Generator-Constant-Axiom-Class (GCAC) CSS Methodology

By Joseph Juma

My approach to CSS has recently become more structured, after an associate told me about `utility classes` as a CSS approach. For those unfamiliar, a utility class is a new name for an old approach. You create CSS rules named things like `.dark-background` or `.title-text` and then compose your style by adding these various classes inline to the HTML element you want to style. So a specific element like look like `<div class="dark-background title-text 16px-border text-underlined">...</div>` to style it with those various traits. This can be a very beneficial approach for two reasons.

1. It allows you to not need to repeatedly style CSS, so you can include all the styling you need for a specific look in one class. This makes your CSS much more modular, easier to use without complex rule interactions, and simpler to style given elements. Taking this approach can even heavily reduce the sheer size of the CSS you have to write.
2. It puts your CSS styling in the class list with a modular structure, which is accessible via JavaScript. By exposing such fine-grained control to JS, you open up a lot of possibilities for intelligent styling based on logical operations.

I had last seen this approach before the common adoption of css preprocessor, and so it hadn't crossed my mind until my friend had shown me his usage of it. At the same time, I had been brushing up on my Swiss Style design knowledge, and writing my own CSS library for a complex application I was creating. This was enough reason to sit down and seriously consider my approach, from which I developed a structure I believe is sensible and powerful, if a little heavy for creating strong CSS.

Structure

My CSS approach has 4 parts.

Generators are CSS functions created using a CSS preprocessor to take in a variable and create a class based on the input value. A good example Generator is one which takes in a font-size and creates a properly set typography styling. Being a student of

Swiss Designer and an advocate for the golden ratio, I created a series of text generators in LESS code as follows.

```
.golden-font-size(@size)
{
    /*
        Creates a font where the text takes up 61.81% of the line-height, and
        38.19% is taken up by white space forming a golden ratio visually.
    */
    font-size:@size;
    line-height:1.61819; // unitless makes this into a coefficient multiplied by font-size.
}

.title-font-size(@base)
{
    /*
        Creates a size of font that can fit 1 line of title in the same space
        ( not counting the padding ) as 2 lines of subtitle, and 3 lines of
        normal font.
    */
    .golden-text(2.61819*1.61819*@base);
}

.subtitle-font-size(@base)
{
    /*
        Creates a size of font that can fit 2 lines of subtitle in the same
        space ( not counting the second-line's whitespace ) as 3 lines of
        standard body font.
    */
    .golden-text(1.61819*@base);
}

.standard-font-size(@base)
{
    .golden-text(@base);
}
```

as you can see, this allows me to do complex mathematical equations with precomputed coefficients to create designs based on an input value. Generators should be relatively clean of referring to any variables that aren’t strictly part of the underlying generator’s behavior. This means if a variable, even if it is a global constant, must be referred to, then it should be passed in as a parameter rather than just included. The reason for this will become clear later.

Constants are constant values that are used to configure the overall look of the application. These can be things like predefined colors, font-sizes, type families or

sizing values. You create constants in a constants file and then refer to them in other code files.

Axioms are my name for `utility classes`. These are simple CSS rules that tackle one specific piece of styling. You should use both constants and generators, often together, to create the rules for axioms. By being diligent with your generators, you can make sure that even if you need to change constants, your axioms will still functional well overall. Axioms should be relatively atomic and simple. The reason to call these axioms rather than utility classes, is because the word `utility` has too many other possible uses, while the word axiom does accurately describe these CSS rules.

Classes are just standard CSS classes, but they are constructed by combining together various axioms. This allows you to refer to a single css class in your HTML rather than stringing together a messy and long set of axioms or utility class names. This may seem strange given my previous lauding of the granular control over class-style exposed to JavaScript by inline classes, but many utility classes often create messy lists of overly long classes. For most behaviors it is better to write distinct classes which represent the distinct states of a type of element, and use JavaScript to toggle between these rather than performing array operations on the classList object. However this is not a *hard rule* and there are times when appending a class to the classList to achieve a one-off styling is more effective than creating an entirely separate class just for that permutation. For example, if you have an axiom for highlighting the text in paragraphs, tables and divs on mouse-over, and are styling a div with a specific class, say `.fancy-div` you should not need to make an additional `.fancy-highlighted-div` class just to apply highlighting, but instead can just apply the `.highlight` class to the div component. Conversely, say you had 10 divs that had a number of axioms for the background, text and bordering. If you wanted a button press to change 5 of these divs to have different background, text and border styling, that would be 3 styles that all happen in conjunction. In this case, if those styles will always vary together, then that should be it’s own specific class.

Conclusion

With this technique you can create cleaner modular CSS that has complex behaviors and can be reconfigured by primarily changing the axioms and the classes. Although somewhat heavy in comparison to the `utility classes` approach it can allow for cleaner, more sophisticated CSS stylings.