"SMACSS is becoming one of the most useful contributions to front-end discussion in years"

Scalable and Modular Architecture for CSS

A flexible guide to developing sites small and large.

by Jonathan Snook

About the Author

Hi, my name is Jonathan Snook. I am a web developer and designer who has been building websites a
a hobby since 1994 and as a professional since 1999.

I maintain a blog at Snook.ca where I write tips, tricks and bookmarks on web development. I also speak at conferences and workshops and have been thankful to have been able to travel the world to share what I know.

I’ve co-authored two books to date: The Art and Science of CSS (from Sitepoint) and Accelerated DOM Scripting (from Apress). I’ve also written for .net magazine, A List Apart, Sitepoint.com, and many more resources online and off.

Having worked on hundreds of web projects, including most recently on the successful Yahoo! Mail redesign, I’ve written this book to share my experience with building websites small and large.

I’d like to express my deepest gratitude to everybody within the community. Each and every one of you make this a career that I continue to enjoy having. A special thank you to Kitt Hodsden for pushing me to write this and share it with everyone. Lastly, to my boys, Hayden and Lucas, who continue to push me to be a better person.

Introduction

I have long lost count of how many websites I’ve built. You would think after having built a few hundred of them I would have discovered the “one true way” of doing it. I don’t think there is one tru way. What I *have* discovered are techniques that can keep CSS more organized and more structured, leading to code that is easier to build and easier to maintain.

I have been analyzing my process (and the process of those around me) and figuring out how best to structure code for projects on a larger scale. The concepts were vaguely there with the smaller sites that I had worked on but have become more concrete as a result of working on increasingly complex projects. Small sites don’t often hit the same pain points as larger sites or working with larger teams; small sites aren’t as complex and don’t change as often. However, what I describe in these pages is a approach that works equally well for sites small and large.

SMACSS (pronounced “smacks”) is more style guide than rigid framework. There is no library withi here for you to download or install. SMACSS is a way to examine your design process and as a way t fit those rigid frameworks into a flexible thought process. It is an attempt to document a consistent approach to site development when using CSS. And really, who isn’t building a site with CSS these days?! Feel free to take this in its entirety or use only the parts that work best for you. Or don’t use it at all. I understand that this won’t be everybody’s cup of tea. When it comes to web development, the answer to most questions is “it depends”.

What’s in here?

My thoughts have been compartmentalized around a number of topics related to CSS architecture. Each thought is detailed in its own section. Read the sections in sequence or out of order or pick and choose what seems most relevant to you. It’s not 1000 pages of writing; the sections are relatively short and easy to digest.

Now get started and dive in!

Categorizing CSS Rules

Every project needs some organization. Throwing every new style you create onto the end of a single file would make finding things more difficult and would be very confusing for anybody else working on the project. Of course, you likely have some organization in place already. Hopefully, what you read among these pages will highlight what works with your existing process and, if I'm lucky, you will see new ways in which you can improve your process.

How do you decide whether to use ID selectors, or class selectors, or any number of selectors that are at your disposal? How do you decide which elements should get the styling magic you wish to bestov upon it? How do you make it easy to understand how your site and your styles are organized?

At the very core of SMACSS is categorization. By categorizing CSS rules, we begin to see patterns and can define better practices around each of these patterns.

There are five types of categories:

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

We often find ourselves mixing styles across each of these categories. If we are more aware of what we are trying to style, we can avoid the complexity that comes from intertwining these rules.

Each category has certain guidelines that apply to it. This somewhat succinct separation allows us to ask ourselves questions during the development process. How are we going to code things and *why* ar we going to code them that way?

Much of the purpose of categorizing things is to codify patterns—things that repeat themselves withi our design. Repetition results in less code, easier maintenance, and greater consistency in the user experience. These are all wins. Exceptions to the rule can be advantageous but they should be justified.

Base rules are the defaults. They are almost exclusively single element selectors but it could include attribute selectors, pseudo-class selectors, child selectors or sibling selectors. Essentially, a base styl says that wherever this element is on the page, it should look like *this*.

Examples of Base Styles

```
html, body, form { margin: 0; padding: 0; }
input[type=text] { border: 1px solid #999; }
a { color: #039; }
a:hover { color: #03C; }
```

Layout rules divide the page into sections. Layouts hold one or more modules together.

Modules are the reusable, modular parts of our design. They are the callouts, the sidebar sections, th
product lists and so on.

State rules are ways to describe how our modules or layouts will look when in a particular state. Is it hidden or expanded? Is it active or inactive? They are about describing how a module or layout looks on screens that are smaller or bigger. They are also about describing how a module might look in different views like the home page or the inside page.

Finally, **Theme rules** are similar to state rules in that they describe how modules or layouts might look. Most sites don’t require a layer of theming but it is good to be aware of it.

Naming Rules

By separating rules into the five categories, naming convention is beneficial for immediately understanding which category a particular style belongs to and its role within the overall scope of th page. On large projects, it is more likely to have styles broken up across multiple files. In these cases naming convention also makes it easier to find which file a style belongs to.

