Scalable and Modular Architecture for CSS

A flexible guide to developing sites small and large.

by Jonathan Snook

Scalable and Modular Architecture for CSS

By Jonathan Snook

SMACSS: Scalable and Modular Architecture for CSS
http://smacss.com

ISBN 978-0-9856321-0-6

Snook.ca Web Development, Inc.
Ottawa, Ontario, Canada
http://snook.ca

Second edition

About the Author

Hi, my name is Jonathan Snook. I am a web developer and designer who has been building websites as 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 true 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 an approach that works equally well for sites small and large.

SMACSS (pronounced "smacks") is more style guide than rigid framework. There is no library within here for you to download or install. SMACSS is a way to examine your design process and as a way to 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 bestow 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* are we going to code them that way?

Much of the purpose of categorizing things is to codify patterns—things that repeat themselves within 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 style 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, the 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 the 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-` but `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 `.exm` 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 be white, your design may look broken. Worse, your font colour choice may clash with the user's setting 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 drawbacks 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 be 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 such 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[1]). These minor Layout styles will use class names instead of IDs so that the styles can be used multiple times on the page.

1.http://960.gs/

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 to 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 class-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 ID 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

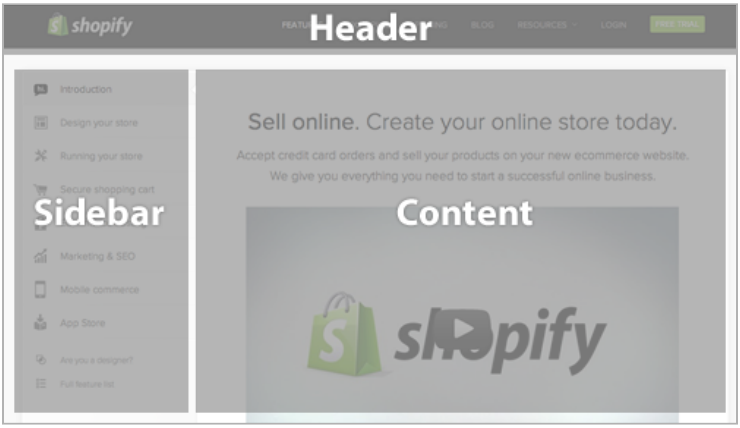
To be clear, using ID attributes in your HTML can be a good thing and in some cases, absolutely necessary. For example, they provide

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 what is part of the layout and what is a module.

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.

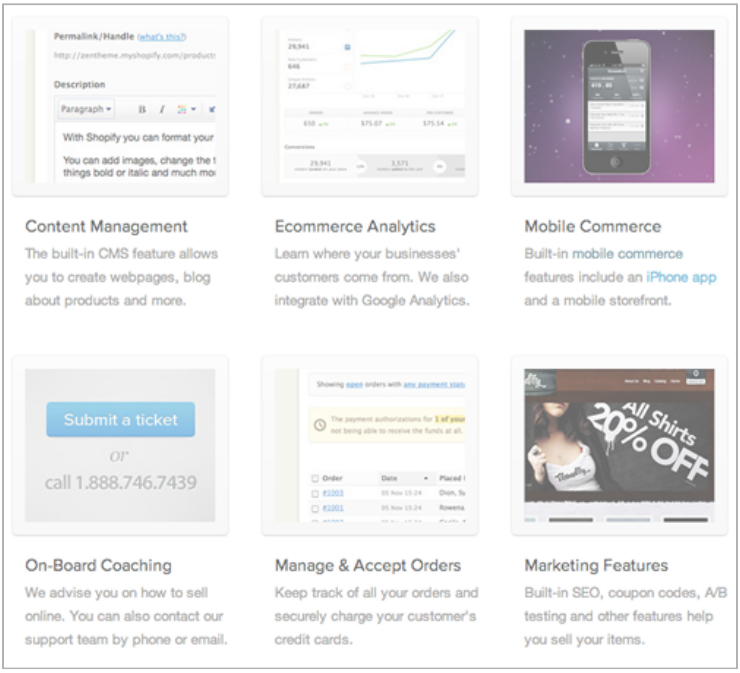


In your head, imagine what the HTML would look like. It's likely to be a set of `div`s. Maybe you're using HTML5 and starting to use `header` and `footer` elements. In either case, you probably would give each of the containers an ID.

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.



Taking a look at the Features section, we see a grid of items. Shopify's markup is a container `div` with 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` to 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 ID 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 Applicability for 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 tallest 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 child 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 is your navigation bars and your carousels and your dialogs and your widgets and so on. This is the meat of the page. Modules sit inside Layout components. Modules can sometimes sit within other Modules, too. Each Module should be designed to exist as a standalone component. In doing so, the page will be 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 class 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 themselves 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 matter 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 styling 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 or 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` class is appropriate. Clicking on a dialog close button will hide the dialog. Therefore, an `is-hidden` class is 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 `!important` 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 base 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 its 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 on 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 to 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 easier 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 50—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 like 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 in that we can only style changes to elements that are descendants or siblings of the element in which the 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 a 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 we 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 have 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.

Why parent and sibling states are problematic

The reason why this approach can be more troublesome over just applying a state to each module is that it is no longer clear where this rule set should go. The menu is no longer just a menu. It's a button menu. If you needed to modify the active state for this module, do you find the CSS in with the button CSS or is it in with the menu CSS?

All that to say that applying a state to each button is the preferred approach. You're creating a better separation between the modules, making your site easier to test, develop, and scale.

Handling State Change with Attribute Selectors

Depending on your browser support, you can also take advantage of attribute selectors to handle state change. This can be useful for:

- isolating states from layout and module classes
- allowing easier transitions between multiple states

Let's take a look at an example of a button that can be in multiple states such as default, pressed or disabled.

You may choose to use a sub-module naming convention.

Sub-module naming convention

```
.btn { color: #333; }
.btn-pressed { color: #000; }
.btn-disabled { opacity: .5; pointer-events: none; }
```

If a button needs to be toggled between states then it might make more sense to use a state naming convention.

State naming convention

```
.btn { color: #333; }
.is-pressed { color: #000; }
.is-disabled { opacity: .5; pointer-events: none; }
```

I like the comparison between these two examples because it highlights that SMACSS is often about clarity and naming convention. I would be happy to see either of these two examples within a project. Now let's take a look at another approach: *attribute selectors*.

Attribute selectors convention

```
.btn[data-state=default] { color: #333; }
.btn[data-state=pressed] { color: #000; }
.btn[data-state=disabled] { opacity: .5;
pointer-events: none; }

<!-- HTML -->
<button class="btn"
data-state="disabled">Disabled</button>
```

The `data-` prefix on the attribute is part of the HTML5 specification that allows you to make up attribute names and place them within the data namespace so as not to conflict with future HTML attribute specifications. Changing the state of a button doesn't require removing classes and adding classes. It just requires changing the value of a single attribute.

Changing State with jQuery

```
// bind a click handler to each button
$(".btn").bind("click", function(){
    // change the state to pressed
    $(this).attr('data-state', 'pressed');
});
```

Admittedly, with JavaScript libraries like jQuery, manipulating classes for state management isn't complicated. jQuery has methods like `hasClass`, `addClass`, and `toggleClass` that make working with class names really easy.

Suffice it to say, you have plenty of choices in the way you can represent state.

Class-based State Change with CSS Animations

Animations are an interesting beast and some may argue that it is defining behaviour in a layer where it shouldn't be defined. CSS is for styling, after all. JavaScript is for behaviour.

The distinction here is to understand that CSS defines a visual *state*. We can use JavaScript to switch the state of an element on our page. JavaScript should not be used to describe the state, though. That is, it shouldn't be used to add inline styles.

Historically, we have used JavaScript to create animation because it was the only way we had available to do so (HTML+TIME[2] notwithstanding).

When we think of things in these terms, it can help shape how we approach various situations. For example, it wouldn't be unusual to have a message appear on the page for a short period of time and then fade out.

JavaScript handling state change

```
function showMessage (s) {
    var el = document.getElementById('message');
    el.innerHTML = s;

    /* set state */
    el.className = 'is-visible';
    setTimeout(function(){
        /* set state back */
        el.className = 'is-hidden';
    }, 3000);
}
```

The message state changes from hidden to visible and back to hidden again. The JavaScript handles the changes in these states and then CSS can be used to animate between these—using either CSS transitions or animations.

