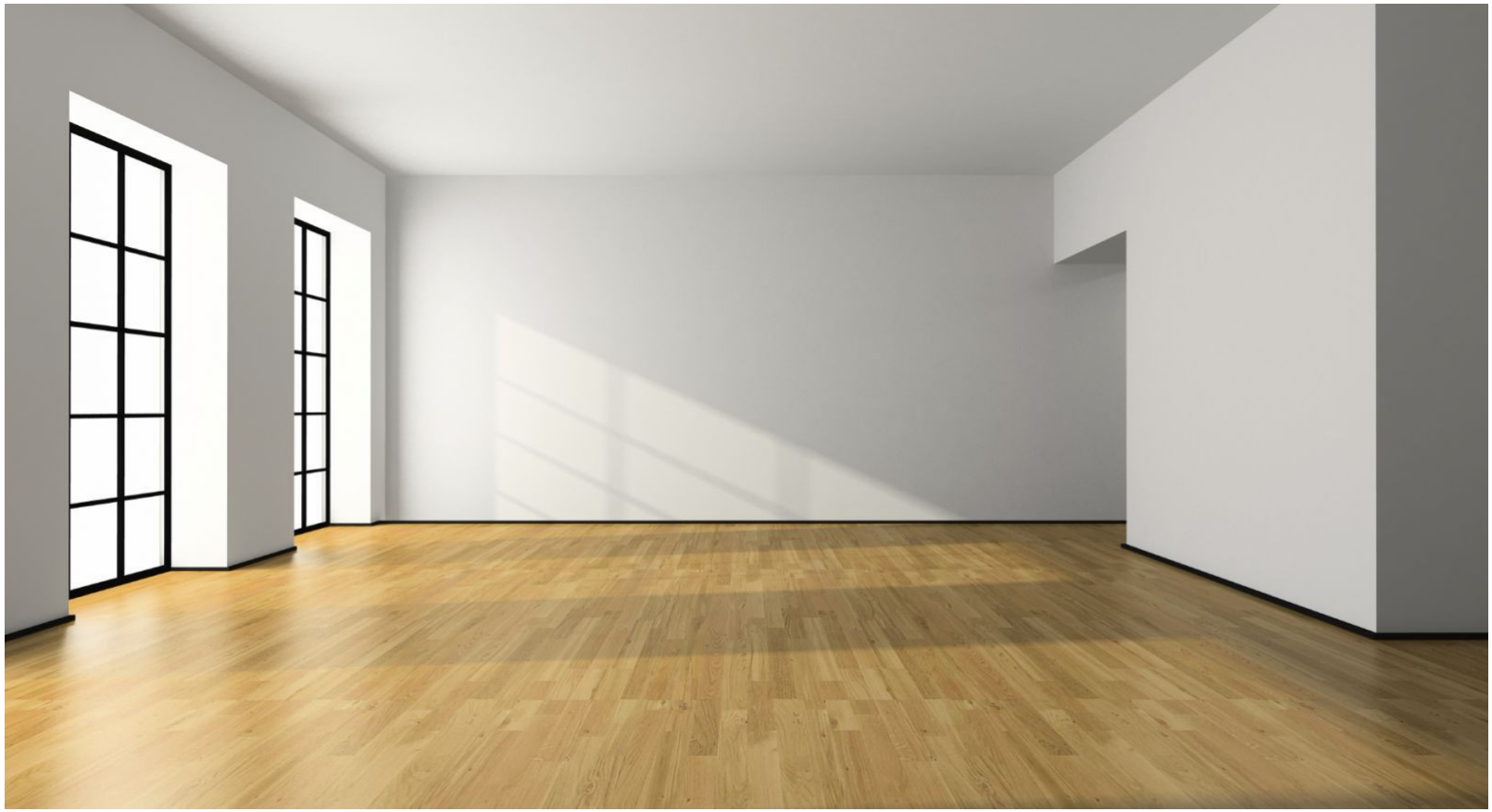
Modern CSS Architecture

Integrator's blueprints

CSS it's simple... It's simple to understand.
But CSS it's not simple to use or maintain.

by Chris Eppstein

It starts looking like this



At the end it looks like this

AGENDA

1. CSS Architecture
2. CSS Methodology
3. CSS Documentation
4. Building CSS like a LEGO

CSS Architecture

What means writing *good CSS code* ?

- To write functional CSS code?
- To write code without tables and as less images as possible?
- To use the latest methods from CSS3 ?

