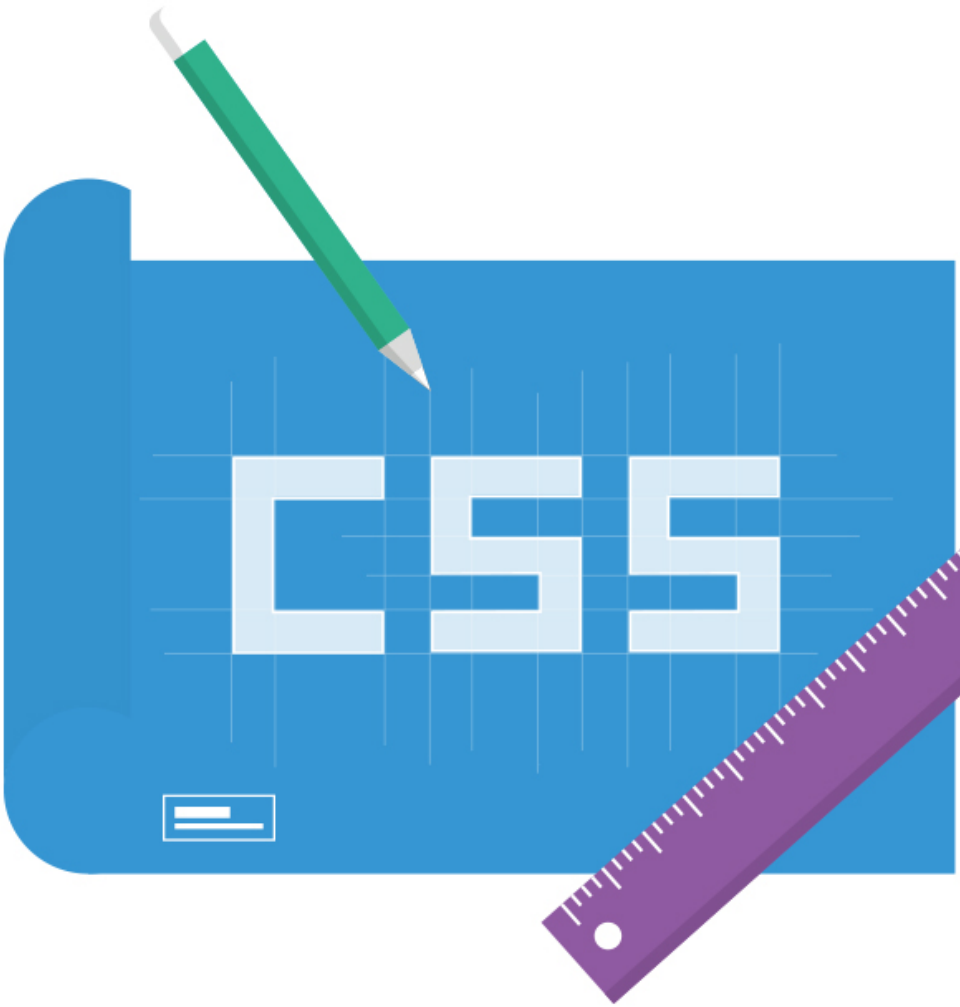
CSS Eficiente

Técnicas e ferramentas que fazem a diferença nos seus estilos



Casa do Código

TÁRCIO ZEMEL

Casa do Código
Livros para o programador
Rua Vergueiro, 3185 - 8º andar
04101-300 – Vila Mariana – São Paulo – SP – Brasil

Sumário

Versão: 18.5.25

Capítulo 1

Introdução

Ah, as CSSs! As queridas folhas de estilo que tanto amamos e, às vezes, odiamos , que estão conosco, os desenvolvedores *front-end*, na maioria dos dias de nosso trabalho!

Desde que nosso colega norueguês **Håkon Wium Lie** brindou o mundo com sua invenção inteligentemente chamada de *Cascading Style Sheets* (CSS), muita coisa já aconteceu com essas danadinhas!

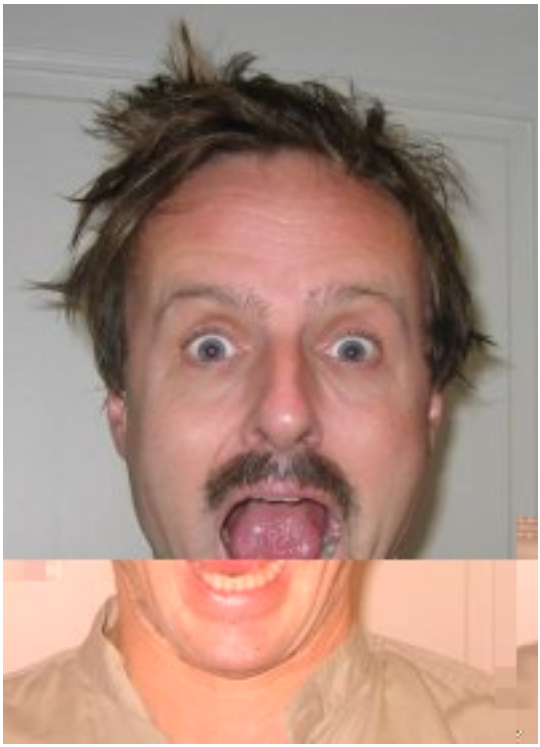


Fig. 1.1: Håkon Wium Lie tentando convencer o W3C a aprovar as CSS

Atualmente, é possível alcançar com a CSS o que, até não muito tempo atrás, não se passava de sonho ou do uso de muitos códigos JavaScript. Devido às suas modernas características recentes, elas se popularizaram e ganharam os corações de muitos desenvolvedores e entusiastas por todo o mundo!

As folhas de estilo não são tão complicadas de se aprender. Ao contrário, são populares justamente pela sua facilidade e rápido início, aliado à visualização imediata do que se está fazendo! Em suma: a CSS é famosa!

Entretanto, como qualquer coisa em nossas vidas, também existe o lado ruim. Com o passar do tempo, essa facilidade de uso trouxe um lado negativo ao ecossistema de estilização para web: a adoção involuntária de más práticas de uso. Não que isso tenha sido algo maquiavélico, uma conspiração de meia dúzia de *webdevs* maléficos rumo à disseminação de práticas ruins. Sim-

plesmente aconteceu. Por conta disso e por ninguém ter avisado que não era certo, ela simplesmente propagou-se e continuou assim.

Com a palavra: quem entende do assunto

Tenho a felicidade de ter contato com alguns dos grandes nomes do cenário mundial quando o assunto é CSS. Certamente, o que pessoas como **Harry Roberts** e **Jonathan Snook** pensam sobre boas práticas conta bastante!

Para enfatizar o quão importante é adotar as boas práticas que estão presentes neste livro, perguntei a estas duas sumidades: *Por que uma Arquitetura CSS é importante? E por que precisamos de uma?*

Harry Roberts, do **CSS Wizardry** (http://goo.gl/sXt93d) , criador da metodologia ITCSS (9) que chega ao Brasil com exclusividade por meio deste livro, respondeu:

CSS é incrivelmente fácil de escrever incrivelmente fácil , mas possui uma ordem de magnitude mais difícil para se gerir, manter e escalar. Como os projetos ficam mais maduros e maiores, CSS pode realmente começar a "comer pelas beiradas", causando quantidades incalculáveis de confusão, dores de cabeça e frustrações para os desenvolvedores responsáveis por geri-los.

Isso se dá, principalmente, devido a uma série de princípios fundamentais de CSS que tornam inerentemente mais difícil a gestão de qualquer projeto de UI não trivial: é impossível encapsular totalmente regras, então todas elas têm a capacidade de herdar ou repassar estilização de e para outras; nada é imutável, logo, é possível acidentalmente desfazer ou interferir com outras regras no projeto; não existe fluxo de controle ou lógica, o que significa que não há nenhuma pista sobre a forma ou o estado do projeto; sua sintaxe liberal (loose syntax) torna muito fácil que as pessoas façam a coisa errada; há total dependência da ordem do código, o que significa que, se porções de código são movidas (ou adicionadas em partes erradas do projeto), um monte de outras coisas podem (potencialmente) quebrar; a especificidade trabalha contra alguns dos outros princípios de design do core da linguagem; e o fato de ser tão simples significa que todo mundo que mexe com CSS, muitas vezes, tem sua opinião de como as coisas devem ser feitas, o que leva a colisões e confusão.

Quando todos esses problemas se reúnem em grande escala, cada um é ampliado e se torna ainda mais evidente. Uma arquitetura CSS nos fornece uma

forma estruturada de escrita e gerenciamento de CSS de uma forma que tanto contorna ou doma essas esquisitices, quanto dá aos desenvolvedores um conjunto de regras e diretrizes a seguir a fim de continuar a escrever CSS que é amigável, são, simples de compreender e mais fácil de escalar.

