



CSS Methodologies

October 2022



Agenda

1 GENERAL OVERVIEW

2 CSS ARCHITECTURE

3 THE SEPARATION OF CONCERNS

4 CONTENT-AGNOSTIC CSS + UTILITY CLASSES

5 CSS METHODOLOGIES



Maintainable Code

You don't write clean markup for the browser; you don't write it for the end users. You write it for the person who takes over the job from you. Much like you should use good grammar.

Unstructured CSS in a large project often became a mess. Minor changes fix one problem but create three more and can require ugly hacks, and small CSS changes can break JavaScript functionality.

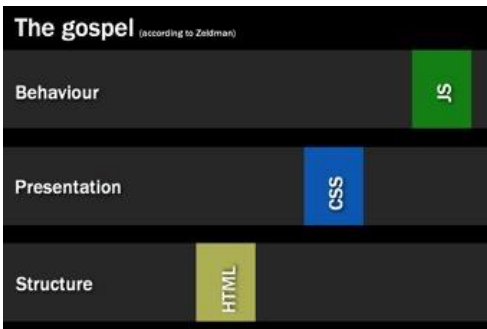


Benefits of a Smart Scalable CSS Architecture

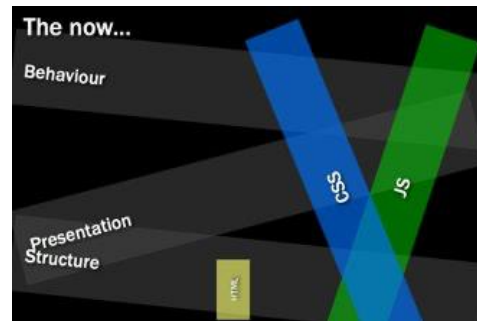
- Fewer styling rules
- Fewer styling collisions
- Long-term maintainability
- Faster ramp-up for new team members
- Easier collaboration between team members
- Smoother project handoffs

The Separation of Concerns

The separation of technologies is an incredibly good idea, but it was not always implemented in real life products.



The separation of technologies: HTML is the structure; CSS is for the visual look and feel and JavaScript is for behavior.



Think of the "separation of concerns" and not layers of development.

After each new web technology is introduced, the separation becomes much harder. What good is a CANVAS without any scripting? Should we animate in CSS or in JavaScript? Is animation behavior or presentation? Should elements needed solely for visual effects be in the HTML or generated with JavaScript or with `:after` and `:before` in CSS?

Separation of Concerns

The idea of "separation of concerns" is that your HTML should only contain information about your content, and all your styling decisions should be made in your CSS. This idea is reflected on a well-known csszengarden.com site (you can completely redesign a site just by swapping out the stylesheet.)

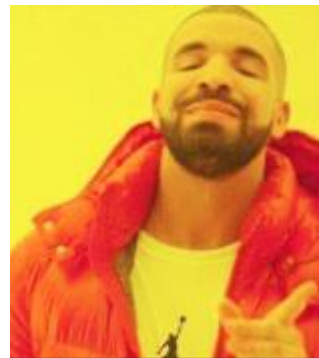
```
<p class="text-center">Hello there!</p>
```

```
<style>
.text-center {
  text-align: center; }
</style>
```



```
<p class="greeting">Hello there!</p>
```

```
<style>
.greeting {
  text-align: center; }
</style>
```



Concerns are Not So Separated

Most of the time the CSS is like a mirror for the markup, perfectly reflecting my HTML structure with nested CSS selectors. The markup isn't concerned with styling decisions, but CSS is very concerned with the markup structure.

```
<div class="company-info">
  
  <div>
    <h2>Epam</h2>
    <p>
      We specialize in 11 industries
      in 25 countries, delivering
      innovative solutions
      to our customers'
      most challenging problems.
    </p>
  </div>
</div>
```

```
.company-info {
  background-color: #999;
  border: 1px solid #76cdd8;
  border-radius: 4px;
  box-shadow: 0 2px 4px #ccc;

  & > img {
    display: block;
    margin: 0 auto;
  }

  & > div {
    padding: 1rem;

    h2 {
      font-size: 1.25rem;
    }

    p {
      color: rgba(0,0,0,0.75);
    }
  }
}
```

