressed)	*24.745 Byte (25 KB)*	*7 1.361 Byte* *8 1.541 Byte* *9 1.841 Byte* *10 1.133 Byte* *11 8.664 Byte* *12 = 14.540 Byte (15 KB)*	*24.173 Byte (25 KB)*
File size (compressed)	*18402 Byte (18 KB)* *-26 %*	*10928 Byte (11 KB)* *-25%*	*10964 Byte (11KB)* *-55%*
File size (compressed & gzip)	*4344 Byte (4 KB)* *-82%* *100%*	*2977 Byte (3KB)* *-80%* *68,53%*	*2937 Byte (3KB)* *-88%* *67, 61%*

13 Lines of code (without comments and empty lines)	*836*	*646*	*619 (uncompiled)* *646 (compiled)*

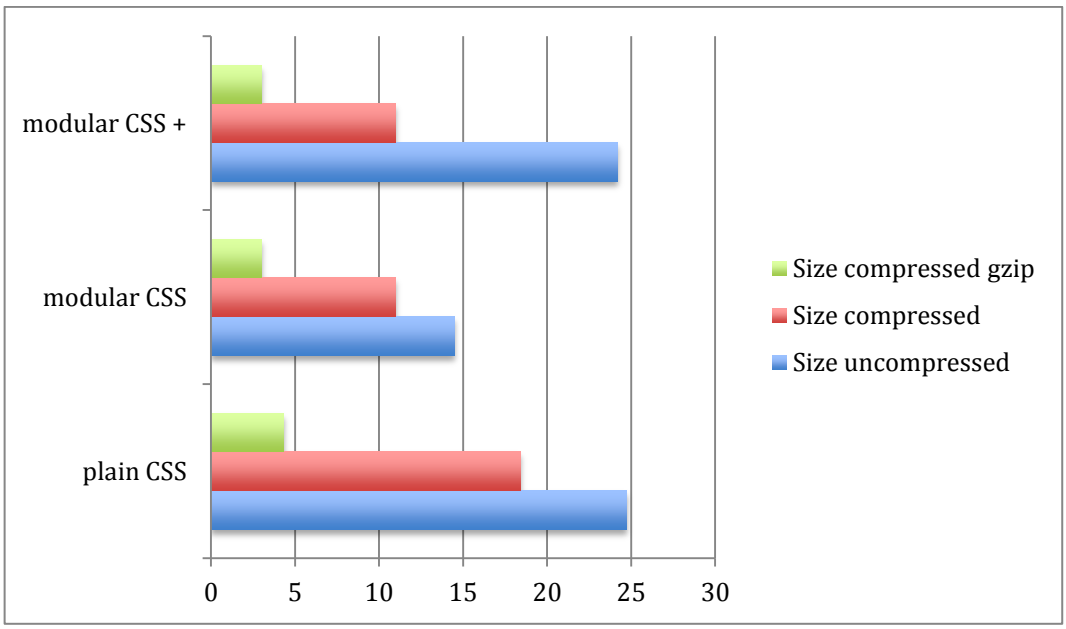

Figure 2: Filesizes of the CSS files in KB

14 Performance testing

This chapter is about performance testing and verification of modular CSS. Additionally the findings of performance testing on the website created in the last chapter will be presented in this chapter.

14.1 Grammar

To check normal CSS after writing there are two ways: validators and debuggers. These tools check if the CSS follows the CSS grammar. A validator is a tool that statically analyses the CSS while a debugger is a dynamical tool with which allows to edit, delete or introduce new style rules. But checking the grammar of the CSS is only one small part of optimizing CSS. It is not really possible to verify the CSS code of modularity. However, performance, which has to do with modularity of CSS can and should be checked (Geneves, Layaida, & Quint, 2012, pp. 809–810).

Not only after completing the CSS code it is important to check the grammar, during the writing process it is important, too. Some code editors have a built-in assistant for the syntax (syntax-highlighter) which helps to create grammatically correct style sheets. Furthermore, there are some plugins and programs that allow real time compiling during Code writing (Geneves et al., 2012, p. 810). These assistants and utility programs are very important, especially when writing modular CSS with the help of CSS pre-processors. Some of these utility programs and plugins will be introduced in chapter 7.