Jonathan Snook, criador do SMACSS (4), disse:

Sistemas grandes, complexos, em crescimento, precisam de organização. Todos estes precisam de alguma maneira de dar sentido à complexidade. Programação orientada a objetos existe há décadas como uma maneira de estabelecer a ordem em meio ao caos. Desenvolvimento server-side tem sido por anos a ascensão popular de frameworks MVC. JavaScript tem sido a ascensão de ferramentas para gerenciamento de dependência.

Somente nos últimos anos, temos visto qualquer tentativa de organizar CSS de uma forma sensata. Abordagens como OOCSS e SMACSS surgiram para proporcionar entendimento mais facilitado de projetos, permitindo qualquer um a, rapidamente, "pular dentro" e ser mais produtivo, a ser capaz de fazer isso sem a sensação de que tudo vai desmoronar quando alguma mudança é feita.

Estas são palavras de eminentes desenvolvedores front-end do planeta Terra. Se elas não são o suficiente para fazer você chegar à conclusão de que uma Arquitetura CSS mais robusta é **imprescindível** para projetos front-end profissionais, então não há mais nada no mundo que fará.

Conclusão

Este livro tem o objetivo de ajudar, pelo menos um pouquinho, a disseminar algumas boas práticas e tecnologias interessantíssimas para se escrever e manutenir projetos CSS para você que já trabalha (ou trabalhou) com as folhas de estilo e quer aprimorar seu trabalho, passando a escrever **CSS e□ciente**.

Portanto, nele você **não vai** encontrar explicações básico-teóricas como: o que é e como é formada uma regra CSS; regras de usuário, de navegador e de autor; comparativo entre CSS inline, incorporado e externo; qual a diferença entre IDs e classes; o que são pseudoclasses e pseudoelementos; quais as propriedades CSS existentes e valores possíveis para cada etc. Na verdade, muitas dessas informações nem serão úteis aqui. Logo, façamos um combi-

nado: ficará subentendido que você é uma pessoa que já estudou tudo isso e agora aprimorará suas habilidades.

Imagine que estamos cuspindo em nossas respectivas mãos e as apertando num chacoalhar amistoso para selar esse pacto.

Aproveite a leitura!

Capítulo 2

Seletores CSS

A maioria dos desenvolvedores front-end, principalmente os que dão menor importância à teoria, cria suas próprias regras CSS como se não houvesse amanhã. Embora isso seja verdade em relação a alguns prazos estabelecidos pela doce e carismática figura do gerente de projetos, as folhas de estilo são escritas sem planejamento, sem preparação, sem consciência e sem muitos outros atributos essenciais para que um trabalho bem feito possa ser executado.

Se você se enquadra nessa infeliz descrição, este capítulo é importantíssimo, já que vai abordar temas como especificidade CSS, explicação de como os seletores CSS são interpretados e outras dicas gerais. O intuito é a disseminação da prática de planejar a escrita de regras CSS antes de escrever código às cegas. Ou seja, pensar antes de fazer.

Além da vantagem óbvia de adquirir mais conhecimentos e passar a escrever CSSs melhores, você também terá condições de argumentar com o seu

gerente de projetos sobre como é importante planejar suas folhas de estilo para obter um CSS mais eficiente e profissional.

E se, ainda assim, não adiantar, peça a ele que me envie um e-mail para ler uma ou duas verdades!

2.1 Especificidade CSS

É simples escrever regras CSS: primeiro escreva o elemento, classe ou ID que se quer estilizar (aí são inclusos seletores avançados) e aplique as propriedades cabíveis. Porém, já aconteceu de você criar uma regra e, por algum motivo, ela não "entrou em ação"? Quer dizer, você a criou esperando que tudo corresse bem, mas, por algum motivo, ela não foi aplicada e outra regra teve precedência?

Provavelmente isso acontece com todos que trabalham com CSS. É devido à **especi□cidade de CSS** que isso ocorre. Um dos motivos de haver "Cascata" em "Folha de Estilo em Cascata" é referente, justamente, a quão específica determinada regra é, para ser "mais importante" que outras e entrar em ação em detrimento às demais.

O que é a especi□cidade CSS?

Não entender a teoria e a prática da especificidade CSS, raramente, não leva a confusões, desgostos e falsas acusações de bugs "Já criei a regra, mas ela não funciona!" , já havendo relatos de aumento de pressão e náuseas. Entretanto, não é algo tão difícil de se entender.

Pense na especificidade CSS como um **sistema de pesos** que serve para determinar qual regra CSS tem precedência quando várias podem ser aplicadas ao mesmo elemento. Regras com maior peso têm preferência sobre regras de menor peso e por isso "ganharão" e entrarão em ação quando o navegador renderiza os estilos.

Toda regra CSS tem sua especificidade implícita; portanto, mesmo não sabendo, você *já* se vale da especificidade! A ideia é que, a partir de agora, você faça isso conscientemente.

Entendendo a especi□cidade CSS

Existem 4 categorias que definem o nível de especificidade a um dado seletor. Preste atenção na ordem, pois é importante. Para facilitar as explicações, vamos atribuir letras a cada uma:

a) Estilos inline;

b) IDs;

c) Classes, pseudoclasses e atributos;

d) Elementos e pseudoelementos.

Por exemplo, seria possível visualizar a especificidade de um elemento `h1` da seguinte maneira:

```
h1 = (a=0), (b=0), (c=0), (d=1)
```

Quer dizer, no seletor `h1` existe 0 estilos inline; 0 IDs; 0 classes, pseudoclasses e atributos; 1 elemento ou pseudoelemento.

Para ficar menos verboso, isso poderia ser mostrado como:

```
h1 = 0,0,0,1
```

Para facilitar ainda mais, também é possível "transformar" isso em um número inteiro:

```
h1 = 0001
```

Se o seletor fosse `#content h1`, teríamos:

```
#content h1 = 0,1,0,1
```

Ou:

```
#content h1 = 0101
```

Isso significa que, se ambas as regras existissem em um conjunto de estilos de uma página, `#content h1` teria precedência em relação a `h1`, dado que **0101 > 0001**.

Atente-se ao fato de que a matemática envolvida é somente para demonstrar a lógica que acontece "nos bastidores". Seria completamente incoerente se, por padrão, CSS priorizasse a estilização de um elemento puro em detrimento à estilização deste mesmo elemento dentro de um contêiner (neste caso, um ID), que é um seletor mais específico.

Mais um:

```
#sidebar ul li a.myclass:hover = 0,1,2,3
```

0 estilos inline; 1 ID; 2 classes, pseudoclasses ou atributos; 3 elementos ou pseudoelementos.

> Pense rápido: qual o peso do seletor universal `*`? **0**! Ele não tem estilos inline, IDs, classes, pseudoclasses ou atributos, nem elementos ou pseudoelementos. Simples assim!

Para ajudar a fixação do conceito, tente responder: dado um conjunto de regras CSS, entre os seletores `#content article ul` e `#content .list`, ambos servindo perfeitamente para estilizar listas dentro de um contêiner, qual dos dois teria mais peso e entraria em ação?

Fazendo os cálculos:

```
#content article ul = 0,1,0,2

#content .list = 0,1,1,0
```

No seletor `#content article ul` existe 0 estilos inline; 1 ID; 0 classes, pseudoclasses e atributos; 2 elemento ou pseudoelemento. Já no `#content .list`, 0 estilos inline; 1 ID; 1 classe, pseudoclasses ou atributo; 0 elemento ou pseudoelemento.

Como **0110 > 0102**, a regra com o seletor `#content .list` teria mais peso, além de também ser um seletor CSS com melhor performance, como se verá a seguir.

Para treinar bastante, invente alguns seletores sem pensar no peso e depois tente chegar ao seu cálculo. Para conferir se você acertou, use a Calculadora de especificidade CSS: http://goo.gl/ekN1Go.

Casos especiais na especi□cidade CSS

Existem alguns casos especiais na especificidade CSS:

- Empate;
- `!important`.

No caso de acontecer um empate, ou seja, duas ou mais regras em um mesmo conjunto de folhas de estilo com o mesmo peso, a cascata CSS entra em ação.

Ao usar `!important`, o valor da respectiva propriedade ganha prioridade sobre quaisquer efeitos da especificidade. Havendo conflito entre vários `!important`, novamente entra em ação o efeito cascata, com precedência das CSS de usuário às de autor.

> `!important` foi originalmente criado para ser usado somente como uma maneira de sobrepor regras de folhas de estilo de autor. Infelizmente, seu uso se espalhou da maneira errada e, atualmente, ele é usado indiscriminadamente: por preguiça e/ou desconhecimento da especificidade CSS.
>
> Agora que você já conhece sobre a especificidade, **comece a usar `!important` somente para o □m a que se destina!**

Como explicado anteriormente, *toda* regra CSS tem sua especificidade implícita; ou seja, você já estava valendo-se disso mesmo sem saber. A ideia é que, a partir de agora, você faça uso dos benefícios da especificidade CSS conscientemente. Ao utilizá-la, você escreverá melhores seletores e escapará de situações em que, sem este conhecimento, não se compreende bem por que "tal" ou "qual" propriedade não está sendo apresentada, mesmo tendo sido declarada.