Decoupling Styles From Structure

Keeping selector specificity low and making your CSS less dependent on your particular DOM structure.

```
<div class="company-info">
  
  <div class="company-info-content">
    <h2 class="company-info-name">Epam</h2>
    <p class="company-info-body">
      We specialize in 11 industries
      in 25 countries, delivering
      innovative solutions
      to our customers'
      most challenging problems.
    </p>
  </div>
</div>
```

```
.company-info {
  background-color: #999;
  border: 1px solid #76cdd8;
  border-radius: 4px;
  box-shadow: 0 2px 4px #ccc;
}

.company-info-image {
  display: block;
  margin: 0 auto;
}

.company-info-content {
  padding: 1rem;
}

.company-info-name {
  font-size: 1.25rem;
}

.company-info-body {
  color: rgba(0,0,0,0.75);
}
```


Dealing with Similar Components

Say we needed to add a new block a preview of an article in a card layout. And it should look exactly like a company-info block with a rounded corners, box-shadow, etc.

```
<div class="company-info">
  
  <div class="company-info-content">
    <h2 class="company-info-name">Epam</h2>
    <p class="company-info-body">...</p>
  </div>
</div>
```

```
<div class="article-preview">
  
  <div class="article-preview-content">
    <h2 class="article-preview-name">Breaking news!</h2>
    <p class="article-preview-body">...</p>
  </div>
</div>
```

Dealing with Similar Components

Option 1. Duplicate styles. But it will break the DRY principle

```
.company-info {...}
.company-info-image {...}
.company-info-content {...}
.company-info-name {...}
.company-info-body {...}

.article-preview {...}
.article-preview-image {...}
.article-preview-content {...}
.article-preview-name {...}
.article-preview-body {...}
```

Option 2. Use SCSS *@extend* construction. Which is better not to use at all

```
.article-preview {
  @extend .author-bio;
}
...
```

Dealing with Similar Components

Option 3. Create a content-agnostic component

Our *.company-info* and *.article-preview* components have nothing in common from a "semantic" perspective. But they have a lot in common from a design perspective. We can create and reuse a new abstract component named after what they do have in common

```
<div class="media-card">
  
  <div class="media-card__content">
    <h2 class="media-card__title">Epam</h2>
    <p class="media-card__body">
      We specialize in 11 industries in 25 countries, delivering innovative
      solutions to our customers' most challenging problems.
    </p>
  </div>
</div>
```

```
background-color: white;
border: 1px solid hsl(0,0%,85%);
border-radius: 4px;
box-shadow: 0 2px 4px rgba(0,0,0,0.1);
overflow: hidden;
}
.media-card__image {
  display: block;
  width: 100%;
  height: auto;
}
.media-card__content {
  padding: 1rem;
}
.media-card__title {
  font-size: 1.25rem;
  color: rgba(0,0,0,0.8);
}
.media-card__body {
  font-size: 1rem;
  color: rgba(0,0,0,0.75);
  line-height: 1.5;
```

Content-agnostic CSS + Utility Classes

The content-agnostic CSS concept could be expanded to creation of a doesn't of a small reusable classes each of which represent a small set of CSS rules

One html element could have a lot of CSS classes attached. This concept prefers composition to duplication. However, it produces an HTML which looks like it uses just inline styles.

```
.btn, .btn-primary, .btn-secondary  
.badge  
.card-list, .card-list-item  
.img-round, .img-round,  
.margin-top-10, .padding-left-large, .centered-text  
.hidden-on-tablet, .hidden-on-less-than-medium
```

```
<div class="flex text-lg text-dark">  
  <span class="py-2 px-4 border-r border-dark-soft">hello</span>  
  <button class="f6 br3 ph3 pv2 white bg-purple hover-bg-light-purple">Button Text</button>  
</div>
```

As opposite to the 'separation of concerns' in this case The CSS isn't concerned with the markup structure, but HTML is very concerned about existing CSS classes.

Why to Use CSS Organizing Methodologies?

There is no "one true way" to write CSS but there are techniques that can keep CSS more organized and more structured, leading to code that is easier to build and easier to maintain.

No matter what methodology you choose to use in your projects, you will benefit from the advantages of more structured CSS and UI. Some styles are less strict and more flexible, while others are easier to understand and adapt in a team.