I like to use a prefix to differentiate between Layout, State, and Module rules. For Layout, I use `l-` bu `layout-` would work just as well. Using prefixes like `grid-` also provide enough clarity to separate layout styles from other styles. For State rules, I like `is-` as in `is-hidden` or `is-collapsed`. This helps describe things in a very readable way.

Modules are going to be the bulk of any project. As a result, having every module start with a prefix like `.module-` would be needlessly verbose. Modules just use the name of the module itself.

Example classes

```
/* Example Module */
.example { }

/* Callout Module */
.callout { }

/* Callout Module with State */
.callout.is-collapsed { }

/* Form field module */
.field { }

/* Inline layout  */
.l-inline { }
```

Related elements within a module use the base name as a prefix. On this site, code examples use `.ex` and the captions use `.exm-caption`. I can instantly look at the caption class and understand that it is related to the code examples and where I can find the styles for that.

Modules that are a variation on another module should also use the base module name as a prefix. Sub-classing is covered in more detail in the Module Rules chapter.

This naming convention will be used throughout these pages. Like most other things that I have outlined here, don’t feel like you have to stick to these guidelines rigidly. Have a convention, document it, and stick to it.

Base Rules

A Base rule is applied to an element using an element selector, a descendent selector, or a child selector, along with any pseudo-classes. It doesn't include any class or ID selectors. It is defining the default styling for how that element should look in all occurrences on the page.

Example Base Styles

```
body, form {
    margin: 0;
    padding: 0;
}

a {
    color: #039;
}

a:hover {
    color: #03F;
}
```

Base styles include setting heading sizes, default link styles, default font styles, and body backgrounds. There should be no need to use `!important` in a Base style.

I highly recommended that you specify a body background. Some users may define their own background as something other than white. If you work off the expectation that the background will b white, your design may look broken. Worse, your font colour choice may clash with the user's settin and make your site unusable.

CSS Resets

A CSS Reset is a set of Base styles designed to strip out—or *reset*—the default margin, padding, and other properties. Its purpose is to define a consistent foundation across browsers to build the site on.

Many reset frameworks can be overly aggressive and can introduce more problems than they solve. Removing margin and padding from elements only to introduce them again creates duplicated effort and increases the amount of code needed to be sent to the client.

Many find resetting styles a helpful tool in site development. Just be sure to understand the drawbac of the framework you wish to use and plan accordingly.

Developing your own set of default styles that you consistently use from project to project can also b advantageous.

Layout Rules

CSS, by its very nature, is used to lay elements out on the page. However, there is a distinction between layouts dictating the major and minor components of a page. The minor components—such as a callout, or login form, or a navigation item—sit within the scope of major components such as a header or footer. I refer to the minor components as Modules and will dive into those in the next section. The major components are referred to as Layout styles.

Layout styles can also be divided into major and minor styles based on reuse. Major layout styles suc as header and footer are traditionally styled using ID selectors but take the time to think about the elements that are common across all components of the page and use class selectors where appropriate.

Layout declarations

```
#header, #article, #footer {
    width: 960px;
    margin: auto;
}

#article {
    border: solid #CCC;
    border-width: 1px 0 0;
}
```

Some sites may have a need for a more generalized layout framework (for example, 960.gs). These minor Layout styles will use class names instead of IDs so that the styles can be used multiple times on the page.

Generally, a Layout style only has a single selector: a single ID or class name. However, there are times when a Layout needs to respond to different factors. For example, you may have different layouts based on user preference. This layout preference would still be declared as a Layout style and used in combination with other Layout styles.

Use of a higher level Layout style affecting other Layout styles.

```
#article {
    float: left;
}

#sidebar {
    float: right;
}

.l-flipped #article {
    float: right;
}

.l-flipped #sidebar {
    float: left;
}
```

In the Layout example, the .l-flipped class is applied on a higher level element such as the body

element and allows the article and sidebar content to be swapped, moving the sidebar from the right t
the ~~left and vice versa for the article.~~

Using two Layout styles together to switch from fluid to fixed layout.

```
#article {
    width: 80%;
    float: left;
}

#sidebar {
    width: 20%;
    float: right;
}

.l-fixed #article {
    width: 600px;
}

.l-fixed #sidebar {
    width: 200px;
}
```

In this last example, the `.l-fixed` class modifies the design to change the layout from fluid (using percentages) to fixed (using pixels).