2.2 Como seletores CSS são interpretados

No dia a dia de trabalho com CSS, infelizmente, não é tão comum o desenvolvedor prestar atenção em como está escrevendo seus seletores CSS. A imensa maioria não recorda que a forma como os seletores são escritos influencia a performance de seu CSS, afetando diretamente a qualidade do código e como (de que maneira) essas regras são avaliadas pelo interpretador. É mais ou menos como acontecia com a especificidade CSS antes de você ter lido o tópico anterior.

Vale sempre lembrar que pelo menos 80% das questões de performance de web sites são referentes a front-end (http://goo.gl/mNb045) . Portanto, é interessante e, em alguns casos, vital para o sucesso de um site, que os seletores CSS sejam escritos e otimizados de modo a garantir o melhor desempenho possível!

É verdade que as dicas apresentadas a seguir aplicam-se, principalmente, a sites de alto desempenho, nos quais a velocidade é uma característica imprescindível e mil elementos podem estar presentes no DOM. Porém, não importa se você está desenvolvendo o próximo Facebook ou um site para o decorador local: as melhores práticas são as melhores práticas!

Seletores CSS não são novidades para você. Os seletores mais básicos são de tipo (exemplo: `div`), ID (exemplo: `#header`) e classe (exemplo: `.tweet`), respectivamente. Os mais incomuns incluem pseudoclasses comuns (exemplo: `:hover`) e seletores CSS3 mais complexos, incluindo `:first-child` ou `[class^="grid-"]`.

Seletores têm uma eficiência inerente que, partindo dos mais eficientes para os menos eficientes, são:

- ID (`#header`);
- Classe (`.promo`);
- Tipo (`div`);
- Irmão adjacente (`h2 + p`);
- Filho (`li > ul`);

- Descendente (`ul a`);
- Universal (`*`);
- Atributo (`[type="text"]`);
- Pseudo-classe/Pseudo-elemento (`a:hover`).

É importante notar que, apesar de, tecnicamente, um ID ser mais rápido e mais performático, esta diferença é mínima, até desprezível (como será mostrado no decorrer do livro).

Combinando seletores CSS

É possível ter seletores independentes, como `#nav`, que vão selecionar qualquer elemento de ID "nav", ou você pode combinar seletores, como `#nav a`, que vão corresponder a qualquer link/ âncora dentro de qualquer elemento de ID de "nav" (no caso, seria somente 1 por página, por tratar-se de ID).

Pode parecer estranho e um pouco difícil de se entender rapidamente, mas, diferentemente do que a maioria dos ocidentais, que costumam ler da esquerda para a direita, **seletores CSS são lidos da direita para a esquerda** pelos navegadores. Para tentar compreender o motivo pelo qual os browsers fazem isso, lembremo-nos daqueles jogos de "encontre o caminho" que brincávamos nas revistas da nossa infância. Todos nós iniciávamos a resolução pelo objetivo final do emaranhado de trajetórias, retrocedendo até um dos pontos iniciais possíveis.

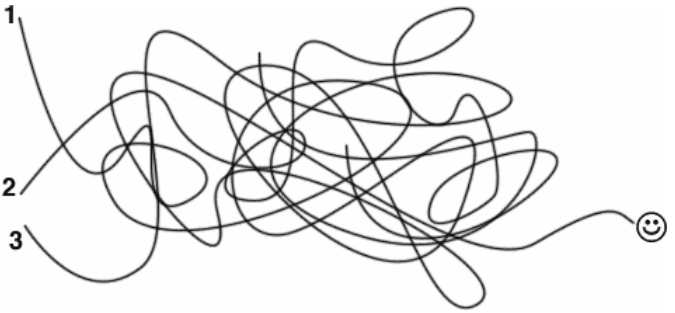


Fig. 2.1: Você já brincou com isso, admita!

É mais eficiente para um navegador começar sua procura por combinações a partir do elemento mais à direita (o que ele sabe que receberá o estilo) e trabalhar o seu caminho de volta através da árvore de DOM, do que começar no alto dessa árvore e percorrer o caminho para baixo, pois poderia nem mesmo acabar no seletor que precisa receber a estilização (também conhecido como **seletor-chave**).

Já deu pra ter uma noção, mas, caso queira uma explicação mais detalhada, confira o tópico *CSS Selectors parsed right to left. Why?* no Stack Overflow (http://goo.gl/IXCgic) .

O seletor-chave

O seletor-chave, como discutido, é a parte mais à direita de um seletor CSS. Ele é o que o interpretador do browser procura em primeiro lugar. Lembre-se da ordem dos seletores mais eficientes que abordamos anteriormente. Seja qual for o seletor-chave da vez, ao escrever CSS eficiente, é este seletor que contém o segredo do alto desempenho!

Um seletor como:

```
#content .intro { }
```

O navegador procurará todas as instâncias de `.intro` e, depois, começar a subir no DOM até encontrar o elemento `#content`.

No entanto, o seguinte seletor não é muito performático:

```
#content * { }
```

O que ele faz é olhar para cada elemento da página (para cada um, mesmo!) e, em seguida, checar se qualquer um deles é descendente de `#content`. Este é um **seletor com performance ruim** e, como seletor-chave, é muito "expensivo". Usando este conhecimento, é possível tomar melhores decisões quanto ao modo de classificar e selecionar elementos.

Vamos dizer que, em uma página realmente grande de um site realmente enorme, há centenas ou, mesmo, milhares de elementos `a`. Há, também, uma pequena seção de links de mídia social em um `ul` com um ID "social", com 3 links, digamos que: Twitter, Facebook e Google+, além de incontáveis outras âncoras.

Portanto, seria excessivamente custoso e nada performático um seletor como:

```
#social a { }
```

O que vai acontecer aqui é que o navegador vai avaliar todos os milhares de links na página antes de fazer o *match* com os de `#social`. O seletor-chave corresponde a muitos outros elementos que nada têm a ver com a estilização pretendida.

Para remediar a situação, uma solução possível seria adicionar um seletor mais específico e explícito, `.social-link`, para cada um dos `a` da área social. Porém, isso iria contra o fato de que não é aconselhado colocar classes desnecessárias na marcação quando é possível usar uma solução mais enxuta.

É por isso que o **desenvolvimento web focado em desempenho** é algo tão interessante. Um estranho equilíbrio entre melhores práticas de padrões web e performance!

Considere o seguinte código HTML:

```
<ul id="social">
    <li><a href="#" class="twitter">Twitter</a></li>
    <li><a href="#" class="facebook">Facebook</a></li>
    <li><a href="#" class="gplus">Google+</a></li>
</ul>
```

Com o seguinte CSS:

```
#social a { }
```

Agora, veja este outro HTML:

```
#social .social-link { }
```

Ou, dependendo da marcação presente:

```
.social-link { }
```

Esse novo seletor-chave vai corresponder a poucos elementos, o que significa que o navegador conseguirá interpretar a estilização mais rapidamente e passar para a próxima coisa a ser feita.

Lembre-se de que seletores-chave determinam quanto trabalho o navegador terá de fazer. Preste bastante atenção neles!

Superquali□cando seletores

Você já viu o que é um seletor-chave e que é nele que está a maior parte do trabalho que o interpretador de CSS do navegador tem de fazer. A boa notícia é que é possível trabalhar para otimizá-lo ainda mais! A melhor coisa sobre ter bons seletores-chave explícitos e criar CSS eficiente e de qualidade é que é possível evitar a "superqualificação" de seletores.

Um seletor superqualificado se parece com:

```
html body .wrapper #content a { }
```

O navegador tem que olhar para todos os elementos `a`, verificar se está em um elemento de ID *content* respectivamente e assim por diante, até chegar em `html`. Isso está fazendo com que ele passe por caminhos que, realmente, não precise, sendo pelo menos 3 desses elementos no seletor totalmente desnecessários.

```
#content a { }
```

Um outro exemplo, infelizmente, bastante comum:

```
ul#nav li a { }
```

Já se sabe que, se o `a` está dentro de `li`, que tem que estar dentro de `#nav`, de modo que é possível, eliminar o `li` do seletor logo de cara. Como o `#nav` é um ID (e já sabemos que só podem existir IDs únicos e exclusivamente nominados na página), o elemento `ul` também pode ser eliminado.

Uma forma melhor (mas ainda não ideal) de se escrever, seria:

```
#nav a { }
```

> Seletores superqualificados fazem com que o navegador trabalhe mais do que precisa e deveria. Construa seletores mais enxutos e de alto desempenho, eliminando as partes desnecessárias.

Toda essa otimização de seletores é realmente necessária?

A resposta curta é: provavelmente, não.