BLOCK, ELEMENT, MODIFIER

BEM

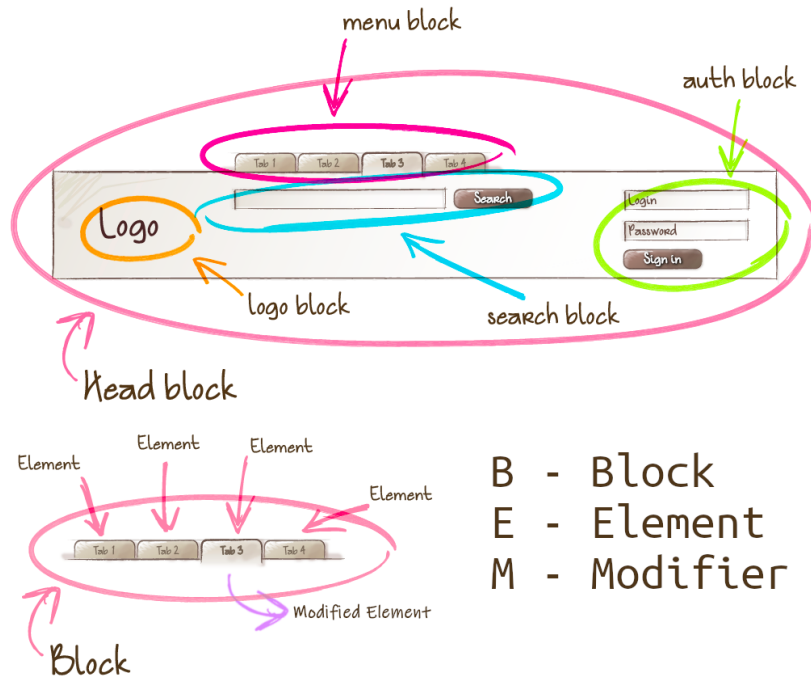
BEM (**B**lock, **E**lement, **M**odifier) is a component-based approach to web development. The idea behind it is to divide the user interface into independent blocks. This makes interface development easy and fast even with a complex UI, and it allows reuse of existing code without copying and pasting.

BEM makes your code scalable and reusable, thus increasing productivity and facilitating teamwork. Even if you are the only member of the team, BEM can be useful for you. Nevertheless, many developers believe that such a system approach like BEM puts additional boundaries on their project and makes your project overloaded, cumbersome, and slow.

BEM

BEM (**B**lock, **E**lement, **M**odifier) is a component-based approach to web development. The idea behind it is to divide the user interface into independent blocks. This makes interface development easy and fast even with a complex UI, and it allows reuse of existing code without copying and pasting.

BEM makes your code scalable and reusable, thus increasing productivity and facilitating teamwork. Even if you are the only member of the team, BEM can be useful for you. Nevertheless, many developers believe that such a system approach like BEM puts additional boundaries on their project and makes your project overloaded, cumbersome, and slow.



The Main Reasons Why We Do Not Use Any Selectors Except Classes

One of the basic rules of the BEM methodology is to use only class selectors. In this section, we'll explain why.

- Why don't we use IDs?
- Why don't we use tag selectors?
- Why don't we use a universal selector?
- Why don't we use CSS reset?
- Why don't we use nested selectors?
- Why don't we combine a tag and a class in a selector?
- Why don't we use combined selectors-
- Why don't we use attribute selectors?

We Don't Use IDs (ID Selectors)

The ID provides a unique name for an HTML element. If the name is unique, you can't reuse it in the interface. This prevents you from reusing the code.

We Don't Use Tag Selectors

HTML page markup is unstable: A new design can change the nesting of the sections, heading levels (for example, from `<h1>` to `<h3>`) or turn the `<p>` paragraph into the `<div>` tag. Any of these changes will break styles that are written for tags. Even if the design doesn't change, the set of tags is limited. To use an existing layout in another project, you must solve conflicts between styles written for the same tags.

We Don't Use CSS Reset

CSS reset is a set of global CSS rules created for the whole page. These styles affect all layout nodes, violate the independence of components, and make it harder to reuse them.

In BEM, "reset" and "normalize" aren't even used for a single block. Resetting and normalization cancel existing styles and replace them with other styles, which you will have to change and update later in any case. As a result, the developer must write styles that override the ones that were just reset.

We Don't Use The Universal Selector (*)

The universal selector indicates that the project features a style that affects all nodes in the layout. This limits reuse of the layout in other projects.