Another *requests:*

- Is it readable?
- Is it easy to change or extend?
- Is it modular and scalable?
- Is it independent of the other parts of the application or DOM elements?

The Goals of a Good CSS Architecture

- ***Predictable***

  Your rules should act as you want them to act.
  It should not affect any parts of your site unintented when adding/updating other rules.

- ***Reusable***

  Use abstract and independent rules so you can quickly build new components and avoid resolving problems that are already fixed.

- ***Maintainable***

  You shouldn't refactor our current style when we add / update some elements / components.

- ***Scalable***

  Your site CSS Architecture is easily approachable by a large team and a large amount of info.

The Common *Bad Practices*

Common solutions might be bad practices even if they are technically valid:

1. Modifying components based on who their parents are

Almost all apps have sections with common elements, except one.

- It's not predictable (several buttons used in different location might look different despite same markup);
- It's not scalable or reusable;
- For new look on homepage, new rules will be added;
- Not maintainable
- It's violating the open/close principle of software dev.

  'Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.'

  Bertrand Meyer

```
.btn {
  background: yellow;
  border: 1px solid black;
  color: black;
  width: 50%;
}

#sidebar .btn {
  width: 200px;
}

body.homepage .btn {
  background: white;
}
```

2. Try to avoid complicated selectors

This has started in the desire to get clean HTML DOM.

```
#wrapper section h1:first-child { ... }
#articles > section > h3 + p { ... }
#navigator ul li a ul li:before { ... }
```

- Not reusable;
- Not predictable;
- Not maintainable or scalable when the HTML changes.

3. Making a rule with too many properties

Try to avoid rule's nesting

- You can not always reuse it;
- Duplicate code.

```
.button {
  position: absolute;
  top: 20px;
  left: 20px;
  background-color: red;
  font-size: 1.5em;
  text-transform: uppercase;
}
```

CAUSES

They all add too much weight on the styiling job of CSS.

Wait. But isn't this the purpose and the goals of CSS?

Yes, BUT separating the HTML of the CSS doesn't mean that the style has to be always responsible on the CSS sholders.

HTML is almost always as a structure and rarely just content.

HTML and CSS must work together in all ways.

CSS must be splitted into presentation and layout.

SOLUTIONS

- CSS should be just a set of visual elements;
- Componenets that must look different should be defined different and HTML is reponsible to call;
- CSS has to know as less possible as he can about HTML structure;

Example:

We define a component with the .button class. If HTML needs an element to look like this, it should call it.

If there some places where the style of this one changes, then that style has to be defined into another element and the HTML will just include it in the new class and to complete the new look.

The *BEST Practices*

1. Be intentional

Don't allow the selector to have unwanted style by using a sniper and not a grenade.

```
1  /* Heavy grenade */
2  #nav ul li ul { ... }
```

```
4  /* Thin sniper */
5  .subnav { ...}
```

2. Modulate your concern

Avoid nesting ***background, color, font*** properties with ***position, width, height*** and ***margin***.

3. Namespace your classes

Parent selectors are not very efficient, that's why it's a good solution to add to the subelements the parent class. Is minimizing confusion and makes components them more modular.

```
1  /* A big risk of style cross-contamination */
2  .button { }
3  .button .icon { }
```

```
1  /* A small risk of style cross-contamination */
2  .button { }
3  .button-icon { }
```

4. Use modifier classes

When we need a components to look different into a different context, we can create a modifier instead of nesting and attaching it to the parent.

```
1  /* Bad practice */
2  .button { }
3  #sidebar .button { }
```

```
5  /* Good practice */
6  .button { }
7  .button-sidebar { }
```

5. Define the purpose of every class

Classes are used for: style, javascript handling, for feature detections, etc.

We get a big AFRAID of deleting a class because we do not know what are used for.

Solution

.js-button – For JavaScript handling

.supports-button – For Modernizr classes

.button – For style and only for style

6. Name your classes with logical structure

CSS now is written using a lot of hypens and words.
But it's not enought to identify our elements.

```
/* A component */
.button-group { }

/* A component modifier (modifying .button) */
.button-primary { }

/* A component sub-object (lives within .button) */
.button-icon { }

/* Is this a component class or a layout class? */
.header { }
```

Looks nice, simple, elegant. But what are every element purposes and what they represent? We can not know.