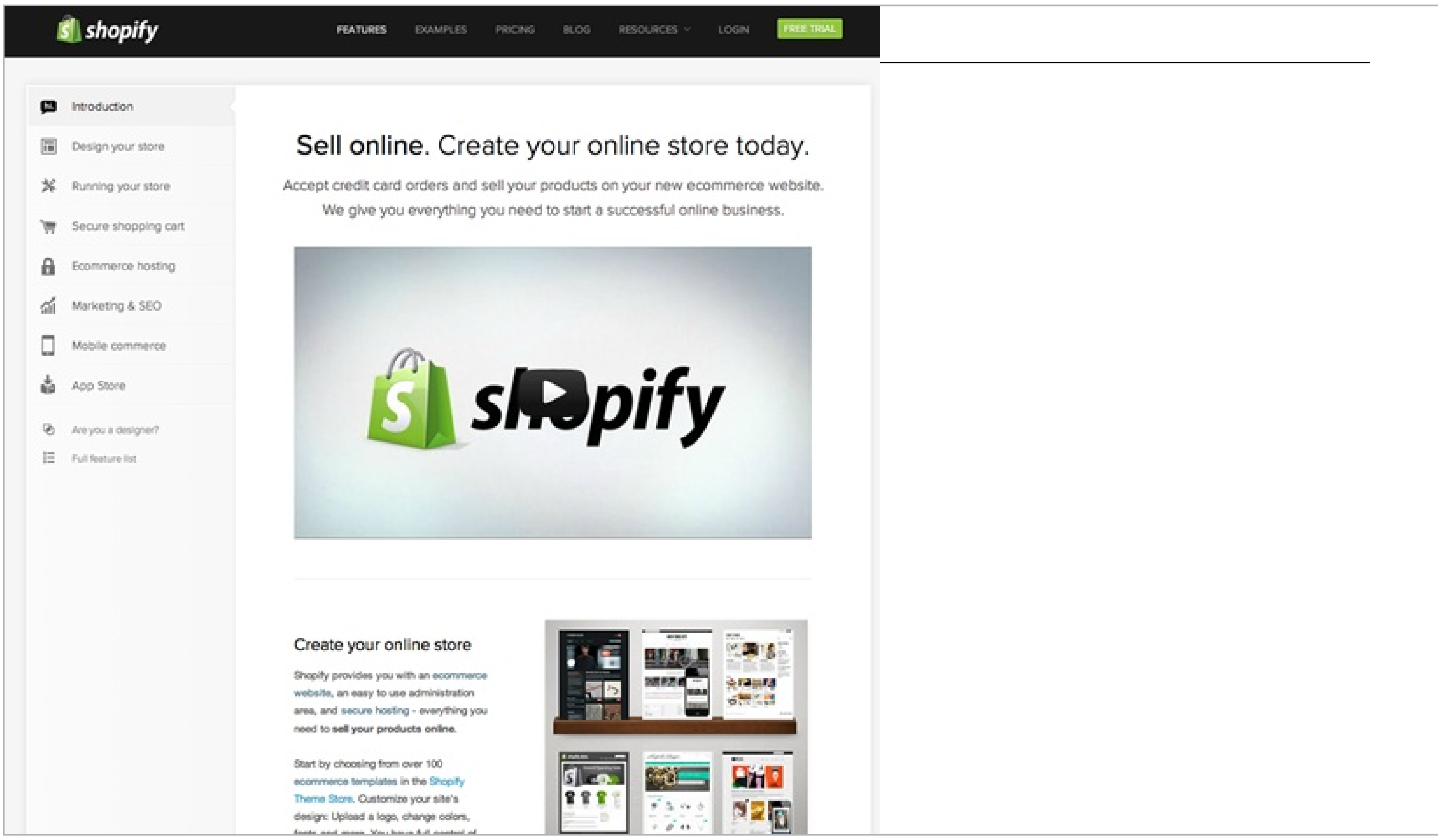
One other thing to note in the Layout example is the naming convention that I have used. The declarations that use ID selectors are named accurately and with no particular namespacing. The clas based selectors, however, *do* use an `l-` prefix. This helps easily identify the purpose of these styles and separate them from Modules or States. Layout styles are the only primary category type to use II selectors, if you choose to use them at all. If you wish to namespace your ID selectors, you can, but it is not as necessary to do so.

Using ID selectors

To be clear, using ID attributes in your HTML can be a good thing and in some cases, absolutely necessary. For example, they provide efficient hooks for JavaScript. For CSS, however, ID selectors aren’t necessary as the performance difference between ID and class selectors is nearly non-existent and can make styling more complicated due to increasing specificity.

Layout Examples

Theory is one thing but application is another. Let’s take a look at an actual website and consider wh is part of the layout and what is a module.



Sell online. Create your online store today.