2.http://www.w3.org/TR/NOTE-HTMLplusTIME

CSS handling the transition

```
@keyframes fade {
    0% { opacity:0; display:block; }
  100% { opacity:1; display:block; }
}

.is-visible {
    display: block;
    animation: fade 2s;
}

.is-hidden {
    display: none;
    animation: fade 2s reverse;
}
```

I admit, this last example wouldn't actually work. Unfortunately, the current browser implementations won't allow for us to specify non-animatable properties in our animation. Thankfully, browser implementations are in the midst of getting update to more recent CSS3 Animation recommendations. In the meantime, we need to do something like the following.

Animations in current browsers

```
@-webkit-keyframes fade {
    0% { opacity:0;  }
  100% { opacity:1; display:block; }
}

.is-visible {
    opacity: 1;
    animation: fade 2s;
}

.is-hidden {
    opacity: 0;
    animation: fade 2s reverse;
}

.is-removed {
    display: none;
}

/* and then our javascript */
function showMessage (s) {
    var el = document.getElementById('message');
    el.innerHTML = s;

    /* set state */
    el.className = 'is-visible';
    setTimeout(function(){
        /* set state back */
        el.className = 'is-hidden';
        setTimeout(function(){
            el.className = 'is-removed';
        }, 2000);
    }, 3000);
}
```

In this case, I've changed it to still do the animation but then to use JavaScript to remove the element from flow after the animation should be done.

In this way, we maintain the separation between style (aka state) and behaviour.

Change via Pseudo-class

As we've just seen, we can use classes and attributes to handle defining state changes on a module. However, CSS offers up plenty of pseudo-classes that can also help us manage states and state change.

From CSS2.1, the three most useful pseudo-classes are the "dynamic" ones that react to user interaction: `:hover`, `:focus`, and `:active`. CSS3 adds a number of new pseudo-classes, most of which style based on HTML structure (such as `:nth-child` or `:last-child`). There are a number of UI pseudo-classes in CSS3 that can respond to form interactions, which can also be quite handy.

The default state for a module is normally defined without a pseudo-class. Define the pseudo-classes as a secondary state of the module.

Defining States with pseudo-classes

```
.btn {
    background-color: #333; /* gray */
}

.btn:hover {
    background-color: #336; /* blueish */
}

.btn:focus {
    /* blueish focus ring */
    box-shadow: 0 0 3px rgba(48,48,96,.3);
}
```

As modules are sub-classed, it can potentially get complicated as you may need to design pseudo-class states for the sub-modules, as well.

Defining sub-module States with pseudo-classes

```
.btn {
    background-color: #333; /* gray */
}

.btn:hover {
    background-color: #336; /* blueish */
}

.btn:focus {
    /* blueish focus ring */
    box-shadow: 0 0 3px rgba(48,48,96,.3);
}

/* a default button state is the default choice
 * from a selection of buttons
 */
.btn-default {
    background-color: #DEDB12; /* yellowish */
}

.btn-default:hover {
    background-color: #B8B50B; /* darker yellow */
}

/* no need to define a different focus state */
```

In this last code example, we have essentially created 5 variations of a single module: a primary module, a sub-module, and then the pseudo-class states that they could appear in. It can get even more complicated when we introduce class-based states on top of each of these.

Modules, sub-modules, class states and pseudo-class states. Oh my.

```
.btn { ... }
.btn:hover { ... }
.btn:focus { ... }

.btn-default { ... }
.btn-default:hover { ... }

.btn.is-pressed { ... }
.btn.is-pressed:hover { ... }

.btn-default.is-pressed { ... }
.btn-default.is-pressed:hover { ... }
```

Thankfully, few modules in your interface are going to need quite this array of state management. However, it is clear that proper organization of your styles will ensure your project is easier to maintain.

Change via Media Query

While state changes via classes and pseudo-classes are fairly commonplace, media queries are fast becoming another approach to managing state change—a way that has traditionally only been possible with JavaScript. Adaptive design and Responsive Web Design[3] use media queries to react to various criteria. Print style sheets were one of the first media queries that allowed us to define how elements should look when printed on a page.

A media query can be defined as a separate style sheet using the media attribute on the link element or it can be defined within a `@media` block within a specific style sheet.

3.http://www.alistapart.com/articles/responsive-web-design/

Linking stylesheets

```
<link href="main.css" rel="stylesheet">
<link href="print.css" rel="stylesheet"
media="print">

/* inside main.css */
@media screen and (max-width: 400px) {
    #content { float: none; }
}
```

Most examples of media queries define a break point and then throw all styles that pertain to that particular break point and place them inside.

With SMACSS, the intent is to keep the styles that pertain to a specific module with the rest of the module. That means that instead of having a single break point, either in a main CSS file or in a separate media query style sheet, place media queries around the module states.

Modular Media Queries

```
/* default state for nav items */
.nav > li {
   float: left;
}

/* alternate state for nav items on small screens */
@media screen and (max-width: 400px) {
    .nav > li { float: none; }
}

... elsewhere for layout ...

/* default layout */
.content {
    float: left;
    width: 75%;
}

.sidebar {
    float: right;
    width: 25%;
}

/* alternate state for layout on small screens */
@media screen and (max-width: 400px) {
    .content, .sidebar {
        float: none;
        width: auto;
    }
}
```

Yes, this does mean that the media query declaration may (and likely will) get declared multiple times but it also allows for all information about a module to be kept together. Remember, keeping the module information together (especially in its own CSS file) allows for isolated testing of the module and (depending on how you build your application) allows assets such as modularized templates and CSS to be loaded after the initial page has been loaded.

It's all about State

This chapter reviewed the three types of state change: class, pseudo-class, and media query. Thinking about your interface not only modularly but as a representation of those modules in various states will make it easier to separate styles appropriately and build sites that are easier to maintain.

Depth of Applicability

When learning the inner workings of CSS, we learn that we have selectors and that we use selectors to select the HTML elements on the page that we want to style. CSS has grown over the years to give us more power using an ever increasing number of selectors. Each rule set that we add to our style sheet, however, creates an ever increasing connection between the CSS and the HTML.

Let's review a typical block of CSS that you might find on a website.

How we tightly couple our CSS to our HTML

```
#sidebar div {
    border: 1px solid #333;
}

#sidebar div h3 {
    margin-top: 5px;
}

#sidebar div ul {
    margin-bottom: 5px;
}
```

By looking at this, you can see that there is some expectation of what our HTML will look like. There is likely one or more sections in a sidebar that have a heading and an unordered list that follows it. If the site doesn't change very often, this style of CSS will work just fine. I haven't changed the design of my blog in two years. My need to scale just isn't there. If I tried using this approach on a larger site, which can change more frequently and have a greater variety of code requirements, I am going to have problems. I will need to

add more rules with more complex selectors. I may find myself in a maintenance nightmare.

Where have I gone wrong? There are two particular concerns with the example CSS:

1. I am relying heavily on a defined HTML structure.
2. The depth of HTML to which the selectors apply is too deep.

Minimizing the Depth

HTML is like a tree structure of parents and children. The depth of applicability is the number of generations that are affected by a given rule. For example, `body.article > #main > #content > #intro > p > b` would have a depth of applicability of 6 generations. If this selector was written as `.article #intro b` then the depth is still the same: 6 generations.

The problem with such a depth is that it enforces a much greater dependency on a particular HTML structure. Components on the page can't be easily moved around. If we look back at the sidebar example, how do we recreate that module in another part of the page such as a footer? We have to duplicate the rules.

Duplication of rules

```
#sidebar div, #footer div {
    border: 1px solid #333;
}

#sidebar div h3, #footer div h3 {
    margin-top: 5px;
}

#sidebar div ul, #footer div ul {
    margin-bottom: 5px;
}
```

The root node is at the `div` and it is from here that we should be creating our styles.

Simplification of rules

```
.pod {
    border: 1px solid #333;
}

.pod > h3 {
    margin-top: 5px;
}

.pod > ul {
    margin-bottom: 5px;
}
```

The pod is a container that still relies on a particular HTML structure but it is of a much shallower depth than what we had before. The trade-off is that we have to repeat the pod class on numerous elements on the page. Whereas before, we just had two elements with IDs. Of course, we want to avoid going back to the days where we did silly things like adding class names to every paragraph.