14.2 Performance

To get more performant modular CSS code duplication of code should be avoided. Furthermore, old code should be removed immediately if not used anymore. Moreover sprites, large images, which include various icons, and other graphics used on the site, should become standard for different content sections, e.g. a sprite with all social media icons for the contact section. Using caching and compression for common files is also very important (Howe, 2013). Some tools

for testing CSS speed are: CSS Lint, Inspector, PageSpeed and YSlow (Geneves et al., 2012, p. 810).

14.3 Performance test of the created project

The web project was loaded on a real live server from world4you and was tested with the network section of the Chrome developer tools. To get reliable test results the reloading without cache had been repeated ten times and then the average value was selected for the comparison of the three pages. The measured loading time represents the duration of loading all stylesheets in milliseconds.

The test shows that there is hardly any difference of loading time in all three versions at this project size. The difference of 8,9 milliseconds between the plain CSS and the modular CSS is immaterial in web loading times. Particularly, if the JavaScript, which indicates and shows one Google Map at the page and includes a little script for navigation has a size of 106.7 KB, it will need 1.5 seconds to load on average. The YSlow Plugin for Chrome (“Chrome Plugin - YSlow,” n.d.) shows the size distribution of all loaded sources which leads to the finding that the CSS loading times are only a small factor of performance optimization.

Table 2: Loading time of CSS files in milliseconds

Plain CSS	***Modular CSS***	***Modular CSS+***
68	*48*	*63*
47	*58*	*57*
72	*60*	*53*
118	*61*	*48*
50	*51*	*50*
76	*56*	*60*
52	*51*	*56*
54	*57*	*40*
54	*53*	*57*
51	*58*	*48*
64,2	***55,3***	***53,2***
100%	***86,1%***	***82,9%***
Standard variance in ms		
20,341	*4,1*	*6,46*