In addition, a universal selector can make the project code unpredictable. For example, it can affect the styles of the universal library components.

We Don't Use Nested Selectors

Nested selectors increase code coupling and make it difficult to reuse the code.

The BEM methodology doesn't prohibit nested selectors, but it recommends not to use them too much. For example, nesting is appropriate if you need to change styles of the elements depending on the block's state or its assigned theme.

The Basics Of BEM

The BEM methodology is a set of universal rules that can be applied regardless of the technologies used, such as CSS, Sass, HTML, JavaScript or React.

BEM helps to solve the following tasks:

- Reuse the layout;
- Move layout fragments around within a project safely;
- Move the finished layout between projects;
- Create stable, predictable and clear code;
- Reduce the project debugging time.

Block

A functionally independent page component that can be reused. In HTML, blocks are represented by the class attribute.

The block name describes its purpose ("What is it?" — menu or button), not its state ("What does it look like?" — red or big).

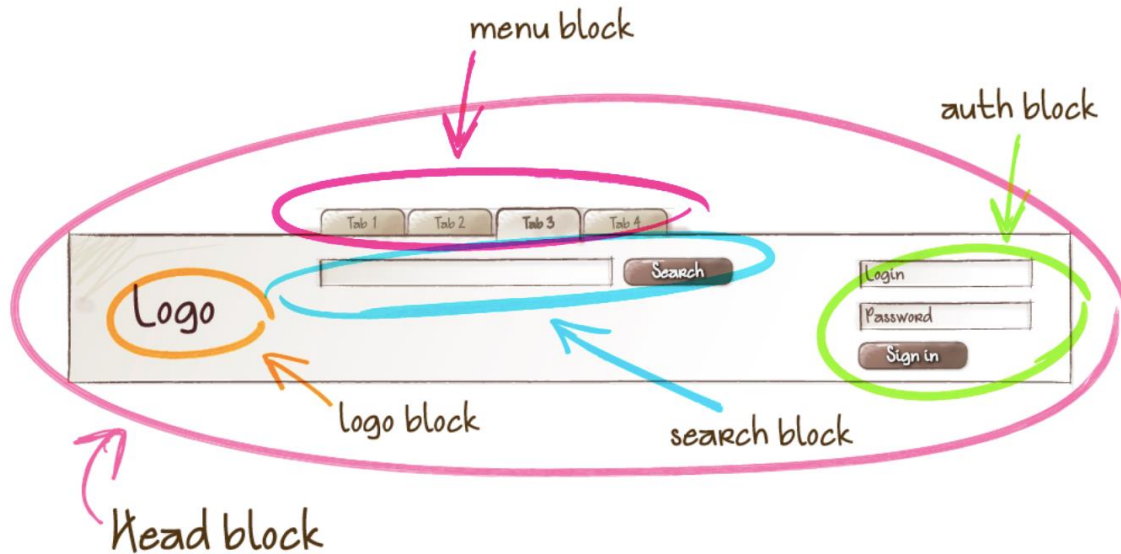
```
<!-- Correct. The `error` block is semantically meaningful -->  
<div class="error"></div>
```

```
<!-- Incorrect. It describes the appearance -->  
<div class="red-text"></div>
```


Block Features. Nested Structure

Blocks can be nested inside any other blocks.

For example, a head block can include a logo (logo), a search form (search), and an authorization block (auth).