An advantage to using this shallow depth of applicability approach is also the ability to more readily convert these modules into templates for dynamic content. At Yahoo!, for example, we've been relying on Mustache for much of our template needs. Here is how we would set up our template for these pods:

An example Mustache template

```
<div class="pod">
    <h3>{{heading}}</h3>
    <ul>
        {{#items}}
        <li>{{item}}</li>
        {{/items}}
    </ul>
</div>
```

We are trying to strike a balance between maintenance, performance, and readability. Going too deep may mean less "classitis" within your HTML but it increases the maintenance and readability overhead. Likewise, you don't want (or need) classes on everything. Adding classes to the `h3` or `ul` in this example would have been a little unnecessary unless we needed to have an even more flexible system.

To go even further on this last example, this design pattern is a common one. It is a container with a header and a body. (Sometimes, you will have a footer, too.) We have a `ul` in there right now but in other examples, we might see an `ol` or a `div` in its place.

Once again, we can duplicate our rules for each variation.

Duplication of rules

```
.pod > ul, .pod > ol, .pod > div {
    margin-bottom: 5px;
}
```

Alternatively, we can classify the pod body.

Simplifying with a class

```
.pod-body {
    margin-bottom: 5px;
}
```

With the module rule approach, it is not even necessary to specify the `.pod` class. We can visually see that `.pod-body` is associated with the pod module and from a code perspective, it'll work just fine.

An example Mustache template

```
<div class="pod">
    <h3>{{heading}}</h3>
    <ul class="pod-body">
        {{#items}}
        <li>{{item}}</li>
        {{/items}}
    </ul>
</div>
```

As a result of this small change, we were able to reduce the depth of applicability to the shallowest it can go. The single selector also means that we are avoiding potential specificity issues, too. All around, that is win-win.

Selector Performance

With work, I have had to do quite a bit of examination of performance. We run a number of tools over an application to determine where the bottlenecks are. One such application is Google Page Speed[4] which provides a number of recommendations to improve JavaScript and rendering performance. Before I get into its recommendations, we need to understand a little better about how browsers evaluate CSS.

How CSS gets evaluated

The style of an element is evaluated on element creation

We often think of our pages as these full and complete documents full of elements and content. However, browsers are designed to handle documents like a stream. They begin to receive the document from the server and can render the document before it has completely downloaded. Each node is evaluated and rendered to the viewport as it is received.

4.http://code.google.com/speed/page-speed/

An example HTML document

```
<body>
   <div id="content">
      <div class="module intro">
         <p>Lorem Ipsum</p>
      </div>
      <div class="module">
         <p>Lorem Ipsum</p>
         <p>Lorem Ipsum</p>
         <p>Lorem Ipsum <span>Test</span></p>
      </div>
   </div>
</body>
```

The browser starts at the top and sees a `body` element. At this point, it thinks it is empty. It hasn't evaluated anything else. The browser will determine what the computed styles are and apply them to the element. What is the font, the color, the line height? After it figures this out, it paints it to the screen.

Next, it sees a `div` element with an ID of content. Again, at this point, it thinks it is empty. It hasn't evaluated anything else. The browser figures out the styles and then the `div` gets painted. The browser will determine if it needs to repaint the body—did the element get wider or taller? (I suspect there are other considerations but width and height changes are the most common effects child elements have on their parents.)

This process continues on until it reaches the end of the document.

For a visualization of the reflow/repaint process in Firefox, visit http://youtu.be/ZTnIxIA5KGw.

CSS gets evaluated from right to left.

To determine whether a CSS rule applies to a particular element, it starts from the right of the rule and works its way left.

If you have a rule like `body div#content p { color: #003366; }` then for every element—as it gets rendered to the page—it will first ask if it is a paragraph element. If it is then it will work its way up the DOM and ask if it is a `div` with an ID of content. If it finds what it is looking for, it will continue its way up the DOM until it reaches the `body`.

By working right to left, the browser can determine whether a rule applies to this particular element that it is trying to paint to the viewport much faster. To determine which rule is more or less performant, you need to figure out how many nodes need to be evaluated to determine whether a style can be applied to an element.

Which rules rule?

As each element gets rendered onto the page, it need to figure out which styles should be applied. Now, take a look through the Google Page Speed recommendations[5]. There are four main rules that they consider inefficient:

- Rules with descendant selectors. E.g. `#content h3`
- Rules with child or adjacent selectors. E.g. `#content > h3`
- Rules with overly qualified selectors. E.g. `div#content > h3`
- Rules that apply `:hover` to non-link elements. E.g. `div#content:hover`

What is important to note with these recommendations is that *the evaluation of any more than a single element to determine styling is inefficient*. That means that you could only ever use a single selector in your rule: a class selector, an ID selector, an element selector, or an attribute selector. If you take this recommendation at face value, they are suggesting we go back to the days of `<p`

5.http://code.google.com/speed/page-speed/docs/rendering.html#UseEfficientCSSSelectors

`class="bodytext">`. (And if you look at the CSS that they generate on products like Search and Google Mail, they follow these recommendations.)

Constrain yourself, don't choke yourself

For the rest of us, I believe that we can be a little more practical and strike a balance between one end of the spectrum (adding classes and identifiers to everything) and the other (using deep selector rules creating tight coupling between HTML and CSS).

I follow three simple guidelines to help limit the number of elements that need to be evaluated:

1. Use child selectors
2. Avoid tag selectors for common elements
3. Use class names as the right-most selector

For example, `.module h3` might be okay if I know I am only going to have a dozen H3s on my page. How deep are my H3s in the DOM? Are they 4 levels down (e.g.: `html > body > #content > h3`) or are they 10 levels down (e.g.: `html > body > #content > div > div > … > h3`)? Can I limit the DOM traversal using child selectors? If I can do `.module > h3` (sorry IE6), then I know for the 12 H3s I have on my page, it will only have to evaluate 24 elements. If I do `.module div`, however, and I have 900 divs on my page (I just loaded up my inbox in Yahoo! Mail and there are 903), then I am going to have a lot more traversal. A simple `<div><div><div></div></div></div>` (3 levels deep) results in 6 evaluations. It is factorial. 4 levels deep results is 24. 5 levels deep results is 120.

With all that said, even these simple optimizations may be overkill. Steve Souders, who works tirelessly on performance testing, examined the performance impact of CSS selectors[6] and determined (back in 2009) that the delta between the best case and the worst

6.http://www.stevesouders.com/blog/2009/03/10/performance-impact-of-css-selectors/

case was 50ms. In other words, consider selector performance but don't waste too much time on it.

HTML5 and SMACSS

As it turns out, it works just as well with HTML5 as it does with HTML4 or any other HTML you might be using with your CSS. That is because the SMACSS approach has two core goals:

1. Increase the semantic value of a section of HTML and content
2. Decrease the expectation of a specific HTML structure

HTML5 introduces a number of new elements which can help us increase the semantic value of a section of HTML and content. Tags like `section`, `header`, and `aside` are more descriptive than a simple `div`. We have new input types that allow us to differentiate a numeric field from a date field from a text field. The extra tags and attributes have allowed us to be more descriptive. That is a good thing.

But even with our new tags to play with, the tags are not necessarily (or very likely) going to describe a very specific module on the page. Is a `nav` element always going to contain the exact same type and style of navigation?

<nav> implementation

```
<nav class="nav-primary">
    <h1>Primary Navigation</h1>
    <ul>…</ul>
</nav>

<nav class="nav-secondary">
    <h1>External Links</h1>
    <ul>…</ul>
</nav>
```

The primary navigation uses a horizontal navigation along the top of the page but the secondary navigation (for a sidebar, for example) lists items vertically. The class names provide differentiation between the types.

Class names help describe our content in very specific ways—ways that are more specific than even HTML5 can provide. This serves our first goal of increasing the semantic value of a section of HTML.

Your first instinct might be to do something like the following:

<nav> CSS

```
nav.nav-primary li {
    display: inline-block;
}

nav.nav-secondary li {
    display: block;
}
```

In doing so, we have indicated that these classes may only be used on `nav` elements. If our code was never going to change, this would be okay. However, the intention of this book is to describe scalable projects, so let us look at an example of how things might change on this project.

Our primary navigation is only a single level but the client comes back and says we need to add drop down menus to every element. Our HTML structure changes.

<nav> implementation

```
<nav class="nav-primary">
    <h1>Primary Navigation</h1>
    <ul>
        <li>About Us
            <ul>
                <li>Team</li>
                <li>Location</li>
            </ul>
        </li>
    </ul>
</nav>
```

With this sub-navigation, how do we style the items such that they are listed vertically and not horizontally?