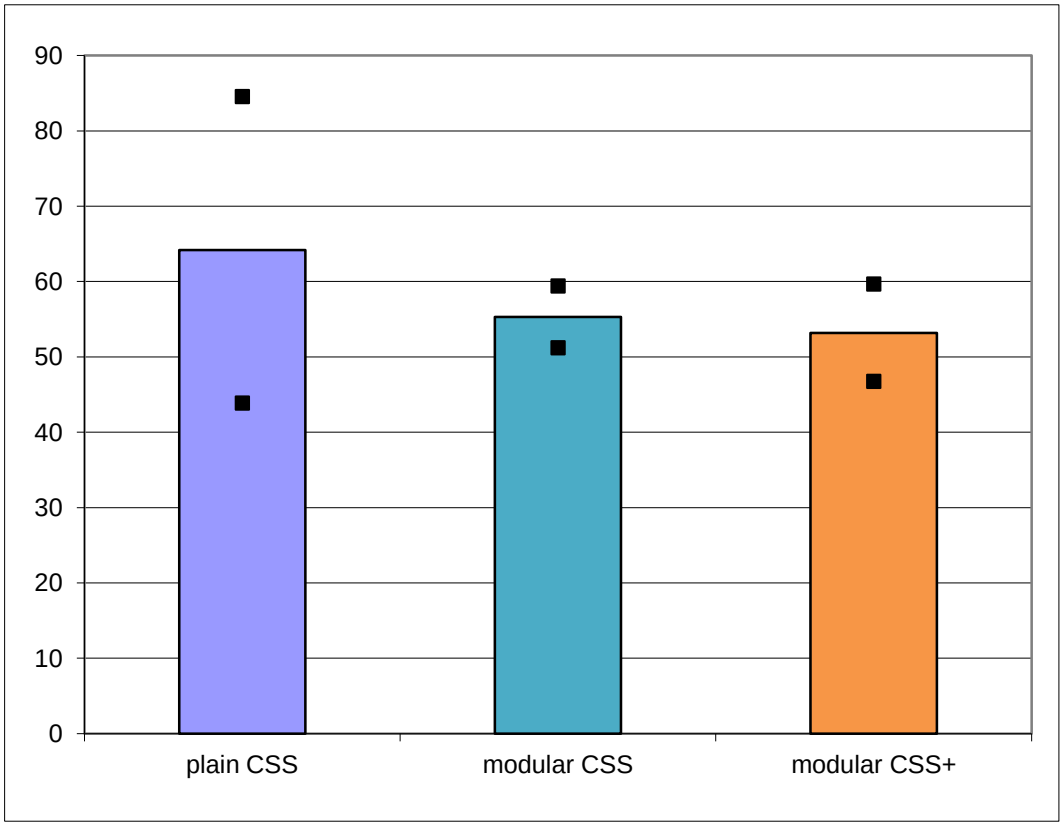

Figure 3: Loading times in ms with standard variance

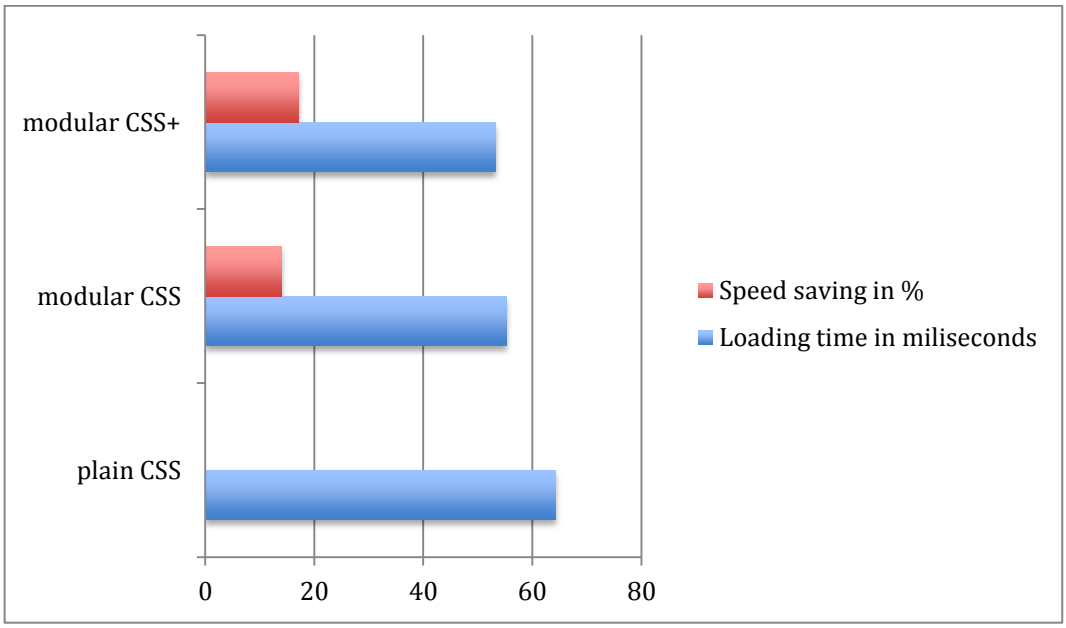

Figure 4: Bar chart of loading times in miliseconds

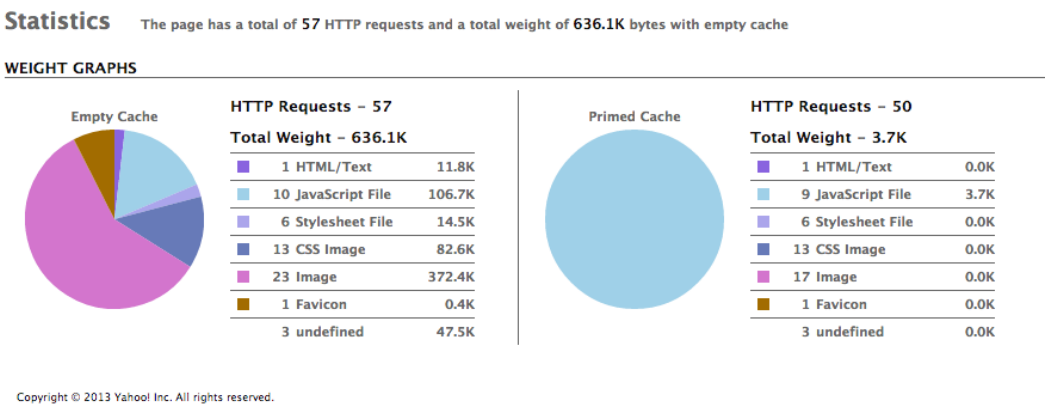

Figure 5: Size distribution modular CSS

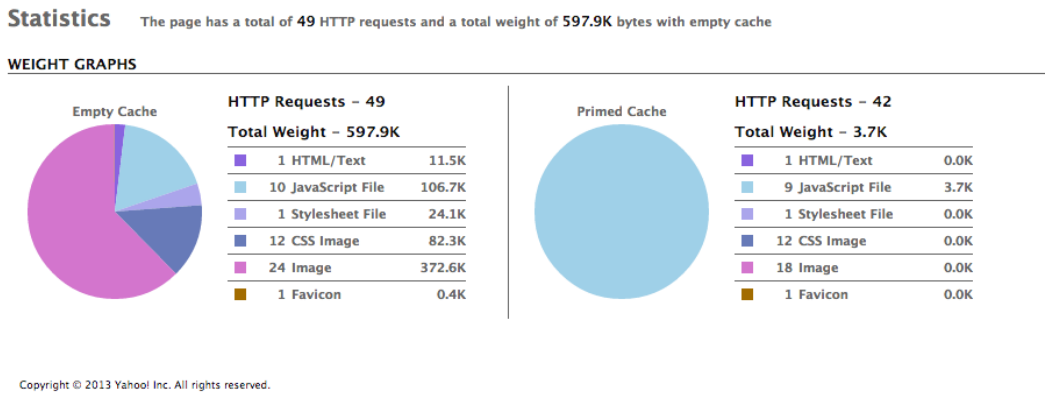

Figure 6: Size distribution modular CSS with Sass

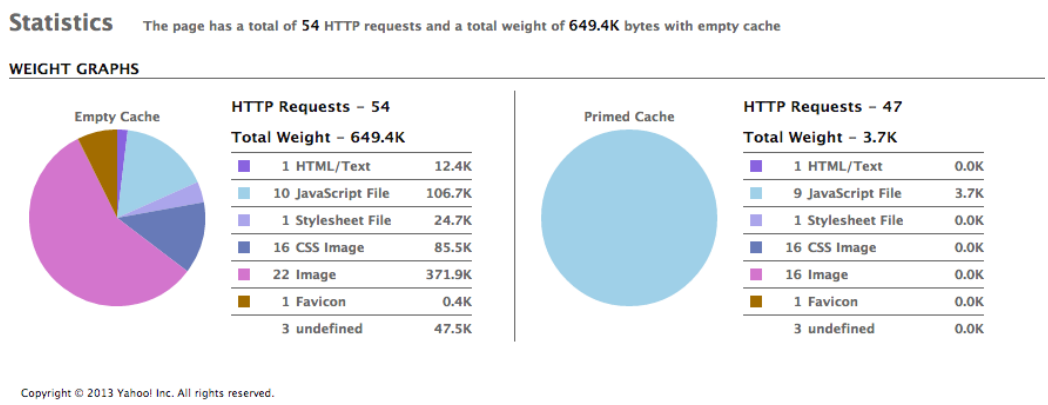

Figure 7: Size distribution plain CSS

15 Analysis of development environments for modular CSS

First of all, it must be mentioned that every development environment that is suitable for normal CSS is also suitable for modular CSS, because as mentioned in the description modular CSS has the same syntax only a different structure as common CSS. The choice of a development environment is only important if you use CSS pre-processors for writing modular CSS. Therefore you need special plugins to ensure that your coding environment gives you suitable syntax highlighting for the respective CSS pre-processor language. Furthermore, with the introduction of the CSS pre-processors some “utility” programs appeared which enable real-time compiling while writing CSS. These programs immediately give a fair warning when writing an error, so coding errors can be prevented. This chapter gives a short overview about plugins and extensions for commonly used development environments and shows up some special CSS pre-processor environments. Furthermore some “utility” programs will be introduced.