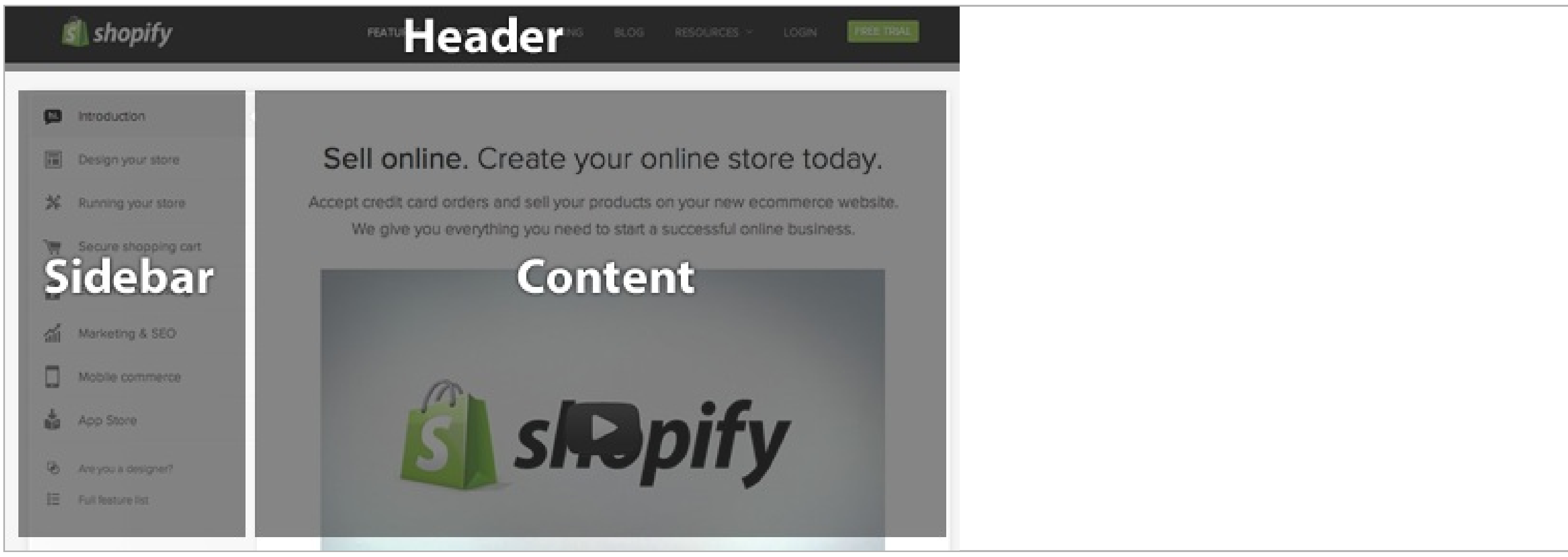
Accept credit card orders and sell your products on your new ecommerce website. We give you everything you need to start a successful online business.

Create your online store

Shopify provides you with an ecommerce website, an easy to use administration area, and secure hosting - everything you need to sell your products online.

Start by choosing from over 100 ecommerce templates in the Shopify Theme Store. Customize your site's design: Upload a logo, change colors,

In taking a look at the Shopify website, there are patterns that occur in the vast majority of websites. For example, there is a header, a main content area, a sidebar, and a footer.



Sell online. Create your online store today.

Accept credit card orders and sell your products on your new ecommerce website. We give you everything you need to start a successful online business.

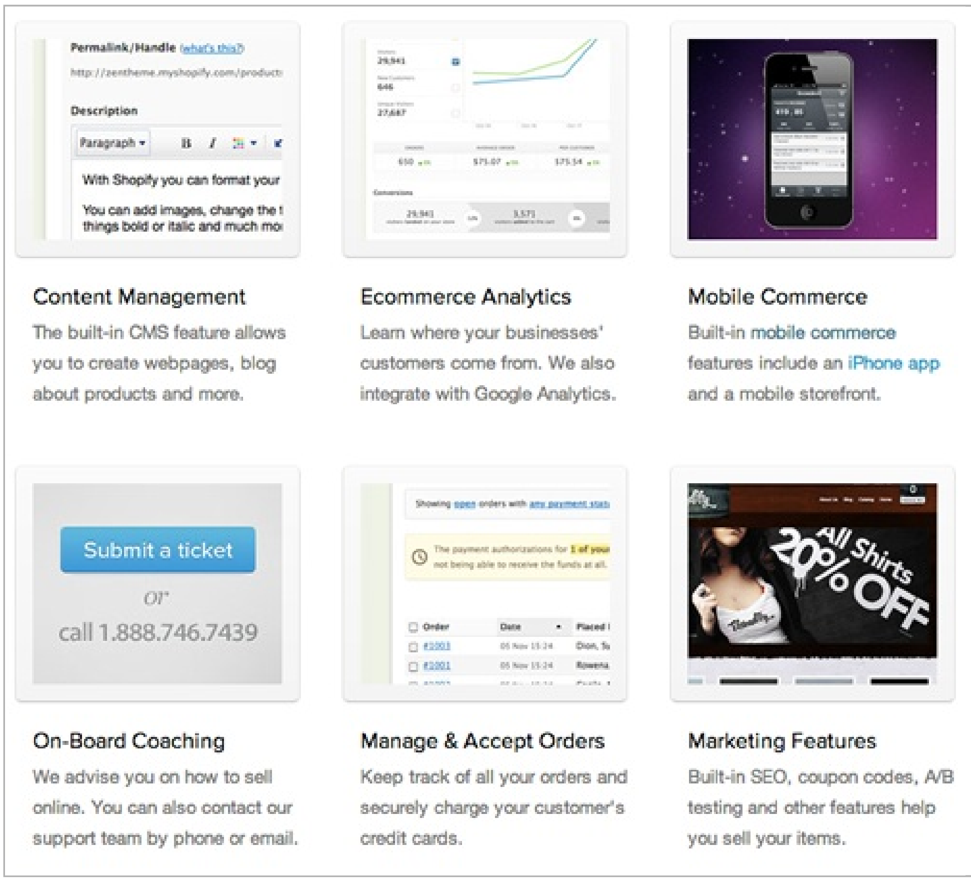
In your head, imagine what the HTML would look like. It’s likely to be a set of `div`s. Maybe you’re using HTML5 and starting to use `header` and `footer` elements. In either case, you probably would give each of the containers an ID.