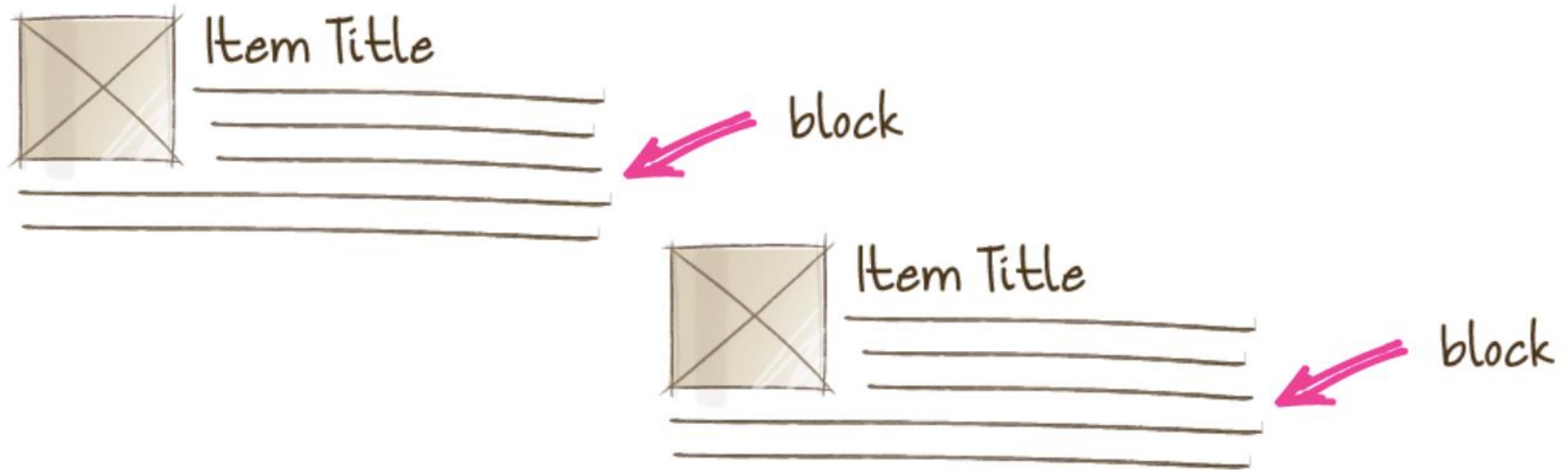
Block Features. Arbitrary Placement

Blocks can be moved around on a page, moved between pages or projects. The implementation of blocks as independent entities makes it possible to change their position on the page and ensures their proper functioning and appearance.



Block Features. Re-use

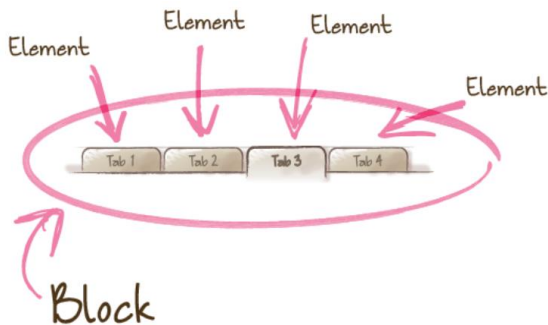
An interface can contain multiple instances of the same block.



Element

A composite part of a block that can't be used separately from it.

- The element name describes its purpose ("What is this?" — item, text, etc.), not its state ("What type, or what does it look like?" — red, big, etc.).
- The structure of an element's full name is *block-name__element-name*. The element name is separated from the block name with a double underscore (__).



```
<!-- `search-form` block -->
<form class="search-form">
  <!-- `input` element in the `search-form` block -->
  <input class="search-form__input">

  <!-- `button` element in the `search-form` block -->
  <button class="search-form__button">Search</button>
</form>
```

Guidelines For Using Elements. Nesting

- Elements can be nested inside each other.
- You can have any number of nesting levels.
- An element is always part of a block, not another element. This means that element names can't define a hierarchy such as block__elem1__elem2.

```
<form class="search-form">  
  <div class="search-form__content">  
    <input class="search-form__input">  
  
    <button class="search-form__button">Search</button>  
  </div>  
</form>
```




```
<form class="search-form">  
  <div class="search-form__content">  
    <!-- Recommended: `search-form__input` or `search-form__content-input` -->  
    <input class="search-form__content_input">  
  
    <!-- Recommended: `search-form__button` or `search-form__content-button` -->  
    <button class="search-form__content_button">Search</button>  
  </div>  
</form>
```




Guidelines For Using Elements. Membership

An element is always part of a block, and you shouldn't use it separately from the block.



```
<!-- `search-form` block -->
<form class="search-form">
  <!-- `input` element in the `search-form` block -->
  <input class="search-form__input">

  <!-- `button` element in the `search-form` block -->
  <button class="search-form__button">Search</button>
</form>
```



```
<!-- `search-form` block -->
<form class="search-form">
</form>

<!-- `input` element in the `search-form` block -->
<input class="search-form__input">

<!-- `button` element in the `search-form` block -->
<button class="search-form__button">Search</button>
```

Guidelines For Using Elements. Optionality

An element is an optional block component. Not all blocks have elements.