15.1 Text Editors

The following text editors were evaluated with the following points:

- Built-in SCSS, LESS support
- Subsequent installation of an plugin for SCSS, LESS
- Syntax-highlighting
- Autocomplete
- Compilation of SCSS, LESS

Netbeans was tested on Windows 8 version 7.3. There is a SCSS Plugin for the versions 6.9 and higher. It only has to be downloaded from the Netbeans Plugin page (“SCSS Support - NetBeans Plugin detail,” n.d.) and then installed over the Tools/Plugins/Downloaded Menu followed by the Netbeans instructions. The 7.3 version is not available in the Tools/Downloads/Available Plugins for Netbeans – that means it is not officially verified by Netbeans. According to the Netbeans website the plugin supports autocomplete for SCSS files. That’s partly true, be-

cause the autocomplete only works after the rule bracket for the first property. For the following properties the feature doesn't work anymore. Furthermore, with the new plugin it is possible to compile SCSS right in Netbeans. There is also a LESS plugin ("LessCSS Module - NetBeans Plugin detail," n.d.), declared on the Netbeans website only for the Netbeans version 6.9, but it works with the higher versions, too. The LESS plugin only supports syntax highlighting.

There is a SCSS syntax highlighting plugin ("kuroir/SCSS.tmbundle at SublimeText2 · GitHub," n.d.) offered for **Sublime Text 2**. It can be installed via the Package Control (recommended), git or the Browser Packages. The Plugin only supports syntax highlighting. There is also an LESS plugin for syntax highlighting, which can be added the same way like the SCSS plugin.

The inventor of the Sass mode plugin, Brajeshwar Oinam, also integrated this plugin to the new **Coda 2** version ("Coda 2 comes built-in with Sass Mode," n.d.). Now it has a built-in SCSS support for syntax highlighting. Unfortunately, there is no built-in LESS support but there is already a plugin, ("bbpaulwelsh/Coda-2-LESS-mode · GitHub," n.d.) which is based on the CSS and Sass mode that supports syntax highlighting for LESS.

Text Mate 2 offers an easy way to install a SCSS extension. The preferences in the bundle tab have to be opened and then SCSS has to be checked to automatically install the plugin. There are also a lot of different plugins to include LESS syntax highlighting to Text mate 2.

Table 3: Development environment SCSS support

	In-built support	***16 Subsequent installation of an plugin***	***Syntax-highlighting***	***Auto-complete***	***Compilation***
Netbeans 7.3		*x*	*x*	*x (with bug)*	*x*
Sublime Text 2		*x*	*x*		
Coda 2	*x*		*x*		
Text Mate 2		*x*	*x*		

Table 4: Development environment LESS support

	In-built support	***17 Subsequent installation of an plugin***	***Syntax-highligh-ting***	***Auto-complete***	***Compilation***
Netbeans 7.3		*x*	*x*		
Sublime Text 2		*x*	*x*		
Coda2		*x*	*x*		
Text Mate 2		*x*	*x*		

17.1 Utility Programs

There are some utility programs which help to compile the pre-processor code to CSS code. These utility programs adopt the work of the commando line, and show the settings in a GUI (= Graphical User Interface).

LiveReload (“LiveReload,” n.d.) is a utility program which observes the file system. If there are some changes in a specified directory path, the program, if necessary, pre processes files and then update the browser automatically the project is shown in. LiveReload works with Sass/Compass, LESS, Coffescript, Stylus, HAML, IcedCoffeSprict, Jade and Slim. But not only Scripts and pre-processors are supported, also changes in “normal” CSS or HTML files are monitored by LiveReload. To get support in browsers, browser extensions or a short script tag at the end of the HTML file can be added. LiveReload is available for Mac in its second version. There is already an Alpha version 0.8.5.1 for Windows and another program called guard-livereload for Linux which uses the same browser extension as LiveReload (“guard/guard-livereload · GitHub,” n.d.).

```
<script>document.write('<script src="http://' + (location.host || 'lo-
calhost').split(':')[0] + ':35729/livereload.js?snipver=1"></' +
'script>')</script>
```

Listing 25: Script snippet for LiveReload support

CodeKit is only available for Mac in version 1.6.1. It works the same like LiveReload and supports the same Scripts and languages. Additional features of CodeKit are one-click image optimization and the usage of frameworks. Furthermore, a compass project could be created with two easy clicks in an easy user interface, all needed folders and files get configured automatically (“CodeKit — THE Mac App For Web Developers,” n.d.).