Our CSS structure might look something like this:

```
#header { … }
#sidebar { … }
#maincontent { … }

<div id="header"></div>
<div id="sidebar"></div>
<div id="maincontent"></div>
```

That was straightforward and I'm sure you are thinking, "Really? You're showing me how to do this?!" Let's take a look at another part of the page.



Content Management
The built-in CMS feature allows you to create webpages, blog about products and more.

Ecommerce Analytics
Learn where your businesses' customers come from. We also integrate with Google Analytics.

Mobile Commerce
Built-in mobile commerce features include an iPhone app and a mobile storefront.

On-Board Coaching
We advise you on how to sell online. You can also contact our support team by phone or email.

Manage & Accept Orders
Keep track of all your orders and securely charge your customer's credit cards.

Marketing Features
Built-in SEO, coupon codes, A/B testing and other features help you sell your items.

Taking a look at the Features section, we see a grid of items. Shopify's markup is a container `div` wit a series of child `div`s. An unordered list may also be a useful way to mark up these items, which is what I will use for this example.

Example HTML code for the Features section layout

```
<div>
<h2>Features</h2>
<ul>
    <li><a href="…">…</a></li>
    <li><a href="…">…</a></li>
    …
</ul>
</div>
```

Without considering the SMACSS approach to this, we might be inclined to add an ID of `features` t the surrounding DIV and then style up the contents from there.

A possible approach to styling the list of featured items

```
div#featured ul {
    margin: 0;
    padding: 0;
    list-style-type: none;
}

div#featured li {
    float: left;
    height: 100px;
    margin-left: 10px;
}
```

There are some assumptions that we make with this approach:

1. There will only ever be one features grid on the page
2. List items are floated to the left
3. List items have a height of 100 pixels

These may be reasonable assumptions to make. This is a prime example of where a small site can get away with this structure: it is unlikely to change and it is unlikely to become more complex than it already is. *Maybe*. Larger sites with a higher rate of change just have a higher chance of refactoring a component within the page and needing to readdress the styling that goes with it.

Looking back at the code example, there are definitely some optimizations that could be made. The I selector didn’t need to be qualified with a tag selector and since the list is a direct descendant of the `div`, the child selector (>) could’ve been used.

Let’s take a look at how this could be readdressed to give us some more flexibility.

From a layout perspective, all we care about is how each item relates to each other. We don’t necessarily care about the design of the modules themselves nor do we want to have to worry about the context that this layout sits within.

Grid Module applied to OL or UL.

```
.l-grid {
    margin: 0;
    padding: 0;
    list-style-type: none;
}

.l-grid > li {
    display: inline-block;
    margin: 0 0 10px 10px;

    /* IE7 hack to mimic inline-block on block elements */
    *display: inline;
    *zoom: 1;
}
```

What problems were solved with this approach and what problems did we introduce? (Very rarely does *any* solution solve 100% of the problem.)

1. The grid layout can now be applied to any container to create a float-style layout
2. ~~We have decreased the *depth of applicability* by 1 (See the chapter on Depth of Applicabili~~ty fo more on that)
3. We have reduced the specificity of the selectors
4. The height requirement has been removed. A particular row will grow to the height of the talles item in that row.

On the flip-side, how did we make things worse?

1. By using a child selector, we are locking out IE6. (We could get around this by avoiding the chil selector.)
2. The CSS has increased in size and in complexity.

The increase in size can’t be disputed but it is nominal. Now that we have this reusable module, we can apply it throughout the site without code duplication. The increase in complexity is also nominal We did have to work around outdated browsers and thrown in hacks that may be frowned upon by some. However, the selectors are less complex which allow us to extend this layout while still minimizing the impact of specificity.

Module Rules

As briefly mentioned in the previous section, a Module is a more discrete component of the page. It i
your navigation bars and your carousels and your dialogs and your widgets and so on. This is the mea
of the page. Modules sit inside Layout components. Modules can sometimes sit within other Module
too. Each Module should be designed to exist as a standalone component. In doing so, the page will b
more flexible. If done right, Modules can easily be moved to different parts of the layout without
breaking.

When defining the rule set for a module, avoid using IDs and element selectors, sticking only to clas
names. A module will likely contain a number of elements and there is likely to be a desire to use
descendent or child selectors to target those elements.