A resposta mais longa é que depende do site que está sendo desenvolvido. Se você está trabalhando no próximo projeto de seu portfólio pessoal, dê preferência a escrever um código limpo em detrimento à performance de CSS, porque, neste caso, não fará tanta diferença. Agora, se você está construindo a próxima Amazon, microssegundos de velocidades na página farão a diferença!

Entretanto, independentemente do tamanho do seu projeto, os navegadores interpretarão de forma igual e terão o mesmo trabalho de análise das regras CSS. Se estiver escrevendo algo como:

```
div:nth-of-type(3) ul:last-child li:nth-of-type(odd) * {
    font-weight: bold;
}
```

Você, muito provavelmente, está fazendo a coisa errada!

Porém, se seu humilde projeto web começa a expandir e a ter cada vez mais acessos, crescendo exponencialmente acima do esperado de uma hora para a outra, os recursos despendidos para otimizar o CSS serão o melhor investimento a ser feito? Será que não haverá arrependimentos por não ter escrito um bom CSS desde o começo se isso acontecer? Fica a questão.

2.3 Não use IDs como seletores CSS

Quando se está aprendendo a trabalhar com CSS, é comum que se estude logo no início as diferenças entre IDs e classes; quais vantagens e desvantagens cada um apresenta; como e quando usar um ou outro e assim por diante. Mas, após a realização de alguns testes de performance, pode ser que esse aprendizado original precise de uma pequena revisão.

Quando se é desenvolvedor web, fazer com que códigos e/ou marcações sejam cada vez melhores, independente de em qual linguagem estiverem, é praticamente inerente ao ofício. Com isso em mente, algumas considerações interessantes podem ser encontradas, principalmente em relação ao uso de

seletores de ID em CSS, após se fazer alguns testes com performance em folhas de estilo.

Porém, existem alguns pontos que merecem ser levados em consideração:

- Performance;
- Fragmentação;
- Tradição.

Performance

Pergunte para qualquer codificador CSS e ele responderá que os seletores de ID são os mais rápidos. Entretanto, isso vem com uma grande limitação: IDs são os seletores mais rápidos somente se forem os seletores-chave.

Para um seletor `#home a`, chega a ser até normal supor que o browser vai localizar o elemento de ID *home* e aplicar os valores das propriedades em todos seus links. Rapidíssimo, não é? Agora, veja o que realmente acontece: ele pega todos os elementos `a` e verifica se são descendentes imediatos de `#home`. Caso não seja, vai checando os ascendentes até chegar a `<html>`.

Será que a diferença de performance é tão grande mesmo? Para ter certeza, foram feitos 3 testes usando o **Criador de Testes CSS** de Steve Souders (http://goo.gl/DwfTuC) . Cada um deles utilizou 1000 elementos que foram selecionados a partir de IDs individuais, classes ou IDs com os seletores `a`. As páginas foram recarregadas diversas vezes para pegar a média dos resultados (números em ms).

#id…	.class…	#id… a
94	93	115
91	92	101
90	91	100
90	91	100
88	92	106
89	90	102
90.3	**91.5**	**104**

Fig. 2.2: Média de resultados dos testes (ms)

Para as 1000 regras testadas, os IDs mostraram-se cerca de 1 milésimo de segundo mais rápido do que as classes como um seletor-chave. Em outras palavras, **a diferença de desempenho entre IDs e classes é irrelevante!**

O cenário mais comum de se usar um ID como *namespace* de um elemento (sendo este o seletor-chave) é realmente mais lento do que usando uma classe. Mas, novamente, somente com 13 milissegundos de diferença. Isso também é válido para os poucos bytes extras para adicionar uma classe a um elemento que já tenha um ID, por exemplo, mudando `<div id="search">` para `<div id="search" class="search">`.

Porém, isso não quer dizer que todos os seletores sejam iguais. O seletor universal `*` e alguns CSS3 têm desempenho relativamente fraco quando usados como seletores-chave.

%	seletor CSS
~100%	.class, .class:not(a), .class[a], #id, .class:nth-child(odd), element, no style
~870%	[class^="a"], [title="a"]
~1020%	[a], [a="b"]

Fig. 2.3: Desempenho de seletores: porcentagem por tipo de seletor

No entanto, usando seletores CSS, não se torna um problema se for para 1 a 2 seletores (elementos e/ou classes). O impacto na sua performance é pequeno se comparado a outras coisas, como otimização de imagens ou redução de requisições HTTP. Esse tipo de coisa é o que realmente importa quando se tem a **performance do site** como fator preponderante.

Fragmentação

Além de serem "anzóis de estilo", IDs são utilizados como identificadores de fragmentos específicos da página (um `href` que termina em `#ancora` leva ao elemento de ID *âncora*) e para uso de JavaScript. Se já existem IDs na página por outras razões, por que não reutilizá-los para estilizar? O problema é que isso torna o código fragmentado: há dependências entre CSS e JavaScript e/ou um identificador de fragmento. Usando classes, é possível decidir mudar para um (novo) esquema de nomeação a qualquer momento e tudo com que é preciso se preocupar é alterar alguns nomes no CSS e HTML.

Existem muito poucos casos em que estilos baseados em ID nunca serão reutilizados. Um exemplo é usar `#search` para a caixa de busca do site. Pode-se decidir mais tarde que uma nova caixa de busca será adicionada em outro local (abaixo dos resultados de busca, por exemplo).

Mesmo ele sendo único nesse projeto, talvez você queira um pouco de CSS rápido para copiar e colar em um projeto futuro, que também pode ter uma caixa de pesquisa no rodapé da página. Ainda que para um grupo complexo de folhas de estilo, é possível dividir esses padrões em vários trechos reutilizáveis de CSS. Usar classes em vez de IDs impede que esses problemas potenciais venham à tona.

Ao contrário de IDs, **não há nenhuma restrição em usar várias classes**, tanto em mais de uma vez na página, quanto várias delas em um mesmo elemento. Navegadores ainda estilizarão IDs, mesmo se, por engano, houver duplicação em uma página. Mas aventurar-se fora do HTML válido o torna propenso a problemas.

Tradição

Estilizar com IDs e classes são uma das primeiras coisas que aprendemos sobre CSS. Por isso, pode ser um desafio ler que não se deve usar IDs em seletores CSS. O ramo do desenvolvimento web está sempre mudando. Ocasionalmente, precisamos reavaliar o que funciona ou não, em vez de apenas ficar com o que está nos servido bem, sem questionamentos.

No momento, estamos passando por um renascimento maciço com HTML5, CSS3, desempenho de JavaScript e outros eventos. Logo, é um bom momento para esse repensar. É importante examinar as razões de forma racional e decidir se as questões apresentadas são realmente válidas, em vez de rejeitá-las de antemão.

Conclusão

Como pôde ser constatado, parece que não há razões convincentes para se usar IDs em seletores CSS (a não ser para *namespaces*, uso de *widgets* e *plugins* específicos em CMSs e casos similares). Classes podem fazer tudo o que IDs fazem! Pense nisso na próxima vez que iniciar um projeto e tente trabalhar com classes, deixando os IDs para identificadores de fragmentos ou *hooks* para JavaScript.

Agora que já vimos mais sobre especificidade e algumas boas práticas, veremos no próximo capítulo como dar o próximo passo na eficiência de seu CSS conhecendo CSS orientado a objetos.

P.S.: aquela conversa de pedir para seu gerente de projetos me enviar e-mails foi brincadeira. É sério. Não faça isso!

Capítulo 3

CSS orientado a objetos

Apesar de ser possível encontrar alguns desenvolvedores web que praticam as duas artes milenares principais da profissão (front-end e back-end), o mais comum é que se escolha uma dessas áreas para investir e seguir com os estudos. Mesmo que você não seja versado nos místicos conhecimentos do back-end, pelo menos já deve ter ouvido falar em **Orientação a Objetos**.

Tentando abstrair e simplificar o máximo possível, o objetivo do paradigma é **evitar repetição de código**. Tenho total desapreço à palavra 'paradigma', mas pesquisas mostram que livros que a têm, em pelo menos uma ocorrência, contam com vendagem 12,73% superior.

No final do dia e, por "dia", leia-se: décadas de esforços combinados por desenvolvedores do mundo inteiro , a Orientação a Objetos está aí para ajudá-los a não repetir código e, consequentemente, facilitar sua expansão e manutenção, por meio de seus inúmeros métodos e técnicas.

Em 2009, devido à comprovada eficiência do paradigma, a desenvolvedora **Nicole Sullivan** (ex-Yahoo!) teve a ideia de trazer o site para o front-end para que o objetivo principal também pudesse ser almejado em CSS. Assim surgiu o **CSS orientado a objetos**. Na verdade, ela chamou de *Object Oriented CSS* (OOCSS); isso é uma tradução.