Scout is available in version 0.7.1 for Windows and Mac for free. It is a utility program only for compiling Sass and Compass over a GUI. The right directories must be added to the project and the compiling mode can be changed. If the sass directory is compiled over Scout, every Compass mixin can be used, without pre-installing compass to the system (“Scout - Compass and Sass without all the hassle,” n.d.).

It could be said that until now there has been no perfect development environment for Sass/Scss and LESS, but the combination of modern text editors with some utility programs enables a comfortable and effective coding workflow.

18 Conclusion

The practical part of this work shows that creating a website in modular CSS is not very difficult, if started from scratch with a totally new HTML structure. HTML is an essential part of creating modular CSS. It is important to learn about the modularity of CSS before starting a new project because then it is easier to take all necessary steps into consideration. The CSS test showed that the number of lines could be reduced about 20%, when writing modular code. In addition to that, some lines of code could be saved, by using CSS pre-processors, too. The danger by reducing lines of code with the usage of CSS pre-processors is to not think about the compiled code. If for example the nesting is to deep, the compiled code can get longer than the modular CSS or the plain CSS code. A good way between using the advantages of CSS pre-processors and too many lines of code after compilation has to be found when working with this technique.

The answer to the question at the beginning of this thesis is, that if a web developer learns the ropes of modular CSS and CSS pre-processors, these techniques can save a lot of time in real live production for daily use. But it also must be mentioned that if a project is started in modular CSS with CSS pre-processors and then gets compiled and added to a live project, the pre-processor file structure must be saved at any place to keep up the advantages of the pre-processors when maintaining the project later. Furthermore, modular CSS brings along a speed advantage in performance of 17% which can be important in comprehensive web projects. As the tests show, the modular CSS and CSS with Sass are about 30% smaller than plain CSS in the minified version. So it can be said that modular CSS can have an influence on performance in real life scenarios. This leads to the verification of the hypothesis that modular CSS code, in addition with CSS pre-processors, provides a better performance than “plain” CSS code. Moreover, it has been worked out that modular CSS in combination with a pre-processor language enables more efficient coding and the biggest advantages these techniques bring along, show in production. The CSS pre-processors give web developers and designers the possibility to use CSS code dynamically with variables, functions and mixins and the earlier constantly copy-and-paste of a styling rule can be avoided. Furthermore, if modular guidelines are complied with the projects, the module part of a project could serve as basis for other projects.

In addition to that a library of modules, mixins and functions could be constructed over various projects.

18.1 Future prospects

The progress of the development environments shows that more and more popular text editors offer plugins for subsequent installation. The integration of built-in Scss support in the new Coda 2 version shows that these languages will become more important in the future. Further text editors and development environments will follow this example in the future and maybe integrate auto compilation, too.

On the LESS website it is mentioned that this pre-processor language can be compiled directly in modern browsers (“LESS « The Dynamic Stylesheet language,” n.d.); it is not exactly defined which browsers these are. Besides a lot of user posts point out to avoid live compiling in browser because there are too many, unforeseeable errors. With the advancement of the pre-processor languages and the growing proliferation of these, browser vendors hopefully will integrate more elaborated support for in-browser compilation.

References

bbpaulwelsh/Coda-2-LESS-mode · GitHub. (n.d.). Retrieved May 5, 2013, from https://github.com/bbpaulwelsh/Coda-2-LESS-mode

Catlin, H. (2011). *Pragmatic guide to Sass*. Dallas, Tex: Pragmatic Bookshelf.