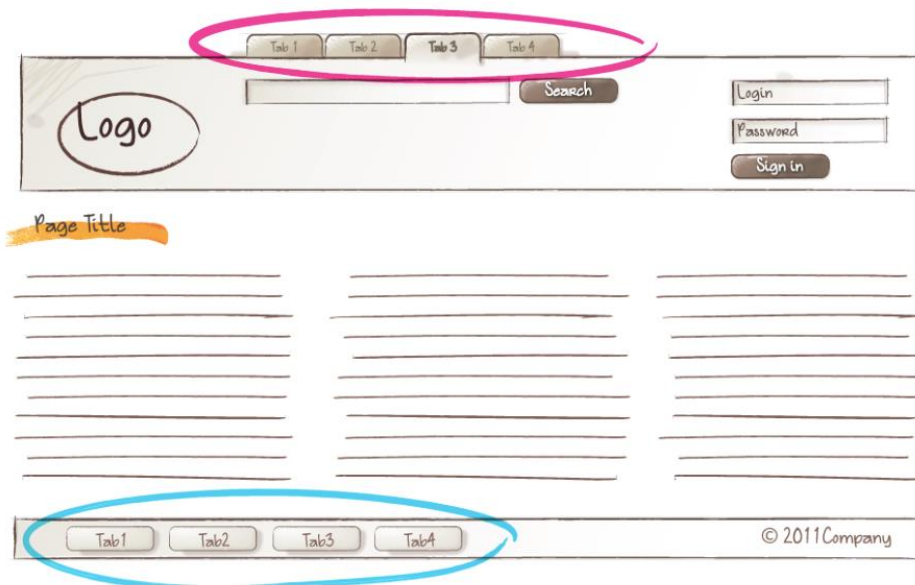
```
<!-- `search-form` block -->
<div class="search-form">
  <!-- `input` block -->
  <input class="input">

  <!-- `button` block -->
  <button class="button">Search</button>
</div>
```

Modifier

Modifiers are similar in essence to HTML attributes. The same block looks different due to the use of a modifier.

For instance, the appearance of the menu block (menu) may change depending on a modifier that is used on it.



Modifier

An entity that defines the appearance, state, or behavior of a block or element.

Features:

- The modifier name describes its appearance ("What size?" or "Which theme?" and so on — `size_s` or `theme_islands`), its state ("How is it different from the others?" — `disabled`, `focused`, etc.) and its behavior ("How does it behave?" or "How does it respond to the user?" — such as `directions_left-top`).
- The modifier name is separated from the block or element name by a single underscore (`_`).

Types of Modifiers. Boolean

Used when only the presence or absence of the modifier is important, and its value is irrelevant. For example, disabled. If a Boolean modifier is present, its value is assumed to be true.

The structure of the modifier's full name follows the pattern:

- *block-name_modifier-name*
- *block-name__element-name_modifier-name*

```
<!-- The `search-form` block has the `focused` Boolean modifier -->
<form class="search-form search-form_focused">
  <input class="search-form__input">

  <!-- The `button` element has the `disabled` Boolean modifier -->
  <button class="search-form__button search-form__button_disabled">Search</button>
</form>
```

Types of Modifiers. Key-value

Used when the modifier value is important. For example, "a menu with the islands design theme": `menu_theme_islands`.

The structure of the modifier's full name follows the pattern:

- *block-name_modifier-name_modifier-value*
- *block-name__element-name_modifier-name_modifier-value*

```
<!-- The `search-form` block has the `theme` modifier with the value `islands` -->
<form class="search-form search-form_theme_islands">
  <input class="search-form__input">

  <!-- The `button` element has the `size` modifier with the value `m` -->
  <button class="search-form__button search-form__button_size_m">Search</button>
</form>
```

A Modifier Can't Be Used Alone

From the BEM perspective, a modifier can't be used in isolation from the modified block or element. A modifier should change the appearance, behavior, or state of the entity, not replace it.



```
<form class="search-form search-form_theme_islands">  
  <input class="search-form__input">  
  
  <button class="search-form__button">Search</button>  
</form>
```



```
<form class="search-form_theme_islands">  
  <input class="search-form__input">  
  
  <button class="search-form__button">Search</button>  
</form>
```

Mix

A technique for using different BEM entities on a single DOM node.

Mixes allow you to:

- Combine the behavior and styles of multiple entities without duplicating code.
- Create semantically new UI components based on existing ones.