Dê uma boa olhada no códigos CSS de seus últimos projetos e veja que há **muita** repetição desnecessária de propriedades. Quantas declarações para definir fontes, espaçamentos e posicionamentos repetidas, linha por linha, existem? Calma, não precisa contar realmente; é apenas para enfatizar que é comum de acontecer. Isso acaba gerando problemas de médio prazo que atrapalham o andamento de qualquer projeto que se preze.

A boa notícia é que existe uma cura: **CSS orientado a objetos**.

> Para deixar bem claro, não se trata de mudar a sintaxe do CSS ou instalar algum pacote mágico que o fará melhor. CSS orientado a objetos é uma metáfora para indicar que é possível escrever um CSS mais eficiente, sem repetições, que enseje a projetos mais profissionais. No caso do que é mostrado neste capítulo, é considerado por alguns até como *framework*.

CSS orientado a objetos visa resolver alguns problemas clássicos de CSS, tais como:

- A dificuldade de tocar projetos de médio/grande porte; é preciso ser um expdert para isso;
- O tamanho dos arquivos CSS é cada vez maior conforme o projeto evolui;
- Reúso de código quase inexistente (pessoas não confiam em código alheio);
- Código frágil (até o melhor código pode se perder quando um não expert mexe nele).

Se você analisar estes itens, facilmente constatará que o problema que os origina é, além da falta de padronização de código, não planejar para escrevê-los! Ou seja, não há mais reflexão a respeito da aplicação da tecnologia CSS, ou, se preferir, o não amadurecimento profissional da maioria dos que a aplicam.

Segundo a **Dona Sullivan** (http://goo.gl/VUV2xA) , as soluções que emergem do uso de OOCSS são inúmeras (e tentadoras), tornando seu CSS:

- **Modular**: combinável, reusável e extensível;
- **Leve**: relacionamento 1:N entre CSS e potenciais layouts;
- **Rápido**: poucas requisições HTTP e tamanhos mínimos de arquivos;
- **Pronto para o futuro**: manutenível, semântico e padronizado;
- **Simpli□cado e acessível**: um bombonzinho!

Se, na sua opinião, essas são características desejáveis a um código CSS, continue lendo.

3.1 Princípios básicos de CSS orientado a objetos

Dentro da proposta de OOCSS, existem dois princípios mais importantes:

- Separar estrutura e skin;
- Separar contêiner e conteúdo.

Separar estrutura e skin significa repetir características visuais como "skins" separadas, que podem ser combinadas em vários "objetos" para conseguir-se uma extensa gama de variações visuais sem muito código. Por exemplo, backgrounds e estilos de borda.

A diretriz também pode significar o uso de classes para nomear objetos e componentes, em vez de confiar somente na semântica HTML. Por exemplo, um objeto de mídia com `class="media"` e seus componentes com

`class="img"` (para componentes de imagem e vídeo) e `class="bd"` (para componentes de texto). Ao referenciar essas classes nas folhas de estilo, o HTML ganha uma flexibilidade maior; ou seja, se um novo elemento de mídia surgir nos próximos anos (como `<svg>`), ele pode receber a estilização sem que seja preciso mexer em 1 linha de CSS!

Separar contêiner e conteúdo essencialmente quer dizer: raramente use estilos que dependam de localização. Idealmente, um objeto deve parecer-se igual, independentemente de onde estiver na página, ou mesmo se trocar de página.

Em vez de estilizar um título secundário específico com `.myObject h2 { }`, crie e aplique uma classe que **descreva** o elemento em questão, como `<h2 class="category-title">`. Isso garante que:

- Todos os `<h2>` sem a classe não sejam afetados inadvertidamente;
- Todos os `<h2>` com a classe tenham o mesmo estilo;
- Não é preciso criar estilos extras para os casos em que seja preciso que um `.myObject` se pareça com um `<h2>` não estilizado.

O que é "objeto" em OOCSS?

Continuando com a metáfora de Orientação a Objetos (OO), um objeto em CSS é análogo a uma instância de uma classe Java ou PHP, por exemplo.

Um objeto CSS é formado por 4 elementos:

- HTML, que pode ser 1 ou mais nós do DOM;
- Declarações CSS, que estilizam estes nós, começando com o nome da classe referente ao *wrapper*;
- Componentes como imagens de background e *sprites*, por exemplo;
- Comportamentos JavaScript, *listeners* ou métodos associados.

Isso pode ser confuso porque cada classe CSS não é um objeto necessariamente, mas pode ser uma propriedade de uma classe *wrapper*. Por exemplo:

```
<div class="mod">
    <div class="inner">
        <div class="hd">Bloco Head</div>
        <div class="bd">Bloco Body</div>
        <div class="ft">Bloco Foot</div>
    </div>
</div>
```

O objeto é um módulo, indicado pela classe `mod`. Ele contém 4 nós-propriedades (que não podem existir sem o módulo), incluindo 2 regiões obrigatórias (`inner` e `bd`) e 2 regiões opcionais (`hd` e `ft`).

OOCSS e performance

Valer-se de OOCSS também traz o benefício da performance, que é duplo:

- **Intenso reúso de código**: o que significa menos código CSS, arquivos menores e transferências mais rápidas;
- **Menos repaints e re□ows**: se não sabe o que isso significa, leia o artigo: *O que todo desenvolvedor front-end deve saber sobre renderização de páginas web* (http://goo.gl/9rCo4B) .

3.2 Exemplos de objetos

Existem diversos tipos de "objetos" em OOCSS. Veja exemplos de alguns deles a seguir para fixar melhor os conceitos e aliar teoria à prática.

Media

O objeto `Media` permite que você tenha uma imagem (ou flash) ou outro tipo de mídia com tamanho fixo à esquerda ou direita com algum conteúdo que a descreva no centro.

Classes:

- `.media`: wrapper do objeto Media;
- `.img`: nó-filho do objeto Media, sendo geralmente um link ou imagem, aparecendo à esquerda do nó `.bd` (opcionalmente consta no objeto);

- `.bd`: área principal de conteúdo do objeto Media, pode conter quaisquer outros objetos (obrigatório constar no objeto);
- `.imgExt`: nó-filho do objeto Media, sendo geralmente um link ou imagem, aparecendo à direita do nó `.bd` (opcionalmente consta no objeto).

HTML:

```
<div class="media">
    <a href="http://twitter.com/stubbornella" class="img">
        <img src="http://a3.twimg.com/image.jpg"
             alt="Stubbornella">
    </a>

    <div class="bd">
        <a href="http://twitter.com/stubbornella">
            @Stubbornella
        </a>
        <span class="detail">14 minutos atr&aacute;s</span>
    </div>
</div>
```

Data Table

O objeto `Data Table` se destina a formatar uma tabela corretamente para determinados tipos de dados.

Classes:

- `.data`: wrapper do `Data Table`;
- `.txtL`: alinha à esquerda, sendo o padrão (aplicável em `table`, `tr`, `td` ou `th`);
- `.txtR`: alinha à direita (aplicável em `table`, `tr`, `td` ou `th`);
- `.txtC`: alinha ao centro horizontal (aplicável em `table`, `tr`, `td` ou `th`);
- `.txtT`: alinha ao topo (aplicável em `table`, `tr`, `td` ou `th`);

- `.txtB`: alinha ao fundo (aplicável em `table`, `tr`, `td` ou `th`);
- `.txtM`: alinha ao centro vertical (aplicável em `table`, `tr`, `td` ou `th`).

HTML:

```
<div class="data">
    <table class="txtC"><!-- alinhamento da tabela ao centro -->
        <tr class="odd txtL">
            <th scope="row" class="txtR">
                Alinhado &agrave; direita
            </th>
            <td>Alinhado &agrave; esquerda</td>
        </tr>

        <tr class="even">
            <th scope="row">Alinhado ao centro</th>
            <td>Alinhado ao centro</td>
        </tr>
    </table>
</div>
```

Existem bem mais objetos, mas, como informado anteriormente, estes exemplos foram somente para se ver algo aplicado na prática. No repositório oficial (http://goo.gl/6ZqGKo) , é possível encontrar tudo e mais um pouco sobre OOCSS. Contudo, para conseguir mexer com segurança, você precisa conhecer um pouco sobre pré-processadores CSS, assunto de um capítulo vindouro.

3.3 Conclusão

A Sra. Sullivan na verdade, não sei se ela é casada, mas mantenho o Sra. em sinal de respeito gentilmente disponibilizou um wiki no GitHub (http://goo.gl/rBwNjN) . O repositório contém um projeto bem ferramentado para brincar com OOCSS, contando com uma configuração de VM com Vagrant, comandos `make` e outras características bonitas de se ver, e um domínio com exemplos práticos dos conceitos de OOCSS para facilitar a disseminação de suas ideias (http://goo.gl/cCFAhH) . Acessando essas referências, você pode

encontrar mais informações a respeito de CSS orientado a objetos e continuar seus estudos.