With the CSS that we already have, we would have to add a `<nav class="nav-secondary">` around each of the inner unordered lists to get the style that we want.

We can augment the CSS to target the inner list items.

Augmented <nav> CSS

```
nav.nav-primary li {
    display: inline-block;
}

nav.nav-secondary li,
nav.nav-primary li li {
    display: block;
}
```

Another alternative is to remove the need for a `nav` element to apply our class to, which works towards our secondary goal of decreasing the expectation of specific HTML.

SMACSS-style <nav> CSS

```
.l-inline li {
    display: inline-block;
}

.l-stacked li {
    display: block;
}
```

In this case, we have switched to indicate that these are Layout rules, since we are impacting how the individual modules (the list items) are to be contained. The `.l-stacked` class can be applied to the sub-navigation `ul`. This will create the result that we desire.

Specifying the list item as a required child element still binds the Layout rules to specific HTML elements. There are certainly multiple ways to skin a cat, as the saying goes. For example, you might wish to say that all child elements will take on that style.

SMACSS-style <nav> CSS

```
.l-inline > * {
    display: inline-block;
}

.l-stacked > * {
    display: block;
}
```

The downfall to this approach is that the rules will have to be evaluated for every single element on the page and not just the list items. The targeting of just direct descendants avoids too much traversal. This lets us use the inline and stacked classes on pretty much any element where we want the child elements to take on those styles.

<nav> implementation

```
<nav class="l-inline">
    <h1>Primary Navigation</h1>
    <ul>
        <li>About Us
            <ul class="l-stacked">
                <li>Team</li>
                <li>Location</li>
            </ul>
        </li>
    </ul>
</nav>
```

Even with just this rather straightforward example, we managed to keep our CSS simple and avoided making our selectors more complex. The HTML is still understandable.

Remember the two goals: increase semantics and decrease reliance on specific HTML.

Prototyping

Good programmers like patterns. Good designers like patterns, too. Patterns establish familiarity and encourage re-use. SMACSS is about identifying the patterns in your design and codifying them.

A prototype should assist in viewing components in part or in whole and to allow the codification of the design language into building blocks. The web design industry likes reusable components and can be seen in many of the frameworks like Bootstrap[7] (for a variety of site components) and 960.gs[8] (for layout grids).

At Yahoo!, the prototyping team creates these building blocks and use them for production. This allowed for greater consistency across multiple products since they were all based on exactly the same foundation.

Goals of a prototype

A prototype can serve multiple goals:

- show states
- review localization
- isolate dependencies

States

From the default state to collapsed states to error states to whatever states you have defined, it is important to be able to visualize

7.http://twitter.github.com/bootstrap/
8.http://960.gs/

each of these states and make sure that the module is built accurately.

If a module is data-driven then real or mock data can be used within your prototype to test that it will render correctly.

Localization

For projects that need to support multiple locales, it will be valuable to be able to test modules using strings from the different locales to ensure layouts don't break as a result.

Dependencies

Lastly, it is important to isolate dependencies. What CSS and JavaScript dependencies are required to render a module correctly? In larger projects where lazy loading is used, being able to isolate dependencies to the bare minimum required means that you have built a module effectively and can integrate that module into the site without negatively impacting other modules on the page.

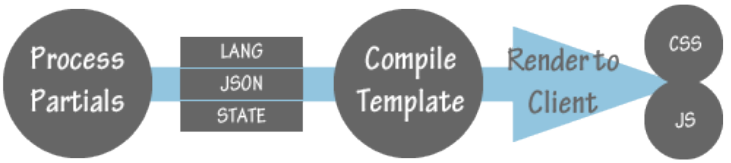
At Yahoo!, modules are isolated into individual CSS files and are bundled using a combo handler when needed. For example, when the inbox loads, it combines CSS files together for buttons, message list, sidebar, tabs, and the header. The moment the user requests the Search page, the combo handler combines search-specific styles and delivers them over the pipe. Search uses a variation of the default message list and sidebar which means it only has to load the sub-classed modules.

Pieces of the puzzle

At Yahoo!, we built a prototyping engine to help facilitate this process. Whether you need something similar will depend on the size of your project.

The prototype engine uses a mustache template[9] as the root. Mock data is stored in a JSON file, localization strings are stored in key/value pairs in a text file, and CSS and JavaScript dependencies are pulled in as needed. This allows the team to view a menu or a dialog or a form by itself or in the context of the entire site. In doing so, everybody can review functionality and design before going into engineering. We can also shift assets to engineering knowing that integration will be more seamless as a result.

The Yahoo! Prototype Engine



In the case of our prototype engine, some state management is handled before the module gets rendered. This handles conditional items, data filtering and anything else that might normally be handled via server-side processing. State management isn't always just a case of applying a class name to an HTML element.

Your Prototype

Having a full-blown engine to compile your modules could very well be unnecessary, especially for a small site. It is still advantageous to isolate your components into an easy-to-review format. MailChimp, for example, has an internal cheat sheet of design patterns[10] that they use to build the site. This documents various modules that are used throughout the site and the code required for each module.

9.http://mustache.github.com/
10.http://www.flickr.com/photos/aarronwalter/5579386649/

Remember, patterns are good. Codifying those patterns is also good. Having a process in place to review and test those patterns is great!

Preprocessors

As great as CSS is, it is still missing features that many designers and developers would like it to have. To help fill this void—and to help speed up development—tools have been created to make our lives easier.

One type of tool is the CSS preprocessor. I'll review what a preprocessor is, what it can do, and how it can help you in building scalable and modular CSS.

What is a preprocessor?

A CSS preprocessor allows you to use a special syntax in your CSS that is then compiled within your project. Some preprocessors try and stick as closely as possible to actual CSS syntax, whereas others try to simplify things as much as possible.

Take a look at this example for the Stylus[11] preprocessor.

Coding using Stylus

```
@import 'vendor'

body
  font 12px Helvetica, Arial, sans-serif

a.button
  border-radius 5px
```

11.http://learnboost.github.com/stylus/

For those who know Ruby[12] or CoffeeScript[13], this will feel familiar. It strips out the need for curly braces and semi-colons and relies on significant white space instead. The indentation indicates which properties apply to which rules. Property names never have spaces in them, which means that the first space after the property name must separate the property from the value.

In addition to Stylus, the two preprocessors that currently lead the market are Sass[14] (or similarly, Compass[15]) and Less[16].

Installing a preprocessor

One of the complaints, especially from designers not as familiar with the command line, is that these preprocessor tools can be complicated to install. Depending on your environment, they can be. Or, it can be as simple as dragging an application into your Applications folder.

For installing Sass on a Mac, for example, I just hopped to the command line and typed:

Installing Sass

```
sudo gem install sass
```

Or installing Compass

```
sudo gem install compass
```

Gem[17] is a command line tool from the Ruby world that acts as a *package manager* for installing applications and is pre-installed on recent versions of Mac OS X.

12.http://www.ruby-lang.org/en/
13.http://coffeescript.org/
14.http://sass-lang.com/
15.http://compass-style.org/
16.http://lesscss.org/
17.http://rubygems.org/

Once Sass is installed on my machine, I set Sass to watch a folder that I'm working on.

Running Sass

```
sass --watch before:after
```

Where `before` is the folder where all my `.scss` (precompiled CSS files) files are contained and `after` is where the post-compiled CSS files are placed. Any changes I make to my `.scss` files in the `before` folder are automatically compiled and placed as `.css` files in the `after` folder. This is handy for building and testing things quickly during development. (An `.scss` file is like a regular `.css` file but using the Sass syntax which I'll explain later in this chapter.)

There is also the Compass Mac app[18] that avoids the need of doing anything at the command line.

Less uses npm, the node package manager[19], which isn't pre-installed. Therefore, you'd need to install both the node package manager and then Less after that. Less also has a JavaScript version that can run client-side during the development process.

Less' Client-side Implementation

```
<link rel="stylesheet/less" type="text/css"
href="styles.less">
<script src="less.js"></script>
```

Remember not to deploy this to a live site. Always compile the CSS before launch.

18.http://compass.handlino.com/
19.https://github.com/isaacs/npm

Less Command Line Compilation

```
lessc styles.less
```

Increasingly, site development is requiring command line tools. They can be a handy way to streamline your process and even get you out of a pickle that GUI tools sometimes can't solve.