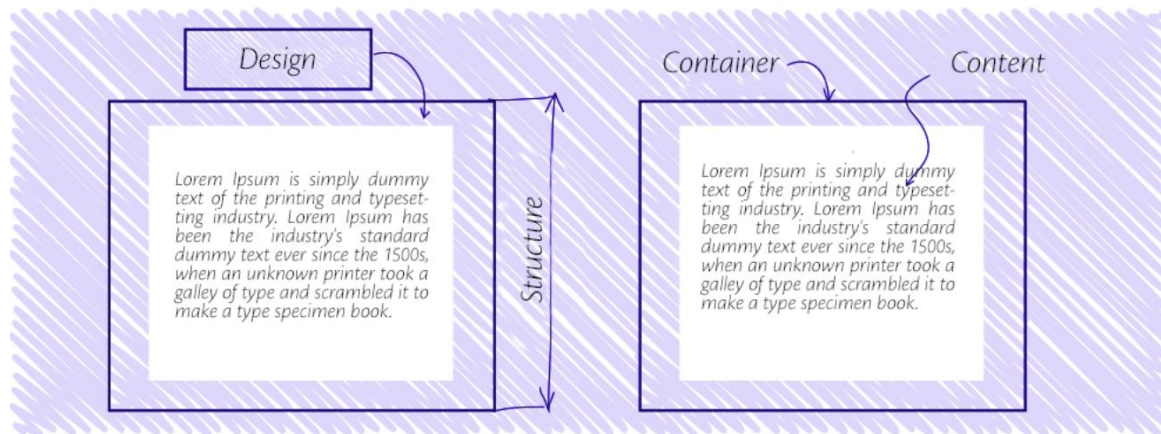
```
<!-- `header` block -->
<div class="header">
  <!--
    The `search-form` block is mixed with the `search-form` element
    from the `header` block
  -->
  <div class="search-form header__search-form"></div>
</div>
```

OBJECT-ORIENTED CSS

OOCSS

This approach has two main ideas:

- Separation of structure and design
- Separation of container and content



OOCSS

Good: reducing the amount of code by reusing it (DRY principle).

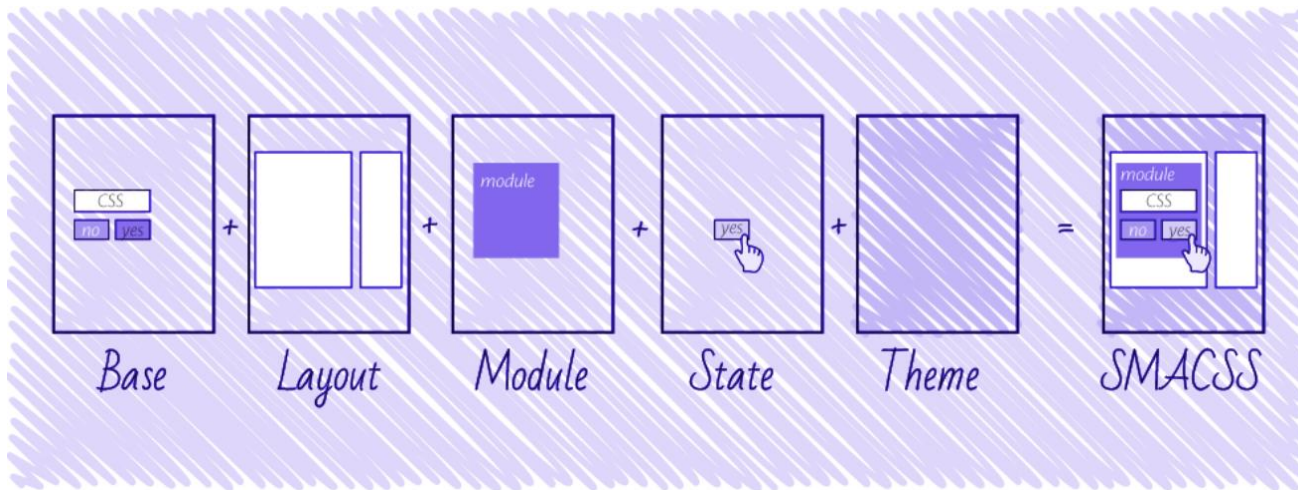
Bad: complex support. When you change the style of a particular element, you will most likely have to change not only CSS (because most classes are common), but also add classes to the markup.