Chrome Plugin - YSlow. (n.d.). Retrieved May 17, 2013, from https://chrome.google.com/webstore/detail/yslow/ninejjcohidippngpapiilnmkgllmakh

Coda 2 comes built-in with Sass Mode. (n.d.). Retrieved May 5, 2013, from http://brajeshwar.com/2012/coda-2-comes-built-in-with-sass-mode/

CodeKit — THE Mac App For Web Developers. (n.d.). Retrieved May 5, 2013, from http://incident57.com/codekit/

Compass Core Framework | Compass Documentation. (n.d.). Retrieved March 24, 2013, from http://compass-style.org/reference/compass/

CSS Reset - YUI Library. (n.d.). Retrieved April 28, 2013, from http://yuilibrary.com/yui/docs/cssreset/

Geneves, P., Layaida, N., & Quint, V. (2012). On the analysis of cascading style sheets (p. 809). ACM Press. doi:10.1145/2187836.2187946

guard/guard-livereload · GitHub. (n.d.). Retrieved May 5, 2013, from https://github.com/guard/guard-livereload

Howe, S. (2013, März). Front End Legos: Better Design with Reusable HTML & CSS - Shay Howe - HTML 5.tx 2013. Retrieved April 7, 2013, from http://confreaks.com/videos/2230-html5tx2013-front-end-legos-better-design-with-reusable-html-css

Kennedy, A., & León, I. de. (2011). *Pro CSS for high traffic Websites*. [Berkeley, Calif.]; New York: Apress ; Distributed to the book trade worldwide by

Springer Science+Business Media. Retrieved from http://proquest.safaribooksonline.com/?fpi=9781430232889

kuroir/SCSS.tmbundle at SublimeText2 · GitHub. (n.d.). Retrieved May 5, 2013, from https://github.com/kuroir/SCSS.tmbundle/tree/SublimeText2

LESS « The Dynamic Stylesheet language. (n.d.). Retrieved March 24, 2013, from http://lesscss.org/#synopsis

LessCSS Module - NetBeans Plugin detail. (n.d.). Retrieved April 7, 2013, from http://plugins.netbeans.org/plugin/32782/lesscss-module

LiveReload. (n.d.). Retrieved May 5, 2013, from http://livereload.com/

Minimize round-trip times - Make the Web Faster — Google Developers. (n.d.). Retrieved April 26, 2013, from https://developers.google.com/speed/docs/best-practices/rtt#AvoidCssImport

Module: Sass::Script::Functions. (n.d.). Retrieved April 26, 2013, from http://sass-lang.com/docs/yardoc/Sass/Script/Functions.html

Nicole Sullivan (stubbornella) Object Oriented CSS at 2009 Fronteers. (2012). Retrieved from http://www.youtube.com/watch?v=EW8NAFELWzo&feature=youtube_gdata_player

Online YUI Compressor. (n.d.). Retrieved May 3, 2013, from http://refresh-sf.com/yui/

Poll Results: Popularity of CSS Preprocessors | CSS-Tricks. (n.d.). Retrieved March 24, 2013, from http://css-tricks.com/poll-results-popularity-of-css-preprocessors/

Sass - Syntactically Awesome Stylesheets. (n.d.-a). Retrieved March 24, 2013, from http://sass-lang.com/docs.html

Sass - Syntactically Awesome Stylesheets. (n.d.-b). Retrieved March 24, 2013, from http://sass-lang.com/

Scout - Compass and Sass without all the hassle. (n.d.). Retrieved May 5, 2013, from http://mhs.github.io/scout-app/

SCSS Support - NetBeans Plugin detail. (n.d.). Retrieved April 7, 2013, from http://plugins.netbeans.org/plugin/34929/scss-support

Snook, J. (2012). *Scalable and Modular Architecture for CSS*.

Sullivan, N. (2012, July 17). *OOCSS and Preprocessors in a Tree, K-I-S-S-I-N-G*. Freiburg.

Top 12 Browser Versions (Partially Combined) from Oct 2012 to Mar 2013 | StatCounter Global Stats. (n.d.). Retrieved April 28, 2013, from http://gs.statcounter.com/#browser_version_partially_combined-ww-monthly-201210-201303-bar

Wium Lie, H., Elika J.Etemad, & Tab Atkins Jr. (2013, March 1). CSS Cascading and Inheritance Level 3. Retrieved April 26, 2013, from http://www.w3.org/TR/css3-cascade/#at-import

Table of figures

List of tables

Table of listings

Appendix

A. CD

The CD includes the three different web projects from this thesis, with the uncompressed versions of the CSS files and the used web sources in this paper.