Module example

```
.module > h2 {
    padding: 5px;
}

.module span {
    padding: 5px;
}
```

Avoid element selectors

Use child or descendant selectors with element selectors if the element selectors will and can be
predictable. Using `.module span` is great if a span will predictably be used and styled the same way
every time while within that module.

Styling with generic element

```
<div class="fld">
    <span>Folder Name</span>
</div>

/* The Folder Module */
.fld > span {
    padding-left: 20px;
    background: url(icon.png);
}
```

The problem is that as a project grows in complexity, the more likely that you will need to expand a
component’s functionality and the more limited you will be in having used such a generic element
within your rule.

Styling with generic element

```
<div class="fld">
    <span>Folder Name</span>
    <span>(32 items)</span>
</div>
```

Now we are in a pickle. We don’t want the icon to appear on both elements within our folder module
Which leads me to my next point:

Only include a selector that includes semantics. A span or div holds none. A heading has some. A class defined on an element has plenty.

Styling with generic element

```
<div class="fld">
    <span class="fld-name">Folder Name</span>
    <span class="fld-items">(32 items)</span>
</div>
```

By adding the classes to the elements, we have increased the semantics of what those elements mean and removed any ambiguity when it comes to styling them.

If you do wish to use an element selector, it should be within one level of a class selector. In other words, you should be in a situation to use child selectors. Alternatively, you should be extremely confident that the element in question will not be confused with another element. The more semantically generic the HTML element (like a span or div), the more likely it will create a conflict down the road. Elements with greater semantics like headings are more likely to appear by themselve within a container and you are more likely able to use an element selector successfully.

New Contexts

Using the module approach also allows us to better understand where context changes are likely to occur. The need for a new positioning context, for example, is likely to happen at either the layout level or at the root of a module.

Subclassing Modules

When we have the same module in different sections, the first instinct is to use a parent element to style that module differently.

Subclassing

```
.pod {
    width: 100%;
}
.pod input[type=text] {
    width: 50%;
}
#sidebar .pod input[type=text] {
    width: 100%;
}
```

The problem with this approach is that you can run into specificity issues that require adding even more selectors to battle against it or to quickly fall back to using `!important`.

Expanding on our example pod, we have an input with two different widths. Throughout the site, the input has a label beside it and therefore the field should only be half the width. In the sidebar, however, the field would be too small so we increase it to 100% and have the label on top. All looks well and good. Now, we need to add a new component to our page. It uses most of the same styling as a `.pod` and so we re-use that class. However, this pod is special and has a constrained width no matte where it is on the site. It is a little different, though, and needs a width of 180px.

Battling against specificity

```
.pod {
    width: 100%;
}
.pod input[type=text] {
    width: 50%;
}
#sidebar .pod input[type=text] {
    width: 100%;
}

.pod-callout {
    width: 200px;
}
#sidebar .pod-callout input[type=text],
.pod-callout input[type=text] {
    width: 180px;
}
```

We are doubling up on our selectors to be able to override the specificity of `#sidebar`.

What we should do instead is recognize that the constrained layout in the sidebar is a subclass of the pod and style it accordingly.

Battling against specificity

```
.pod {
    width: 100%;
}
.pod input[type=text] {
    width: 50%;
}
.pod-constrained input[type=text] {
    width: 100%;
}

.pod-callout {
    width: 200px;
}
.pod-callout input[type=text] {
    width: 180px;
}
```

With sub-classing the module, both the base module and the sub-module class names get applied to the HTML element.

Sub-module class name in HTML

```
 <div class="pod pod-constrained">...</div>
<div class="pod pod-callout">...</div>
```

Try to avoid conditional styling based on location. If you are changing the look of a module for usage elsewhere on the page or site, sub-class the module instead.

To help battle against specificity (and if IE6 isn't a concern), then you can double up on your class names like in the next example.

Subclassing

```
.pod.pod-callout { }

<!-- In the HTML -->
<div class="pod pod-callout"> ... </div>
```

You may be concerned about this, depending on the order of loading. For example, on Yahoo! Mail, we have code coming from different places. We had our base button styles and then we had a special set of buttons for the compose screen. However, when you clicked to add a contact to your address book, it loaded a component from a different product: Address Book. (Yes, the address book is a different product within Yahoo!.) The address book loaded its own base button styles, thereby overwriting the sub-classed button styles that we had.

If load order is a factor in your project, watch out for specificity issues.

While more specific layout components assigned with IDs could be used to provide specialized stylin for modules, sub-classing the module will allow the module to be moved to other sections of the site more easily and you will avoid increasing the specificity unnecessarily.

State Rules

A state is something that augments and overrides all other styles. For example, an accordion section may be in a collapsed or expanded state. A message may be in a success or error state.

States are generally applied to the same element as a layout rule or applied to the same element as a base module class.