A ideia de trazer conceitos de Orientação a Objetos para CSS foi louvável, mas, na prática, a proposta apresentada por Nicole é um tanto engessada, dado que é preciso conhecer o que está disponível e trabalhar com classes e estruturas predefinidas. Não seria interessante se tudo isso fosse um pouco mais flexível e, como plus, tivesse algum tipo de aprimoramento de arquitetura?

OOCSS inspirou muitos desenvolvedores a realizar trabalhos melhores e, até, a desenvolver novas soluções inspiradas/baseadas na ideia de desacoplamento e reúso intenso de CSS. É o caso de **SMACSS**, que você verá no próximo capítulo.

Capítulo 4

SMACSS

Jonathan Snook é um desenvolvedor canadense que, assim como você, percebeu que a maneira como o CSS é escrito e gerenciado poderia ser melhor. Evidentemente, o fato de ter trabalhado durante anos em um sem-número de projetos inclusive para grandes empresas, como Yahoo!, mesma empresa que Sullivan, do CSS orientado a objetos, do capítulo 3) o ajudou a constatar isso. Ante tamanha indignação e com intuito de auxiliar os colegas desenvolvedores, como também a si próprio, decidiu normatizar determinadas regras para que isso fosse possível. Em 2011, 2 anos depois do surgimento de OOCSS, surgia a **SMACSS**.

SMACSS (pronuncia-se “smacks”) significa)

projetos, mas que, como você verá logo a seguir, pode ser usado em projetos de qualquer porte.

Nas palavras do próprio Jonathan, SMACSS está mais para um guia do que para um framework, como alguns costumam considerar. Não se trata de nenhum arquivo que precise ser baixado ou ser instalado. Trata-se, na verdade, de tentar documentar uma abordagem prática, útil e consistente ao escrever CSS.

Por esse mesmo motivo, não precisa ser seguido à risca necessariamente: pode-se, depois de aprender SMACSS, aplicar o que se julga mais conveniente para determinado projeto ou incorporar somente as partes que fazem sentido **para você** ao seu próprio *know-how* de desenvolvimento.

Este capítulo não tem a pretensão de substituir o conhecimento que se encontra no livro oficial, que pode ser adquirido em https://smacss.com/ por um precinho muito camarada. Ao contrário, pretende apresentar SMACSS e passar por suas principais diretrizes para que você possa conhecer mais a respeito. Ao constatar que é um método que realmente vale a pena ser aprendido mais a fundo, você pode optar por adquirir o livro e saber de mais detalhes e nuances a respeito.

4.1 Introdução a SMACSS

Um dos primeiros conhecimentos que se adquire em SMACSS é a respeito da estrutura e organização de seus arquivos. Mesmo em projetos de médio porte, raramente é possível encontrar todas as regras CSS em um único arquivo de desenvolvimento! Em SMACSS, existem 5 categorias básicas de organização de código CSS:

- Base;
- Layout;
- *Module* (Módulo);
- *State* (Estado);
- *Theme* (Tema);

Regras **Base** são as regras padrão. São quase que exclusivamente seletores de elementos, mas também pode haver seletores de atributos, pseudoclasses ou seletores mais avançados, como seletores-irmãos.

Regras **Layout** dividem a página em seções, podendo conter 1 ou mais módulos.

Módulo contém as regras principais de um projeto, as que dão o "volume" de CSS. Elas são, como indica o próprio nome, as partes modulares e, consequentemente, as mais usadas.

Regras **Estado** descrevem como determinado layout ou módulo se comporta em determinada condição ou "estado" (ativo ou inativo; oculto ou visível). Também descrevem como um módulo pode ser diferente dependendo da página em que está; por exemplo, ser apresentado de um jeito na página inicial e de outro em uma interna.

As regras **Tema** são similares às **Estado**, haja vista que descrevem como layouts ou módulos devem se parecer em determinadas situações. A maioria dos sites não precisa de temas, mas é bom estar preparado para quando for o caso.

Não é necessário preocupar-se em decorar as categorias; você as fixará com o tempo e prática de SMACSS. Preocupe-se, neste momento, em saber que a ideia é que o CSS seja organizado em vários arquivos, separados por "função", com regras/ linhas-guia específicas que se aplicam a cada "tipo" e sua razão de existir.

Com essa organização, torna-se possível evitar repetições de regras e o código fica muito mais organizado e manutenível, além de também facilitar bastante o período de adaptação e entendimento de projeto de novos participantes que também conheçam SMACSS. Criam-se **padrões**, certamente algo de que todo projeto que almeje contar com o status "profissional" precisa.

4.2 Nomenclatura

Valer-se de determinadas regras de nomenclatura é conveniente ao se separar as regras nestas 5 categorias, porque ajuda a identificar, de imediato, a qual delas certa folha de estilo pertence e seu papel no escopo geral do projeto.

Convencionou-se usar um prefixo para diferenciar entre Layout, Módulo

e Estado:

- Usar `l-` (ou `layout-`) para Layouts;
- Usar `is-` para Estados (exemplo: `is-hidden`, `is-collapsed`);
- Não usar prefixo `m-` ou `module-` para Módulos; por serem as folhas de estilo em maior número em projetos, torna-se desnecessário.

Veja alguns exemplos:

```
/* Módulo "Example" */
.example { }

/* Módulo "Callout" */
.callout { }

/* Módulo "Callout" com Estado */
.callout.is-collapsed { }

/* Módulo de campo de formulário */
.field { }

/* Layout "Inline" */
.l-inline { }
```

4.3 Base

Regras **Base** são aplicadas a elementos usando seletor de elemento, de seletor-descendente ou de seletor-filho, além de pseudoclasses. Não estão inclusos seletores de ID e classes.

Servem para definir como é a aparência padrão de determinados elementos todas as vezes em que estes aparecem nas páginas do projeto. Incluem especificações de tamanhos de títulos, aparência padrão de links, estilos de fontes, backgrounds gerais etc. Geralmente, não há necessidade de se usar `!important` em estilos Base.

Exemplo:

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

input[type="text"] {
    border: 1px solid #999;
}

a {
    color: #039;
}

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

Também servem de exemplo aqueles *CSS reset*, com regras que "zeram" determinadas propriedades para tentar uniformizar a apresentação entre navegadores. Embora eu considere a abordagem de usar um "padronizador", como `normalize.css` (http://goo.gl/TxmiBv) , mais eficiente.

Veja outro exemplo de uma folha de estilo Base, extraído de um projeto real meu:

```
*,
*:before,
*:after { box-sizing: border-box; }

/* @see http://stackoverflow.com/a/2610741/922143 */

::-webkit-input-placeholder { /* WebKit browsers */
    color: #333;
}

:-moz-placeholder { /* Mozilla Firefox 4 to 18 */
    color: #333;
    opacity: 1;
}
```

```
::-moz-placeholder { /* Mozilla Firefox 19+ */
    color: #333;
    opacity: 1;
}

:-ms-input-placeholder { /* Internet Explorer 10+ */
    color: #333;
}

body {
    font-family: Arial, sans-serif;
    overflow-x: hidden;
}

h1,
h2,
h3,
h4,
h5,
h6 {
    letter-spacing: 1px;
}

figure {
    margin: 0;
}

input,
select,
textarea {
    font-size: 1rem;
}
```

4.4 Layout

Layout pode ser entendido de muitas maneiras diferentes, dependendo de seu contexto e da área de atuação em que a palavra é usada. No caso de SMACSS,

é preciso diferenciar entre os componentes maiores e menores que formam uma página, sendo aqueles as regras Layout e, estes, Módulos, que serão vistos logo em seguida.

Para facilitar o entendimento, relembre da explicação dada anteriormente: layouts podem conter 1 ou mais módulos. Eles também podem ser divididos entre "maiores" e "menores" (bem menos comuns), baseando-se na sua frequência de reúso nas páginas de um projeto.

Apesar daquele princípio de que não se deve usar IDs como seletores CSS (2.3), em SMACSS não é tão incomum seu uso para designar layouts maiores, como headers, footers etc. Contudo, não há nenhum tipo de problema em continuar não usando IDs como seletores CSS.

Exemplo de regras de Layout:

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

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

Geralmente, seletores de layout são únicos (razão da leniência ao usar IDs), mas há casos em que ele precisa responder a alguns fatores. Por exemplo, um site pode apresentar layouts diferentes baseado na preferência de quem o está acessando. Nesse caso, essa regra continuaria sendo uma regra Layout em SMACSS, mas usada em combinação com outro estilo de layout.