That's why, we use naming conventions that will help us to know exactly the purpose of every item.

```
/* Component Rules */
.component-name { }
.component-name--modifier-name { }
.component-name__sub-object { }
.component-name__sub-object--modifier-name { }

/* Layout Rules */
.l-layout-method { }
.grid { }

/* State Rules */
.is-state-type { }

/* Non-styled JavaScript Hooks */
.js-action-name { }
```

CSS Methodology

We need a methodology for CSS because

- Code reused is almost NONEXISTENT
- Code is too fragile
- Everybody wants to write their clever code
- The size of the CSS will increase almost 1:1 in a relantionship with the number of the blocks, pages and complexity of content

MOST EFFICIENT METHODOLOGIES

OOCSS

(Object Oriented CSS)

SMACSS

(Scalable & Modular Architecture for CSS)

BEM

(Block, Element, Modifier)

OOCSS (Object Oriented CSS) (Nicole Sullivan)

Write CSS for thousands of pages by writing CSS code based on reusable objects.

A CSS Object is a visual pattern that can be converted in a html snippet.

PRINCIPLES

1. Separate Structure and Skin

Create abstract elements and decorate them with skins. Extend class with aditional classes.

2. Separate Content and Container

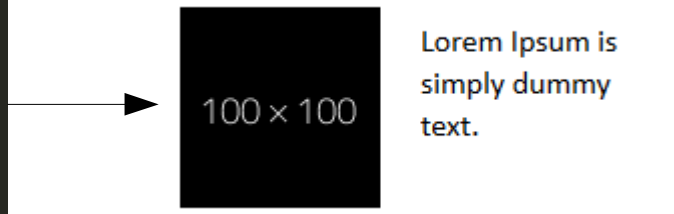
*Break code into container module (**module object**) and content of it.*

```css
.media{
    overflow:hidden;
    zoom: 1;
    width: 250px;
}
.media-img{
    float:left;
    margin-right: 10px;
}
```

```css
.media-img > img{
    display: block;
    margin: 10px;
}
.media-body{
    overflow:hidden;
}
.media-shadow{
    box-shadow: 0 1px 5px rgba(0,0,0,0.75);
}
```

BASE CLASS

```html
<div class='media'>
    <div class='media-img'>
        <img src='http://dummyimage.com/100x100/000/fff' />
    </div>
    <div class='media-body'>
        <p>Lorem Ipsum is simply dummy text.</p>
    </div>
</div>
```



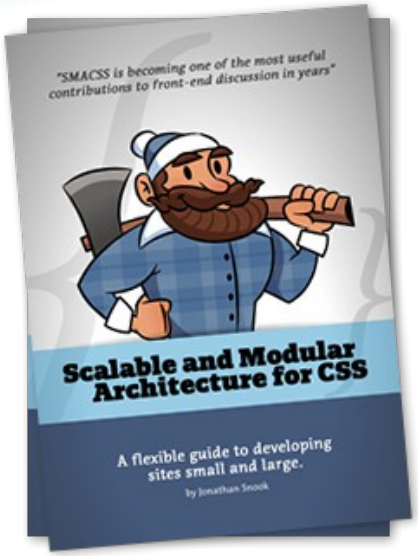
MODIFIER

DESCENDENT

```html
<div class='media media-shadow'>
    <div class='media-img'>
        <img src='http://dummyimage.com/100x100/000/fff' />
    </div>
    <div class='media-body'>
        <p>Lorem Ipsum is simply dummy text.</p>
    </div>
</div>
```

SMACSS (Scalable & Modular Architecture for CSS)

SMACSS is a way to examine your design process and an attempt to document a consistent approach to site development when using CSS



PRINCIPLES

1. Categorizing CSS Rules

Create a 'box' where you have rules splited into different categories.
Base / layout / module / state / theme

2. Naming rules

Adapt a naming convention so you can identify quickly the type of the element.
l-, layout-, grid-, is-opened, is-hidden, is-collapsed, module-.

3. Minimizing the depth of applicability

It cares more on adding extra classes to HTML elements rather than use elements with a big depth.
NO *body.article > #main > #content > #intro > p > b*
YES *.article #intro b*

Build a house

Build an app

1. Foundation
Main holder for all things inside the house.

→

1. Normalize
Reset / normalization of the css

2. Structure
Keep the building up

→

2. Grid System
Play with the main layout

3. Fixtures
Doors, stares

→

3. Elemenets style
Main HTML tags without classes

4. Decoration
Wallpaper, paint, floor

→

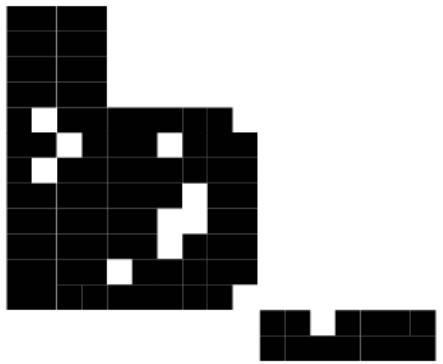
4. Components and elements with classes
Navigation, galleries, buttons, forms