State applied to an element

```
<div id="header" class="is-collapsed">
    <form>
        <div class="msg is-error">
            There is an error!
        </div>
        <label for="searchbox" class="is-hidden">Search</label>
        <input type="search" id="searchbox">
    </form>
</div>
```

The header element just has an ID. As such we can assume that any styles, if there are any, on this element are for layout purposes and that the `is-collapsed` represents a collapsed state. One might presume that without this state rule, the default is an expanded state.

The `msg` module is simple enough and has an error state applied to it. One could imagine a success state could be applied to the message, alternatively.

Finally, the field label has a hidden state applied to hide it from sight but still keep it for screen readers. In this case, we are actually applying the state to a base element and not overriding a layout module.

Isn’t it just a module?

There is plenty of similarity between a sub-module style and a state style. They both modify the existing look of an element. However, they differ in two key ways:

1. State styles can apply to layout and/or module styles; and
2. State styles indicate a JavaScript dependency.

It is this second point that is the most important distinction. Sub-module styles are applied to an element at render time and then are never changed again. State styles, however, are applied to elements to indicate a change in state while the page is still running on the client machine.

For example, clicking on a tab will activate that tab. Therefore, an `is-active` or `is-tab-active` cla
is appropriate. Clicking on a dialog close button will hide the dialog. Therefore, an `is-hidden` class i
appropriate.

Using !important

States should be made to stand alone and are usually built of a single class selector.

Since the state will likely need to override the style of a more complex rule set, the use of `!importar` is allowed and, dare I say, recommended. (I used to say that `!important` was never needed but on complex systems, it is often a necessity.) You won’t normally have two states applied to the same module or two states that tend to affect the same set of styles, so specificity conflicts from using `!important` should be few and far between.

With that said, be cautious. Leave `!important` off until you actually and truly need it (and you will see why in this next example). Remember, the use of `!important` should be avoided for all other rule types. Only states should have it.

Combining State Rules with Modules

Inevitably, a state rule will not be able to rely on inheritance to apply its style in the right place. Sometimes a state is very specific to a particular module where styling is very unique.

In a case where a state rule is made for a specific module, the state class name should include the module name in it. The state rule should also reside with the module rules and not with the rest of the global state rules.

State rules for modules

```
.tab {
    background-color: purple;
    color: white;
}

.is-tab-active {
    background-color: white;
    color: black;
}
```

If you are doing just-in-time loading of your CSS, generic states should be considered part of the bas and global styles and loaded on initial page load. The styles for a particular module won’t need to be loaded until that particular module is loaded.

Theme Rules

Theme Rules aren’t used as often within a project and because of that I was quite reluctant to include them as their own category. Some projects do have a need for them, though, as we did when working on Yahoo! Mail.

It is probably self-evident but a theme defines colours and images that give your application or site i look and feel. Separating the theme out into its own set of styles allows for those styles to be easily redefined for alternate themes. The need for theming within a project is necessary when you want the user to receive an alternate skin that provides some cosmetic alterations.

For example, your site may have different colours for different sections of the site. Or you may allow users to customize the colour based on a user preference. Or you may need to provide themes based o locale such as country or language.

Themes

Themes can affect any of the primary types. They can override base styles like default link colours. They can change module elements such as colours and borders. They can affect layout with different arrangements. They can also alter how states look.

Let’s say you have a dialog module that needs to have a border colour of blue, the border itself would be initially defined in the module and then the theme defines the colour:

Module Theming

```
/* in module-name.css */
.mod {
    border: 1px solid;
}

/* in theme.css */
.mod {
    border-color: blue;
}
```

Depending on how extensive the theming is, it may be easier to define theme-specific classes. In the case of Yahoo! Mail, we kept the theming to specific regions of the page. This made it easier for us t build new themes without sacrificing the overall design balanced with still giving the user some customization.

For more extensive theming, using a `theme-` prefix for specific theme components will make it easie to apply them to more elements on the page.

Theme Classes

```
/* in theme.css */
.theme-border {
    border-color: purple;
}

.theme-background {
    background: linear-gradient( ... );
}
```

At Yahoo! Mail, to help with maintaining consistency across all of our theme files—they have over 5
—we used a Mustache template for our CSS that allowed us to specify a number of colour values, a
background image, and create a final CSS file for production.

Typography

As a facet of theming, there are times when you need to redefine the fonts that are being used on a wholesale basis, such as with internationalization. Locales such as China and Korea have complex ideograms that are difficult to read at smaller font sizes. As a result, defining specific rules to isolate font styles makes it easier to change font size across multiple components.

Font rules will normally affect base, module and state styles. Font styles won't normally be specified at the layout level as layouts are intended for positioning and placement, not for stylistic changes lik fonts and colours.

Like theme files, there may not be need to define actual font classes (like `font-large`). If you do, your site should only have 3 to 6 different font-sizes. If you have more than 6 font sizes declared in your project, your users will likely not notice and are making the site harder for you to maintain.