Useful features of a preprocessor

Preprocessors offer up plenty of interesting features that can help make authoring CSS easier. A few of them are:

- Variables
- Operations
- Mixins
- Nesting
- Functions
- Interpolation
- File importing
- Extending

What does all that mean? Let's look deeper at some of them. (I'll be using Sass for the examples moving forward but be aware that Less and Stylus also have similar approaches for the same concepts.)

Variables

Anybody who has worked with CSS for more than an hour has likely wished for the ability to set a colour value once in a CSS file and then to re-use that colour anywhere else in the CSS. In Sass, you can define a variable prefacing a word with `$` and assigning it a value.

Using Variables

```
$color: #369;

body {
    color: $color;
}

.callout {
    border-color: $color;
}
```

The compiler will then convert this into the final file for deployment.

Compiled Variables

```
body {
    color: #369;
}

.callout {
    border-color: #369;
}
```

This is very handy for keeping site-wide changes all in one place. (Of note, the W3C is working on a draft specification of CSS Variables[20].)

Nesting

When coding CSS, it's quite common to have a selector chain.

20.http://dev.w3.org/csswg/css-variables/

Selector Chains

```
.nav > li {
    float: left;
}

.nav > li > a {
    display: block;
}
```

Nesting allows these styles to be grouped more clearly in the precompiled CSS file.

Nesting with Sass

```
.nav {
    > li {
        float: left;
        > a {
            display: block;
        }
    }
}
```

Each set of styles is nested inside the one above it. What does this look like generated?

Generated CSS from Sass

```
.nav > li {
  float: left; }
  .nav > li > a {
    display: block; }
```

The nesting helps clarify which styles are grouped with what elements but not much different than just indenting the styles on your own. There is some saved typing from not having to type `.nav` every time.

Mixins

Mixins come with a lot of power. A mixin is a group of styles that can be re-used throughout your CSS. They can take parameters that customize the output of the mixin. One of the more common uses for mixins is to handle vendor prefixes. (Although they can really be used for any repetitive CSS rules.)

An Example Mixin for `border-radius`

```
@mixin border-radius($size) {
  -webkit-border-radius: $size;
     -moz-border-radius: $size;
          border-radius: $size;
}
```

Once you've declared a mixin, you can then call it from anywhere within your CSS using the `include` directive.

An Example Mixin for `border-radius`

```
.callout {
    @include border-radius(5px);
}
```

The preprocessor will then compile that into this:

Generated CSS for the `border-radius` mixin

```
.callout {
  -webkit-border-radius: 5px;
     -moz-border-radius: 5px;
          border-radius: 5px;
}
```

Functions

The Mixins example already looks like a function. There is, however, the ability to do some powerful stuff with calculating values. For example, the `lighten` function takes a colour value and a percentage and will adjust the lightness of that value.

Adjusting colour value using a function

```
$btnColor: #036;
.btn {
    background-color: $btnColor;
}
.btn:hover {
    background-color: lighten($btnColor, 20%);
}
```

The preprocessor will then compile that into this:

Compiled CSS with colour functions

```
.btn {
    background-color: #003366;
}
.btn:hover {
    background-color: #0066cc;
}
```

Sass comes with a number of handy functions like this and Compass adds even more. (*If you're going to use Sass, I highly recommend taking advantage of Compass, too.)*

Extensions

Extensions are the ability to extend one module with the properties of another. In Sass, this is done with the `extend` directive.

Sass Extensions

```
.btn {
    display: block;
    padding: 5px 10px;
    background-color: #003366;
}
.btn-default {
    @extend .btn;
    background-color: #0066cc;
}
```

The extension takes the styles from `btn` and applies them to `btn-default`. Sass is fairly smart, though. It doesn't simply duplicate the rules in the second declaration. It creates a combination selector for the first set of rules.

Compiled CSS from Sass Extensions

```
.btn, .btn-default {
  display: block;
  padding: 5px 10px;
  background-color: #003366; }

.btn-default {
  background-color: #0066cc; }
```

The extensions are limited to simple selectors. You couldn't extend `#main .btn`, for example. We will come back to discuss extensions and how they impact the SMACSS approach later on in this chapter.

Even more

This is only the tip of the iceberg when it comes to preprocessors. There are plenty of more features and examples on the respective websites. It may seem daunting at first. Don't feel the need to use every feature right away.

Getting into and out of trouble

You know the saying. "With great power comes great responsibility." These preprocessors offer up lots of great functionality that can keep your precompiled CSS files nice and lean. However, once compiled, the magic can result in bloated CSS that is difficult to debug. In other words, you're right back where you started. Bloated code is a possibility no matter how you code—whether by hand, by a WYSIWYG tool like Dreamweaver, or via a command line preprocessor. It's also possible to create great code using any of these tools.

Let's look at where we can get into trouble.

Deep Nesting

Once you start nesting, it can easily be taken too far. Imagine, if you will, something like this:

Deep Nesting with Sass

```
#sidebar {
  width: 300px;
  .box {
    border-radius: 10px;
    background-color: #EEE;
    h2 {
      font-size: 14px;
      font-weight: bold;
    }
    ul {
      margin:0;
      padding:0;
      a {
        display:block;
      }
    }
  }
}
```

That wouldn't be unusual to see. I've seen this quite a bit in working on existing Sass files. Here is what that generates into:

Compiled CSS using Deep Nesting with Sass

```
#sidebar {
  width: 300px; }
  #sidebar .box {
    border-radius: 10px;
    background-color: #EEE; }
    #sidebar .box h2 {
      font-size: 14px;
      font-weight: bold; }
    #sidebar .box ul {
      margin: 0;
      padding: 0; }
      #sidebar .box ul a {
        display: block; }
```

Nesting with SMACSS

The SMACSS approach, by its very nature, minimizes deep nesting due to depth of applicability. The separation of layout and modules also avoids these issues. With SMACSS, the previous example would look more like this:

Deep Nesting with SMACSS

```
#sidebar {
  width: 300px;
}

.box {
  border-radius: 10px;
  background-color: #EEE;
}

.box-header {
  font-size: 14px;
  font-weight: bold;
}

.box-body {
  margin: 0;
  padding: 0;
  a {
    display: block;
  }
}
```

There's barely any nesting at all! That's because we don't need long selectors to get the styles that we want. It's only when we need to target element selectors in a part of the module where we really need to worry about nesting.

Creating long selector chains just makes the browser work harder than it needs to when determining whether a style applies to the current element.

Unnecessary extending

Returning back to our extension example of a default button extending from default button styles, let us look at that code example again.

Sass extension of a button

```
.btn {
    display: block;
    padding: 5px 10px;
    background-color: #003366;
}
.btn-default {
    @extend .btn;
    background-color: #0066cc;
}
```

This allows us to do the following HTML, no longer needing to specify both the `btn` and `btn-default` classes on an element. Only one needs to be specified. We move the multiple declarations to the CSS from the HTML.

Application of the class on a link

```
<a class="btn-default">My button</a>
```

Extending modules to create sub-modules are a way of avoiding having to define multiple classes in the HTML. The naming convention becomes more important in this situation. Having a module name of `btn` and a sub-module name of `small` would complicate things when the only class that gets applied is `small`. A small what? With `btn-small`, we can use the sub-module name on its own and still know what its purpose is for.

Looking at the SASS source, we can also see that the `btn-default` is a sub-module because of the use of `@extend`. Looking at the generated source, we can still see that `btn-default` is a sub-module because it will be grouped with the `btn` class.

Where extending modules fails is when we extend across disparate modules.

Sass extension across modules

```
.box {
    border-radius: 5px;
    box-shadow: 0 0 3px 0 #000000;
    background-color: #003366;

}
.btn {
    @extend .box;
    background-color: #0066cc;
}
```

Extending across modules now ties these two concerns together. There is no longer one source of truth for where the styles for a module live.

By tightly grouping everything in the CSS, you lose the ability to do just-in-time loading or to do conditional compilations depending on the components you need to load into your app. Button *and* box styles need to be loaded at the same time.

Extending even within the same module can introduce complexity to a project that does just-in-time loading of styles. For example, at Yahoo!, we would load the default button styles upon page load but only load secondary styles—such as those for compose—when that screen was requested. This kept the initial page load times very quick.

SMACSS Extensions

Using the SMACSS approach, extensions are just handled at the HTML level instead of the CSS level by defining the multiple classes in the HTML.

Applying SMACSS module classes on a button

```
<a class="btn btn-default">My button</a>
```

This has the added benefit of recognizing where the root element of a module is. When looking at compiled HTML in the browser, it may be difficult to discern where one module starts and where another one ends. Since root module names don't have any hyphenation, they stand out in the crowd.