Veja:

```
article {
    float: left;
}

#sidebar {
```

```
    float: right;
}

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

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

No exemplo, `.l-flipped` seria aplicado a um elemento em nível acima, como `body`, o que permitiria que o posicionamento dos principais componentes fosse alternado.

Ou, em um exemplo mais condizente às (irrefreáveis e irreversíveis) tendências de web design responsivo:

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

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

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

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

4.5 Módulo

Como já citado brevemente, regras **Módulo** são o cerne de sua aplicação, comumente o que mais se trabalha em projetos que utilizam SMACSS. São com-

ponentes mais específicos, como barras de navegação, widgets e assim por diante.

Módulos se encontrarão dentro de layouts quase sempre. Também é possível que eles sejam usados dentro de outros módulos, o que, na verdade, é até comum de acontecer. Porém, tenha sempre em mente que, ao ser criado, seu objetivo é conseguir funcionar como um componente isolado (*standalone*).

Idealmente, deve ser possível usar quantos módulos forem necessários em uma página, ou até mesmo trocar um de lugar ou, mesmo, de uma página para outra sem que seja preciso realizar nenhum ajuste. Em função disto, para eles é **estritamente proibido** usar IDs e/ou seletores de elementos!

Só tome cuidado para não incorrer no erro de fazer algo como:

```
<div class="module">
    <span>Tipo de Informa&ccedil;&atilde;o</span>
</div>

.module > h2 {
    padding: 5px;
}

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

Em um primeiro momento, ele atende aos requisitos de um módulo de ser possível alterá-lo de lugar ou página e ser independente, não sendo afetado por estilos condicionais baseados em sua localização. Porém, além de essa solução apresentar um seletor ineficiente, pode ser necessário o acréscimo de outras informações que, não necessariamente, são apresentadas da mesma maneira, como:

```
<div class="module">
    <span>Tipo de informa&ccedil;&atilde;o</span>
    <span>Outro tipo de Informa&ccedil;&atilde;o</span>
</div>

[...]
```

O segundo `<span>` e quaisquer outros que ali forem colocados sem a intenção de receber essa mesma estilização serão afetados pelas mesmas regras que o primeiro. Se não for a intenção, será preciso "remendar" a situação com outras regras ou com o infame `!important`. Portanto, atenção a esse tipo de "tentação" que, apesar de ser mais fácil de planejar e escrever, pode reservar surpresas desagradáveis para um futuro próximo!

Se você estiver se perguntando: "Mas como resolver isso?", não se preocupe, pois antes de terminar este livro você já terá visto bastante coisa sobre nomenclatura e semântica em CSS.

4.6 Estado

Em SMACSS, **Estado** é algo que serve para incrementar/ sobrescrever outros estilos, geralmente (e por convenção) com classes prefixadas com `is-`. Por exemplo, um menu ou seção com a aparência *accordion* pode estar no estado aberto ou fechado; uma mensagem do sistema pode estar em um estado de sucesso ou erro.

Geralmente, estados são aplicados ao mesmo elemento como uma regra de layout ou módulo. Por exemplo:

```
<div id="header" class="is-collapsed">
    <form>
        <div class="msg is-error">Mensagem de erro!</div>

        <label for="searchbox" class="is-hidden">Buscar</label>
        <input type="search" id="searchbox">
    </form>
</div>
```

Outro ponto importante ao qual ficar atento é que geralmente estados possuem vínculo com JavaScript (JS). Afinal, como se trata de um elemento que possui estados diferentes, é preciso alternar entre estes de alguma maneira quando algum *trigger* é acionado. Na atual tecnologia web disponível, isso é feito via JS.

Esta é mais uma convenção que ajuda bastante: (quase) sempre que vir um prefixo `is-` sendo usado, automaticamente você já pode identificar que se trata de estado com dependência de código JavaScript.

Porém, também é possível se valer do atributo `data-` para trabalhar com estados, dado que seletores de atributo também podem desempenhar bem esse papel. Entretanto, mesmo usando-os, o JS ainda é necessário para realizar a alternância. Veja:

```
.btn[data-state="default"] {
    color: #333;
}

.btn[data-state="pressed"] {
    color: #000;
}

.btn[data-state="disabled"] {
    opacity: .5;
    pointer-events: none;
}
```

```
<button class="btn" data-state="disabled">Desabilitado</button>
```

Como observação final, tratando-se de estados, na perspectiva de SMACSS não há restrições em se usar `!important`, embora, para CSS, você deva evitar sempre que possível.

Exceções e □exibilidade com regras de Estado

Haverá casos em que um estado será bastante específico e terá relação com somente um módulo em particular.

Para esses casos em que uma regra de estado é feita especialmente para algum módulo, o nome da sua classe deve ter o mesmo nome dele. Também, a regra pode estar no mesmo arquivo em que ele está, não havendo necessidade de ficar separado junto com os outros arquivos de estado.

Por exemplo:

```
.tab {
    background-color: #800080;
    color: #fff;
}
```

```
.is-tab-active {
    background-color: #fff;
    color: #000;
}
```

4.7 Tema

O significado de **Tema** em SMACSS é o mesmo que em qualquer outro site que não a utiliza: um conjunto diferenciado de cores, imagens e elementos de UI que caracterizam o *look & feel* das páginas de um site.

Quanto à elaboraç

```
.mod {
    border: 1px solid #000;
}

/* theme.css */

.mod {
    border-color: #00f;
}
```

Sim, é tão simples quanto se valer das próprias características de cascata de CSS! Tendo um arquivo de tema pronto, para que este sobrescreva as regras de uma regra base, layout, módulo ou estado, basta carregar seu arquivo dinamicamente (via JavaScript) ou não.

Também existe outra possibilidade, caso se demande muita personalização: regras específicas para o tema. Uma vantagem adicional para essa possibilidade é conseguir um ajuste fino de aparência, ao colocar as classes somente nos base, layout, módulo ou estado que se queira.

Um exemplo prático:

```
/* theme.css */

.theme-border {
    border-color: #800080;
}

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

Atente ao fato de que, caso opte por usar classes específicas de tema, existe a convenção de se usar o prefixo `theme-`.

4.8 Mais exemplos da prática de SMACSS

Como já citado anteriormente, este capítulo nunca teve a pretensão de substituir o conhecimento que se encontra no livro oficial de SMACSS. A parte

teórica a respeito já está de bom tamanho como proposta deste livro. Porém, para ajudar na fixação do que foi explicado, alguns exemplos práticos podem ser de bastante ajuda.

Estrututa de diretórios

Em projetos do mundo real, não importando qual seja a estrutura do projeto que abriga os estilos, geralmente cada uma das categorias de SMACSS fica separada por um diretório. Algo como:

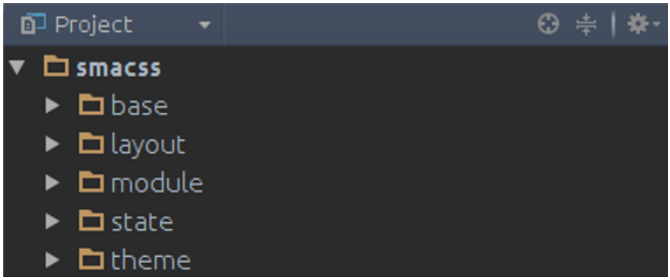


Fig. 4.1: Estrutura de diretórios SMACSS

Não há quaisquer restrições em haver outros diretórios no mesmo nível hierárquico dos que constituem as categorias de SMACSS.

Como foi visto, nem todos os projetos precisam de *theme* e, para os casos em que estados são feitos especificamente para um módulo, podem estar no mesmo arquivo que ele. Como SMACSS não é rígida, nada impede de o diretório *state* também ser dispensável, caso essa seja sua preferência. O que nos traz para:

Fig. 4.2: Estrutura de diretórios SMACSS sem state e theme

Vale lembrar novamente que SMACSS é sobre convenções não rígidas, então faça como **você** e sua equipe acharem melhor. É possível até não quererem usar *state* e *theme* em um primeiro momento, mas deixar os diretórios presentes na estrutura caso se decidam ou precisem se valer deles.

Se você estiver usando Git para controlar as versões do projeto, basta adicionar um arquivo `.gitkeep` em cada um dos diretórios para que eles sejam considerados no repositório mesmo não tendo outros arquivos.

Arquivos na estrutura SMACSS

Dentro de cada diretório, portanto, constam os respectivos arquivos referentes àquela categoria SMACSS. Em um projeto fictício, poderíamos ter algo como:

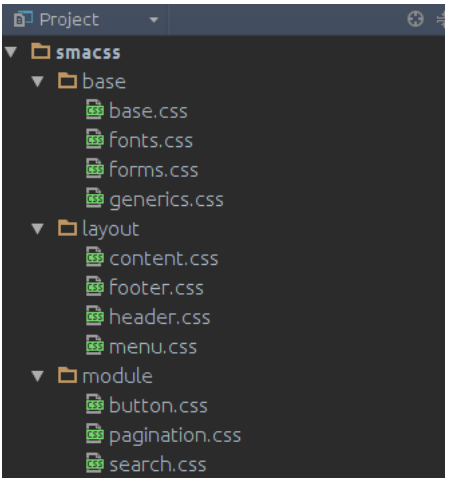


Fig. 4.3: Estrutura de diretórios SMACSS com exemplos de arquivos

Neste projeto-exemplo, no diretório "base" a função dos arquivos, além dos óbvios `fonts.css` e `forms.css`, seriam: `base.css` com regras mais gerais referentes a elementos, tais como estilização de links, tamanhos de títulos, parágrafos e listas, entrelinhas etc.; e `generics.css` para classes genéricas, que podem ser usadas em qualquer ponto do site para surtir determinado efeito visual que se queira. Um trecho desse `generics.css` poderia ser:

```
[...]