What's in a name

Naming theme and typography classes are usually the hardest to feel comfortable with because we're in an industry that considers them *unsemantic*. In the case of theme components, they're inherently visual and unsemantic. In the case of typography, though, this isn't really the case. Design is about visual hierarchy afterall, and your typography should reflect that. Therefore, the naming convention you end up using should indicate the various levels of importance, just as you would with heading levels in HTML.

Changing State

You’ve got a Photoshop document open in front of you and you have been told to turn it into the magic that is HTML and CSS (with maybe a little JavaScript thrown in for good measure).

It may seem straightforward to start mapping things directly from the composition to the code. However, various components on your page are likely to need to be represented in various states. There is the default state that something should appear in and then what it should look like when the state changes.

What is a state change?

State changes are represented in one of three ways:

1. class name
2. pseudo-class
3. media query

A **class name** change happens with JavaScript. Via some interaction, be it moving the mouse around hitting something on the keyboard, or some other event occurring. An element gets a new class applied and then the visual appearance changes.

A **pseudo-class** change is done via any number of pseudo-classes, and there are a lot. In these cases, we no longer have to rely on JavaScript to describe the state change. Pseudo-classes are still limited i that we can only style changes to elements that are descendants or siblings of the element in which th pseudo-class applies. Otherwise, we are back to using JavaScript.

Lastly, **media queries** describe how things should by styled under defined criteria, such as different viewport sizes.

With a module-based system, it is important to consider state-based design as applied to each of the modules. When you actively ask yourself, “what is the default state,” then you’ll find yourself thinking proactively about progressive enhancement. It also can have you approaching issues slightly differently.

Change via Class Name

For the most part, class name changes are straightforward. These are applied to elements that take on different state. For example, a user clicks on a disclosure icon to show and hide an element on the page.

JavaScript changing state via class name

```
// with jQuery
$('.btn-close').click(function(){
    $(this).parents('.dialog').addClass('is-hidden');
})
```

The jQuery example adds a click event handler to every element with the `btn-close` class name. When the user clicks on the button, it takes the event source and works up the DOM tree to find the ancestor element with the class of `dialog` on it. Then it applies the `is-hidden` state class.

Other times, a state change has a greater impact.

A common interface design pattern is that of a button being pressed and displaying a menu. In this case, the menu changes to a pressed state and the menu changes to a visible state. What options do w have for handling this change? It depends heavily on your HTML structure. For example, at Yahoo!, menus get loaded at request time and are, therefore, inserted at the top of the DOM. We had used a naming convention to hook the two together.

Button and menu in separate parts of the same document

```
<div id="content">
   <div class="toolbar">
      <button id="btn-new" class="btn" data-action="menu">New</button>
   </div>
</div>
<div id="menu-new" class="menu">
   <ul> ... </ul>
</div>
```

The data-action tied into a JavaScript click call that said, "hey, you want to load a menu." It would take the button ID and find the menu that matched. This is how it might work with jQuery:

Loading Menu with jQuery

```
// bind a click handler to the button
$('#btn-new').click(function(){
    // wrap the clicked button in jQuery
    var el = $(this);

    // change the state of the button
    el.addClass('is-pressed');

    // find the menu by stripping btn- and
    // adding it to menu selector
    $('#menu-' + el.id.substr(4)).removeClass('is-hidden');
});
```

As this illustrates, the state change for a single item is modified on two different items in two different places via JavaScript.

But what if the menu actually resided right next to the button?

Button and menu in the same part of the document

```
<div id="content">
   <div class="toolbar">
      <button id="btn-new" class="btn" data-action="menu">New</button>
      <div id="menu-new" class="menu">
          <ul> ... </ul>
      </div>
   </div>
</div>
```

The previous code would work exactly the same and could definitely stay the same. However, we hav alternatives. Your first instinct might be to add a class to a parent element and style the button and menu from there.

Adding a class to parent element to style child elements

```
<div id="content">
   <div class="toolbar is-active">
      <button id="btn-new" class="btn" data-action="menu">New</button>
      <div id="menu-new" class="menu">
          <ul> ... </ul>
      </div>
   </div>
</div>

/* CSS for styling */

.is-active .btn { color: #000; }
.is-active .menu { display: block; }
```

The problem with this approach is that this HTML structure is now tied together. There must be a containing element. The menu and button must exist within that containing element. Let’s hope we don’t need to add any more buttons into that toolbar!

Another approach to this is to apply the active class to the button as we did before and use the sibling selector to activate the menu.

Activating the menu with a sibling selector

```
<div id="content">
   <div class="toolbar">
      <button id="btn-new" class="btn is-active" data-action="menu">New</button>
      <div id="menu-new" class="menu">
          <ul> ... </ul>
      </div>
   </div>
</div>

/* CSS for styling */

.btn.is-active { color: #000; }
.btn.is-active + .menu { display: block; }
```

I prefer this approach over applying a state class to a parent element as the state is more accurately combined with the module in which it applies. It still has the dependency of tying the menu HTML with the button HTML: one has to come immediately after the other. If you can establish that consistency in your project then this is an approach that can work well for you.