I recognize that it begins to feel like *classitis* where multiple (seemingly unnecessary) classes are being added to the HTML. However, these classes aren't unnecessary. They clarify intent and increase the semantics of the element in question.

Overused Mixins

Mixins are a handy way of avoiding repetition. However, CSS classes are also a handy way of avoiding reptition. If you have the same CSS being applied in multiple places, it may be worthwhile to move that style into its own class.

Let's say that a collection of modules all share a similar visual style of a gray background and a blue rounded border. You might decide to create a mixin for this.

Sass Mixin for common pattern

```
@mixin theme-border {
  border: 2px solid #039;
  border-radius: 10px;
  background-color: #EEE;
}

.callout {
  @include theme-border;
}

.summary {
  @include theme-border;
}
```

That gets compiled into this:

Compiled CSS from Mixin

```
.callout {
  border: 2px solid #039;
  border-radius: 10px;
  background-color: #EEE; }

.summary {
  border: 2px solid #039;
  border-radius: 10px;
  background-color: #EEE; }
```

SMACSS for Repetitive Patterns

In this case, because it is a visual treatment being added to various containers, it's worthwhile to place this into its own class.

Defining a class for common pattern

```
.theme-border {
  border: 2px solid #039;
  border-radius: 10px;
  background-color: #EEE;
}

.callout {
}

.summary {
}
```

Then apply it to elements as required:

Applying the class

```
<div class="callout theme-border"></div>
<div class="summary theme-border"></div>
```

Parameterized Mixins

To be clear, parameterized mixins offer up a lot of power that is just not possible with CSS and there is no equivalent approach to solve this beyond creating a lot of variations. The border-radius mixin example from early in this chapter is a great example of a parameterized mixin.

Smack that preprocessor

We've looked at a few common pitfalls of preprocessors in comparison to how we might accomplish it with SMACSS. The typical answer is: everything in moderation. Review the generated files and see if the final result is what you expect. If there is plenty of repetition then take a look at refactoring your approach.

Let's look at a couple more ways in which a preprocessor can encourage better modularization of your code.

State-based Media Queries with Nesting

As we saw in the chapter, Changing State, media queries are one of the ways in which we can detect and manage state changes. Most tutorials demonstrate with a separate style sheet and stuff all of the styles related to that state into a single file. This separates a module definition into possibly multiple files, making it more difficult to manage.

Sass allows media queries to be nested, allowing those state changes to be reflected where they belong: with the module.

Here is an example demonstrating nested media queries:

Nested Media Queries in Sass

```
.nav > li {
  width: 100%;

  @media screen and (min-width: 320px) {
      width: 100px;
      float: left;

  }

 @media screen and (min-width: 1200px) {
          width: 250px;
      }

}
```

The default state is defined and then the alternate states of that module are defined right from within that module. You can even embed media queries inside other media queries and Sass will concatenate the media query conditions.

Here is what the nested example generates into:

Compiled CSS from Nested Media Queries in Sass

```
.nav > li {
   width: 100%; }
@media screen and (min-width: 320px) {
  .nav > li {
    width: 100px;
    float: left; } }
  @media screen and (min-width: 1200px) {
    .nav > li {
      width: 250px; } }
```

Sass creates the separate media queries and embeds the selector inside them. With this particular example, I specifically chose the default state to be the *small screen* view that would match for any screen under 320px. Then I switched to a floated navigation once it reaches a specific width. Finally, it changes the width at 1200px but

does not re-declare the float. I like this inheritence that occurs from the default state through the various media queries.

Best of all, any alternate states for your module are declared with the module.

Organizing Your Files

Preprocessors encourage the separation of concerns that SMACSS recommends.

Here are some guidelines on how to separate the files in your project:

- Place all Base rules into their own file.
- Depending on the type of layouts you have, either place all of them into a single file or major layouts into separate files.
- Put each module into its own file.
- Depending on size of project, place sub-modules into their own file.
- Place global states into their own file.
- Place layout and module states, including media queries that affect those layouts and modules, into the module files.

With files separated in this way, it'll make your project easier to prototype. HTML templates can be created for individual components and the CSS and template for an individual component (or even a subset of components) can be tested in isolation of each other.

Preprocessor-specific components such as mixins and variables should also be specified in their own files.

A sample directory structure

```
+-layout/
| +-grid.scss
| +-alternate.scss
+-module/
| +-callout.scss
| +-bookmarks.scss
| +-btn.scss
| +-btn-compose.scss
+-base.scss
+-states.scss
+-site-settings.scss
+-mixins.scss
```

Finally, create the primary CSS file that will include the other files. For many sites, this might mean just including all of the files into the master style sheet. For projects with conditional asset loading, you can have container files that import only the necessary files for specific screens.

Inside the master file

```
@import
   "site-settings",
   "mixins",
   "base",
   "states",
   "layout/grid",
   "module/btn",
   "module/bookmarks",
   "module/callout";
```

The precompiler will compile this into a single file for development or deployment.

When you're ready to launch, create a compressed version of your CSS file for deployment. (Your environment may have build scripts for deploying the rest of the application. Be sure to integrate preprocessor compilation in that final build process.)

Command line for compressed CSS file using Sass

```
sass -t compressed master.scss master.css
```

Post mortem on preprocessors

We looked at what a preprocessor is and how to install one. We looked at some popular features and a few of the pitfalls. Finally, we looked at how preprocessors can make organizing your project easier. Preprocessors can definitely be a beneficial part of your process.

Drop the Base

There are some elements—not many, but a few—that aren't used very often. As a result, you might think (as I have) that it is safe to style them as Base Rules expecting that their purpose will be singular and never changing. As you are likely to have read by now, things change. We can plan for change and prevent future change from complicating the work we have already done.

What elements fall prey to this potential problem? The `button`, `table`, and `input` elements are the most common ones I've seen. Let's delve into an example of what can often happen on a project.

Table

Long gone from the web standardista's playbook is the use of tables for layout. As a result, the need to use a table on a project is often not needed.

Until one day, it is.

This first and only table that is needed is to display a certain set of data, such as a comparison table. The comparison table has a certain padding, column alignment, and borders. It looks great.

Table Styles

```
table {
    width: 100%;
    border: 1px solid #000;
    border-width: 1px 0;
    border-collapse: collapse;
}

td {
    border: 1px solid #666;
    border-width: 1px 0;
}

td:nth-child(2n) {
    background-color: #EEE;
}
```

A few days, weeks, or months later, there's a need to add another table in. This time, however, it serves another purpose. Headers on the left, data on the right. The borders are getting dropped and backgrounds are changing. Normally what we'd see here is an overriding of the default styles.

Overriding Previous Styles

```
table.info {
    border-width: 0;
}

td.info {
    border-width: 0;
}

td.info:nth-child(2n) {
    background-color: transparent;
}

.info > tr > th {
    text-align: left;
}

.info > tr > td {
    text-align: right;
}
```

The problem is that we've overriding styles because the base rules were designed for a singular purpose. The base rules should be designed for how we want them to appear as the default style and then augment them for specific modules. The comparison table was a module. It served a singular purpose and had a custom design, even if it was the only occurrence of the elements used in that module.

The solution is clear: make a module.

Creating a module instead

```
table {
    width: 100%;
    border-collapse: collapse;
}

.comparison {
    border: 1px solid #000;
    border-width: 1px 0;
}

.comparison > tr > td {
    border: 1px solid #666;
    border-width: 1px 0;
}

.comparison > tr > td:nth-child(2n) {
    background-color: #EEE;
}

.info > tr > th {
    text-align: left;
}

.info > tr > td {
    text-align: right;
}
```

The table element still has some base styles set. I have so rarely *not* needed a table to expand to fill its container. Nor have I *not* used `border-collapse: collapse`. These feel like they should be browser defaults!

Our comparison module now sits in isolation, as it should. I specified child selectors to keep the impact as small as possible. If I needed to embed a table within the table (which should generally be avoided) then I can be assured that the comparison module wouldn't impact the table embedded within. Our info module is now simplified to just two simple rules.

Overall, we used less CSS to achieve the result we wanted while at the same time being clearer with our code. Win-win.

As mentioned before, `button` and `input` elements can often suffer the same fate as tables. If the style serves a specific purpose then create a module. It'll avoid the need to override styles or rewrite old code.