5. Ornaments
Paintings, canvases.

→

5. Style trump (promotions)
Logo for seasons, promotions, etc.

BEM (Block, Element, Modifier)



BEM is added to our practice by Yandex team – Russia and is having
As main principle **to give transparency and meaning to our CSS classes.**

It's the best choise for large projects.
- by Jonathan Snook

COMPONENTS

- **.block {}** – the highest level of an abstraction or a component
- **.block__element{}** - descendent that helps .block{} to be one complet component. Is a best practice to use modifier even for childrens of .block{}like : **.block__element--modifier{}**
- **.block--modifier{}** - different state of .block{}

CSS Documentation

KSS (Knyle Style Sheets)

A **tool** that is generating a documentation for our css or css preprocesor (less, sass, scss).

How can KSS help me ?

Think it as a shelter full of shoes from where you can choose what pair you think it's appropiate.

Facilities of using it:

- Easy to read the code and the description
- Display all elements into a hierarchical mode
- Pick-up an entire section
- Display animation of created elements
- Customize the output with specifically colors / logos and other elements
- Output an HTML styleguide divided in sections

Do you want to see how this beauty works ?

Build CSS like a LEGO

What is a preprocessor ?

A preprocessor is a program that processes its input data to produce output that is used as input to another program.

We use preprocessors because:

- We can write a nested syntax
- We can define variables for anything we want (font, color, size, padding, margin)
- We can define all kinds of mixins functions
- We can use math functions
- Operational functions (ligthen, darkness, saturate, etc)
- Split our code into multiple files

Nesting

Nesting allow us to have a cleaner method to target DOM's elements and to avoid writing a class multiple times.

```
1   .button{
2       &.button--action {
28      }
29
30      &.button--dark {
46      }
47  }
```

Variables

We can define variables for any usage we need like fonts type fonts family, colors, padding, margin, sizes, etc.

```
72  @baseBgColor:           #1d1d1c;
73  @lightBaseBgColor:      lighten(@baseBgColor, 15%);
74  @darkBaseBgColor:       darken(@baseBgColor, 5%);
```

```
136  @urlImg:'/bundles/
```

```
120  @fontRoboto: 'Roboto', sans-serif; // Available weights: 100, 300, 400, 700
121  @fontRobotoCond: 'Roboto Condensed', sans-serif; // Available weights: 400, 700
122  @fontOpenSans: 'Open Sans', sans-serif; // Available weights: 400, 600
```

Mixins

Mixins allow us to group multiple CSS lines to reuse it through entire stylesheet

```
1  .positionRelative(){
2      position:relative;
3      left:0;
4      top:0;
5  }
```

```
7  .common{
8      .positionRelative();
9  }
```

Math functions

Allow us to perform calculations.

```
@base-size: 10px;
.small {
  font-size: @base-size;
}
```

```
.medium {
  font-size: @base-size * 1.2;
}
```

```
.large {
  @_large: @base-size * 1.5;
  font-size: @_large;
  line-height: @_large + 4;
}
```

Operational functions

Operational functions help us to handle our less so we can adjust the style based on the current variables, but with small details in different scenarios.

```
1  .decorateElement(@startColor, @endColor){
2      @middleColor: mix(@startColor, @endColor, 50);
3      box-shadow: 1px 1px 5px darken(@startColor, 10%);
4  }
```

Operational functions are : darkness, lighten, mix, rgb, red, green, alpha,saturate, mix, etc.

What makes preprocessors so GREAT?
The power of example will show us

REQUESTS:

- Create 2 elements and 2 modifiers;
- Each element should be extended with 5 colors;
- Our preprocessor code should be: maintainable, extendable and scalable;

CSS isn't just visual design.

Don't throw out programming best practices just because you're writing CSS.

Concepts like OOP, DRY, the open/closed principle separation of concern, etc... still apply to CSS.

THANK YOU!

p e n t a l o g . f r

Eduard OMUSORU, email: eomusoru@pentalog.fr, github: https://github.com/eomusoru/css