SCALABLE AND MODULAR ARCHITECTURE FOR CSS

SMACSS

The main goal of the approach is to reduce the amount of code and simplify code support.

Jonathan Snook divides styles into 5 parts:



- 1. Base rules.** These are styles of the main website elements – body, input, button, ul, ol, etc. In this section, we use mainly HTML tags and attribute selectors, in exceptional cases – classes;
- 2. Layout rules.** Here are the styles of global elements, the size of the cap, footer, sidebar, etc. Jonathan suggests using id here in selectors since these elements will not occur more than 1 time on the page. However, the author of the article considers this a bad practice;
- 3. Modules rules.** Blocks that can be used multiple times on a single page. For module classes, it is not recommended to use id and tag selectors;
- 4. State rules.** In this section, the various statuses of the modules and the basis of the site are prescribed. This is the only section in which the use of the keyword “! Important” is acceptable.
- 5. Theme rules.** Design styles that you might need to replace.

ATOMIC CSS

Atomic CSS

This approach has significant drawbacks:

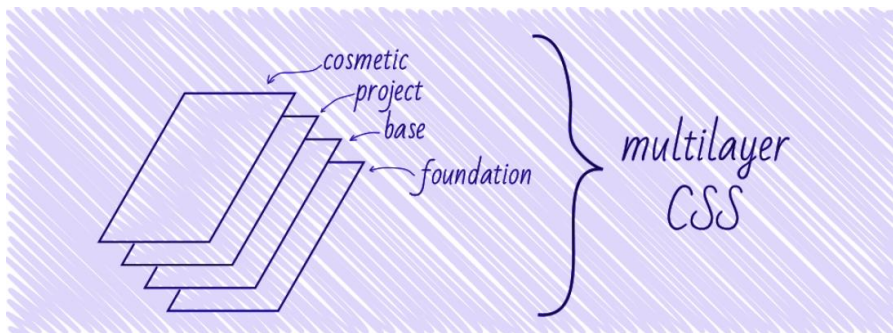
- Class names are descriptive property names, not describing the semantic nature of the element, which can sometimes complicate development.
- Display settings are directly in the HTML.

MULTILAYER

MCSS

This style of code writing suggests splitting styles into several parts, called layers.

- **Zero layer or foundation.** The code responsible for resetting browser styles (e.g. reset.css or normalize.css);
- **Base layer** includes styles of reusable elements on the site: buttons, input fields for text, hints, etc.
- **Project layer** includes separate modules and a “context” – modifications of the elements depending on the client browser, the device on which the site/application is viewed, user roles, and so on.
- **Cosmetic layer** is written the OOCSS style, making minor changes in the appearance of elements.



The hierarchy of interaction between layers is very important:

- The base layer defines neutral styles and does not affect other layers.
- Elements of the base layer can only affect the classes of its layer.
- Elements of the project layer can affect the base and project layers.
- The cosmetic layer is designed in the form of descriptive OOCSS-classes (“atomic” classes) and does not affect other CSS-code, being selectively applied in the markup.

FLAT HIERARCHY OF SELECTORS, UTILITY STYLES,
NAME-SPACED COMPONENTS

FUN

There is a certain principle behind each letter of the name:

- **F**, flat hierarchy of selectors: it is recommended to use the classes to select items, avoid a cascade without the need, and do not use ids.
- **U**, utility styles: it is encouraged to create the service atomic styles for solving typical makeup tasks, for example, `w100` for `width: 100%`; or `fr` for `float: right`;
- **N**, name-spaced components: the author recommends to add namespaces for specifying the styles of specific modules elements. This approach will avoid overlapping in class names.

FUN

This approach imposes quite a few requirements on the project and the code structure, it only establishes the preferred form of recording selectors and the way they are used in the markup. But in small projects, these rules can be quite enough to create high-quality code.

Flat hierarchy of .selectors

Utility styles .b-white

Name-spaced name-components

Useful links

BEM <http://bem.info/>, <https://en.bem.info/methodology/quick-start/>

Bootstrap grid <http://simplegrid.io/>, <https://960.gs/>

CSS Grid [css-tricks with grid](#), [grid visualizer](#)

Statistics of usage [methodologies and frameworks](#)

Other CSS [methodologies](#)

Style guides <http://styleguides.io/>

Material design <https://material.io/guidelines/>

THANK YOU!