The Icon Module

CSS sprites have become a mainstay of modern web development—and for good reason. They allow for multiple assets to be compiled into a single resource, minimizing the number of HTTP requests and ensuring that images for things like rollover states are already loaded when needed.

Before sprites came along, images would be used in two different contexts: as a background image, allowing other things to layer on top of it; or as a foreground image, residing inline with text. With sprites, everything is now a background image and positioned within the mask of the element.

It is this latter scenario that this chapter will be specifically addressing.

This will be best explained with an example, so let's look at a menu with its accompanying icons.

Menu of items with icons

Menu HTML

```
<ul class="menu">
    <li class="menu-inbox">Inbox</li>
    <li class="menu-drafts">Drafts</li>
</ul>
```

The HTML is straightforward. We have a list of menu items. A class is added to each menu item so that we can style each one differently.

Menu CSS

```
.menu li {
    background: url(/img/sprite.png) no-repeat 0 0;
    padding-left: 20px;
}

.menu .menu-inbox {
    background-position: 0 -20px;
}

.menu .menu-drafts {
    background-position: 0 -40px;
}
```

All list items are set to the single sprite and then each individual list item repositions the background to show the correct icon.

On the surface, this looks pretty good and it worked for us, for the most part. As always, we began to run into edge cases where this complicated matters.

- We became reliant on a very specific HTML structure: the list item.
- Sprites had to be redefined to be used in other modules.
- Positioning within the element was very fragile: bumping up font size could reveal other parts of the sprite.
- Handling right-to-left interfaces was more difficult since we could only use horizontal sprites and fix the x position to 0.

To resolve these issues, we moved to where the icon itself became a module: the icon module.

Restructuring the HTML to create the icon module

```
<li><i class="ico ico-16 ico-inbox"></i> Inbox</li>
```

Many people would likely balk at the use of the `i` tag. I chose to use it because it was small, mostly devoid of semantics, and was an empty element with no content. Why is there no content? In this example, the icon supports the visible text beside it. If the icon were to sit alone, then we would add a title attribute so that it could be read by a screen reader or used as a tooltip. If you disagree and feel a span is more appropriate, I'll understand.

By having a single tag with the various icon classes applied to it, it no longer has any other HTML dependencies. That's good. But why three different classes? Each plays a slightly different role and all ultimately come together to mimic a traditional `img` element.

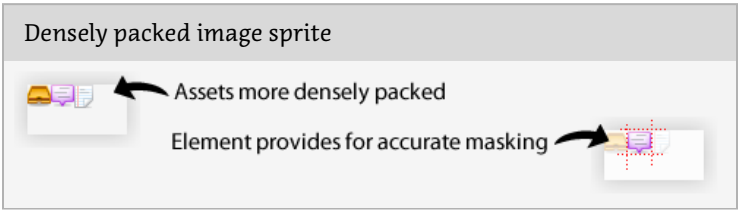
Icon Module CSS

```
.ico {
    display: inline-block;
    background: url(/img/sprite.png) no-repeat;
    line-height: 0;
    vertical-align: bottom;
}

.ico-16 {
    height: 16px;
    width: 16px;
}

.ico-inbox {
    background-position: 20px 20px;
}

.ico-drafts {
    background-position: 20px 40px;
}
```

The `ico` class defines the basics of turning the element into an inline-block element, which is what an image is. You may need to adjust `vertical-align` so that the icon sits relative to the text correctly. Internet Explorer has issues with applying `inline-block` to block elements. Since we're applying it to an inline element, it doesn't have the same issue. Alternatively, you can use `{ zoom:1; display:inline; }` for block elements in IE to behave like inline-block elements.

The `ico-16` class sets the height and width. If your project only has one size of icon, you could define the size in the `ico` class. If every icon has a different size, you can define it in the class for the specific icon. For this project, we had sets of icons in about four different sizes.

The last class, `ico-inbox`, positions the sprite to the correct coordinates. By having a fixed icon size, there's no longer a concern of the parent element growing too large and we can use the same code for right-to-left interfaces without having to modify the background position.



A more densely packed image also allows for better compression. Smaller file sizes will improve the performance of your site. (And if you're not already, I recommend using Yahoo!'s Smush.it[21] service or ImageOptim[22] for the Mac to ensure your images are as small as possible.)

We've just looked at an example of how to refactor a specific part of a project to make things more flexible. There are many ways to ap-

21.http://www.smushit.com/ysmush.it/
22.http://imageoptim.pornel.net/

proach a problem and what looks like it'll work on the surface can run into problems further into a project. Projects evolve as complexities reveal themselves and part of the fun in doing web development is deciding how best to solve these problems.

Complicated Inheritance

This chapter looks at how inheritance can sometimes cause our best laid plans to run afowl.

In this example, we will take a look at a Calendar that uses generic State rules that conflict with the inheritance within the table cells and how to possibly work around the problem.

Calendar Table

```
<table class="cal">
    <tr>
        <td>1</td> <td>2</td>
        <td>3</td> <td>4</td>
        <td>5</td> <td>6</td>
        <td>7</td>
    </tr>
    <!-- repeated 3-4 times -->
</table>
```

The calendar consists of a table with rows and columns. Each cell is a single day. The default style is is how a day cell should look under normal circumstances.

The Day Cell

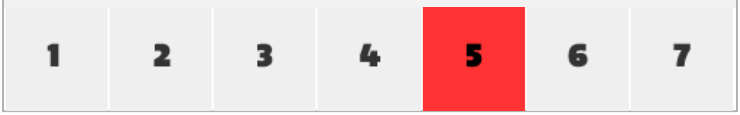
```
.cal td {
    background-color: #EFEFEF;
    color: #333;
}
```

1	2	3	4	5	6	7

Each cell in my table has a light gray background with dark gray text. Now, we want to highlight which day is *today*.

Styling Today

```
.cal td.cal-today {
    background-color: #F33;
    color: #000;
}
```



The `cal-today` class shows that the today class is part of the `cal` module. We are also increasing the specificity which will have this style override the default style. Alternative selector choices could've included just a plain `td.cal-today` which would have worked as long as it was declared after the default state rule. Had we used just `.cal-today` as our selector, we would've had to resort to using `!important` to get this style to work.

It is important to recognize that there are small decisions that have to be made like this as the project develops. In the direction I chose, I'm saying that the `.cal-today` class can only be applied on a table cell (`<td>`) and that it would be inside an element with the `cal` class (which, using SMACSS, should be considered a given).

Getting back to our example, everything looks good. Now, our calendar is a smaller view that is connected to a larger view that shows a week in full detail. Our miniature calendar needs to show which week is the currently selected week.

Selected Row

```
<tr class="is-selected">
    <td>1</td>
    <td class="cal-today">2</td>
    <td>3</td>
    ...
</tr>
```

The selected state is being used throughout the application and so it only made sense to use it here. What does the selected style look like?

Selected Row Rule

```
.is-selected {
    background-color: #FFD700; /* Yellow */
    color: #000;
}
```

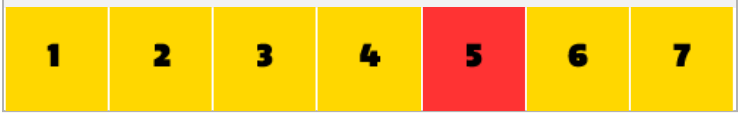
Can you see the problem? The problem is that the background colour applies to the table row while the color inheritance gets overridden by the greater specificity of both the base day style and the today style.

I could add `!important` to my state, which I have mentioned above as being an acceptable thing to do, but while it increases the specificity when applying the style to the same element, it doesn't help us here because it won't be inherited down to the cell. `!important` does not override inheritance, just specificity.

That means I need to create new rules to allow the selected state to be reflected on child elements.

Selected Row Rule for Table Cells

```
.is-selected td {
    background-color: #FFD700; /* Yellow */
    color: #000;
}
```



If this selector is defined after our calendar day selectors then everything should render just as we would expect.

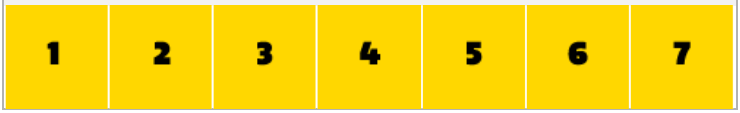
What colour will our cells be now? That depends. Was this style declared before or after the `cal` classes? If it was after, then all cells in the row will be styled correctly. Our today cell will still be coloured in red, which, on this particular project, was exactly what we wanted.

Where !important can go wrong

For argument's sake, what if we added the `!important` rule to our select cell CSS? Suddenly our today cell would no longer show today; it would be styled like the rest of the week.

What adding `!important` would do

```
.is-selected td {
    background-color: #FFD700 !important; /* Yellow
*/
    color: #000 !important;
}
```



To get the today cell styled correctly, we would have to create a new rule that combines the state rule with the module rule.

Adding extra rules to work around specificity

```
.is-selected td {
    background-color: #FFD700 !important; /* Yellow
*/
    color: #000 !important;
}

.is-selected td.cal-today {
    background-color: #F33 !important;
    color: #000 !important;
}
```

From this last example, you can see that we are having to add more selectors and more `!important` to keep things working correctly. This is definitely not ideal.

An Imperfect World

The purpose of this example was to demonstrate that inheritance can wreak some havoc on our well-laid plans and there isn't a perfect solution. SMACSS tries to mitigate many of these problems but at the end of the day, sometimes you still have to create some less-than-ideal solutions.

Minimizing the number of situations like these will still help keep your project more maintainable.

Formatting Code

Everybody has their own way. The tools and techniques that you use are ones that you have tried either through trial and error or you have tried what you heard works for other people. When I first got into development, I used Dreamweaver. It had plenty of features and allowed me to build static HTML sites quickly and efficiently. After seeing a co-worker using Ultraedit and seeing how fast he was able to get work done, I started to learn it as a way of complementing my existing tool set. The same has occurred with the way I code. I will see a technique or style that someone else uses and I will assimilate those techniques into my own way of working.

This section, Formatting Code, is a brief look at how I code my work and do so in a way that seems to work well for others having to continue working on my code.

Single line versus multiple lines

For many years, I have coded my CSS using the single line approach[23]. This means that all of the properties for a given rule set are declared on the same line. This allows for quick scanning of the selectors along the left. Being able to scan selectors has traditionally been more important to me than seeing properties nicely lined up. Up until just a couple years ago, the list of properties assigned to a rule set were quite small; it would be unusual to have more than a handful. Therefore, I could find the selector I wanted and all of the properties would be visible on the screen.

With CSS3—and the myriad of vendor-specific prefixes that come with it—things can get out of hand rather quickly. Between that

23.http://orderedlist.com/resources/html-css/single-line-css/

and working with a larger team, it was easier for everybody to have each property/value pair on its own line.

CSS3 with the plethora of vendor-prefixed properties can be too much to read easily if all on one line.

```
.module {
    display: block;
    height: 200px;
    width: 200px;
    float: left;
    position: relative;

    border: 1px solid #333;
    -moz-border-radius: 10px;
    -webkit-border-radius: 10px;
    border-radius: 10px;

    -moz-box-shadow: 10px 10px 5px #888;
    -webkit-box-shadow: 10px 10px 5px #888;
    box-shadow: 10px 10px 5px #888;

    font-size: 12px;
    text-transform: uppercase;
}
```

In the example, there are 11 properties declared and we could easily have a half-dozen more if we added another feature or two to our module. Having these all on one line would leave the first handful of properties visible on the screen and we would be left scrolling to the right to uncover the rest of the properties. This makes it hard to scan the document and see what properties have been defined. Having everything on a single line can also make it harder to do diff comparisons in version control.

There are a few other minor things to note with the example:

- There is a space after the colon.
- Four spaces before each declaration (no tabs).
- Properties are grouped by type.
- Opening bracket on the same line as the rule set.
- Colour declarations use the short form.

These are all preferential and I will not begrudge you for using a completely different approach. This is just what I have found that feels natural and makes the most sense to me.

Grouping Properties

Some people organize alphabetically, others don't organize at all, and others may use some other arbitrary system. I fall in this last category. It's not completely arbitrary, mind you. I would describe it as ordering from most important to least important but what is important when it comes to declaring styles on an element?

I organize in the following order:

1. Box
2. Border
3. Background
4. Text
5. Other

Box includes any property that affects the display and position of the box such as `display`, `float`, `position`, `left`, `top`, `height`, `width` and so on. These are most important to me because they affect the flow of the rest of the document.

Border includes `border`, the oft-unused `border-image`, and `border-radius`.

Background comes next. With the advent of CSS3 gradients, background declarations can actually become quite verbose. Once again, vendor prefixes just compound the issue.

Multiple backgrounds with CSS3 declarations. Code example from Lea Verou's CSS3 Pattern Gallery[24].

```
background-color: #6d695c;
background-image: url("/img/argyle.png");
background-image:
  repeating-linear-gradient(-30deg,
rgba(255,255,255,.1), rgba(255,255,255,.1) 1px,
transparent 1px, transparent 60px),
  repeating-linear-gradient(30deg,
rgba(255,255,255,.1), rgba(255,255,255,.1) 1px,
transparent 1px, transparent 60px),
  linear-gradient(30deg, rgba(0,0,0,.1) 25%,
transparent 25%, transparent 75%, rgba(0,0,0,.1)
75%, rgba(0,0,0,.1)),
  linear-gradient(-30deg, rgba(0,0,0,.1) 25%,
transparent 25%, transparent 75%, rgba(0,0,0,.1)
75%, rgba(0,0,0,.1));
background-size: 70px 120px;
```

Complex patterns are possible with CSS3 gradients but create for lengthy background declarations, and the example doesn't even include CSS3 prefixes. Just imagine how long this declaration would be if it did!

Text declarations include `font-family`, `font-size`, `text-transform`, `letter-spacing` and any other CSS properties that affect the display of the type.

Anything that doesn't fall into any of the above categories gets added to the end.

Colour Declarations

This may seem like a silly thing to even mention but I have seen different developers do different things. Some like using keywords like `black` and `white` but I have always tried to stick to either the 3 digit or 6 digit hex format wherever possible. `#000` and `#FFF` are

24.http://leaverou.me/css3patterns/

shorter, albeit barely, then their keyword counterparts. Likewise, I wouldn't use `rgb` or `hsl`, since the hex form is shorter. Of course, `rgba` and `hsla` have no hex form and they would get used.

Be Consistent

At the end of the day, the important part—like much of what SMACSS describes—is to establish a standard, document it, and be consistent with it. It will make it easier on you and others as your project evolves.

Resources

There are plenty of great tools and other resources out there. Some of them are directly related to the concepts discussed here and others are just useful tools to have in your repertoire.

SMACSS Resources

Since the initial release of the book, SMACSS-related resources have started to pop up.

- Middleman SMACSS[25]
- SMACSS for Drupal[26]
- SCSS Toolkit[27]. A starter toolkit based on SMACSS.
- Kickstart SMACSS[28]

CSS Preprocessors

- LESS[29]
- Sass[30]

25.https://github.com/nsteiner/middleman-smacss
26.http://drupal.org/sandbox/johnalbin/1704664
27.https://github.com/davidrapson/scss-toolkit
28.https://github.com/Anderson-Juhasc/kickstart-smacss
29.http://lesscss.org/
30.http://sass-lang.com/

Component-based Frameworks/Methodologies

- Object-Oriented CSS (OOCSS)[31]
    - OOCSS for JavaScript Pirates Slides[32]
    - MailChimp UI Library based on OOCSS[33]
- BEM[34]

Other Frameworks

- HTML5 Boilerplate[35]
- normalize.css[36]
- Bootstrap[37]
- 960.gs[38]
- Eric Meyer CSS Reset[39]

Documentation

- Front-end Style Guides[40]
- Knyle Style Sheets[41]

31.http://oocss.org/
32.http://speakerrate.com/talks/4642-oocss-for-javascript-pirates
33.http://www.flickr.com/photos/aarronwalter/5579386649/
34.http://bem.github.com/bem-method/html/all.en.html
35.http://html5boilerplate.com/
36.https://github.com/necolas/normalize.css/
37.http://twitter.github.com/bootstrap/
38.http://960.gs/
39.http://meyerweb.com/eric/tools/css/reset/
40.http://24ways.org/2011/front-end-style-guides
41.http://warpspire.com/posts/kss/

Other Resources

- mustache[42] is a logic-less template language. It's the format we chose at Yahoo!.
- Pattern Primer[43] is a PHP script that lets you preview your snippets of HTML all on one page.
- Terrifically[44] is a JavaScript/jQuery framework for working with OOCSS.

42.http://mustache.github.com/
43.https://github.com/adactio/Pattern-Primer
44.http://www.terrifically.org/