.bold {
    font-weight: bold !important;
}

.italic {
    font-style: italic !important;
}

.uppercase {
    text-transform: uppercase !important;
}

.font-size-extra {
    font-size: 150% !important;
}

[...]
```

Note a presença de `!important` em cada uma dessas regras. Neste exemplo, o motivo da existência dessas classes é que elas serão usadas em momentos em que suas respectivas estilizações se fizerem necessárias, sem que ocorram repetições desnecessárias. É bem mais rápido, prático e performático definir uma regra somente uma vez, usando uma classe que faça seus efeitos visíveis, do que repetir propriedades e valores em várias diferentes.

Como são genéricas, pode ser que o elemento em que sejam aplicadas já tenha recebido alguma diretriz em contrário. Por exemplo, se determinado elemento foi estilizado com `text-transform: capitalize;`, dependendo do restante dos estilos pode não adiantar inserir a classe `uppercase`. Logo, esse `!important` garante que a regra genérica tente prioridade e, de antemão, entre em ação sem que seja preciso mexer em mais estilos.

Saiba quando quebrar regras

Foi dito anteriormente que, em estilos Base, geralmente não há necessidade de se usar `!important`. Este poderia ser um caso em que se optaria por usar. Não há nenhum problema em quebrar regras, desde que seja algo coerente e justificado, tal como o foi.

Quanto aos arquivos no diretório `layout`, geralmente são somente definições das estruturas principais das páginas do site. Mesmo que não se tenha quaisquer estilizações referentes a algum layout, muitos adeptos a SMACSS criam os respectivos arquivos e definem uma regra vazia, somente para "demarcar território" e, de quebra, ainda poupar um pouco de trabalho futuro, caso seja preciso.

Dessa forma, não haveria qualquer contraindicação em se ter somente isto no arquivo `footer.css`:

```
.l-footer { }
```

Embora seja mais comum haver algumas propriedades e valores, como em:

```
.l-footer {
    background-color: #d0d0d0;
    color: #333;
}
```

Em relação aos arquivos em *modules*, como já explicado, estes são o cerne das folhas de estilo dentro de uma arquitetura SMACSS; são os arquivos que conterão a maioria das regras. Por isso, esse é o diretório mais volumoso. Faça como já está acostumado e escreva lindas, concisas e pensadas regras CSS para aplicar em seu projeto!

4.9 Conclusão

Como ficou evidente, SMACSS é uma arquitetura flexível, elaborada com base em anos de observação e experiência em projetos de alto nível.

Se esta foi a primeira vez que você teve contato com SMACSS, o conselho é que abrace a metodologia em alguns projetos para experimentar seus resultados. Se já havia ouvido falar, mas ainda não sabia bem do que se tratava, dê uma chance a SMACSS e, certamente, verá o nível de organização e produtividade de seus projetos melhorarem bastante!

Ainda é possível ir além com a eficiência de seu código e arquitetura. Lembre-se: muito da qualidade de código vem da padronização. Não seria muito bom caso houvesse uma maneira de se usar uma padronização já consagrada na hora de criar suas classes e, de brinde, essa abordagem ainda garantir que você identifique a estrutura de sua marcação no CSS instantaneamente?

No próximo capítulo, conheça **BEM**.

Capítulo 5

BEM

Você já ouviu falar no **Yandex**? Se você é russo, a probabilidade é bem maior; se não é, saiba que se trata do buscador com maior *market share* da terra de Putin (60%). Ele também é um dos mais usados buscadores do mundo, contando com centenas de milhões de buscas por dia.

A Yandex também oferece outros serviços, contando com um ecossistema web bastante significativo. Quando começou a crescer e sentir a necessidade de aprimorar e manter o front-end desse sistema, surgiu **BEM** (acrônimo para *Block, Element, Modifier* ou **Bloco, Elemento, Modi□cador**).

Na verdade, BEM é uma metodologia completa; um verdadeiro framework de front-end, que abarca diversas regras e ferramentas úteis. Tecnicamente, trata-se de uma arquitetura baseada em templates DSL construídos sobre `Node.js` (saiba mais a respeito em http://bem.info/) .

No ocidente, geralmente quando se fala sobre BEM, está se falando sobre a **convenção de nomenclatura** usada e, consequentemente, a maneira de

pensar as estruturas (ou **módulos**, se você já conhece SMACSS, mostrado no capítulo 4).

É essa convenção de nomenclatura para Bloco, Elemento e Modificador que você conhecerá agora.

> Não faz parte das regras estabelecidas pela BEM; mas, para os exemplos, serão usadas palavras em inglês, o que, dependendo do alcance e intenção de seus projetos, você também deveria fazer.

5.1 Pensando BEM

Como se sabe desde o capítulo sobre CSS orientado a objetos (3), os conceitos de Nicole Sullivan inspiraram diversos desenvolvedores, técnicas e metodologias. Coincidência ou não, a essência de OOCSS também pode ser encontrada em BEM: desacoplamento e reúso de código através de uma série de padrões preestabelecidos.

Em se tratando de BEM, seu uso vem na mudança de como se enxergam as estruturas/módulos que se está desenvolvendo e, obviamente, conhecer suas convenções de nomenclatura.

Em termos simples, temos:

- **Bloco**: entidade independente com seu próprio significado; é a abstração mais geral de um componente/módulo;
- **Elemento**: descendente (e parte) de um Bloco; ajuda a formá-lo, como um todo;
- **Modi□cador**: um estado ou "versão" diferente de um Bloco ou Elemento.

Segundo imagens disponibilizadas por uma das desenvolvedoras da própria Yandex, **Varvara Stepanova**, no memorável artigo escrito em 2012, *A New Front-End Methodology: BEM* (http://goo.gl/ZiBOoB) , será possível visualizar e entender melhor esses conceitos.

Veja a representação de uma página fictícia qualquer:

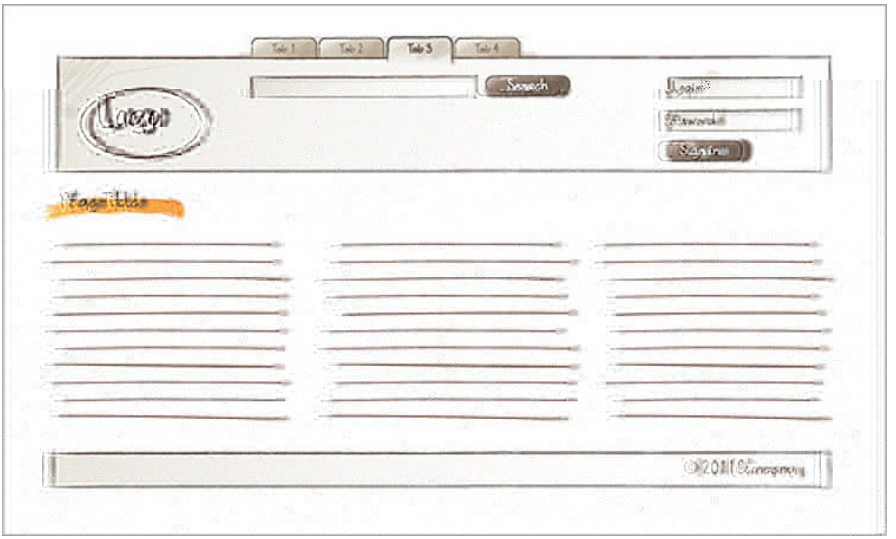


Fig. 5.1: Uma página fictícia qualquer

É possível "dividir" essa página (em um primeiro momento, mentalmente) em diversos Blocos:

Fig. 5.2: Uma página fictícia qualquer mentalmente 'dividida'

Veja mais de perto, por exemplo, o Bloco circulado de azul, Busca:

Fig. 5.3: Bloco de busca mentalmente separado em azul

É possível encontrar os Elementos que formam esse bloco:

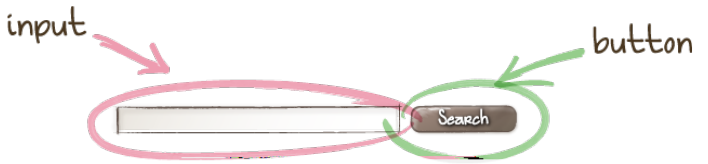


Fig. 5.4: Elementos do Bloco de busca

Como exemplo de modificador, poderíamos ter uma apresentação diferenciada para quando o foco no *input* acontece ou quando se submetem e se buscam informações dinamicamente.

Veja o Bloco das abas com seus respectivos Elementos:

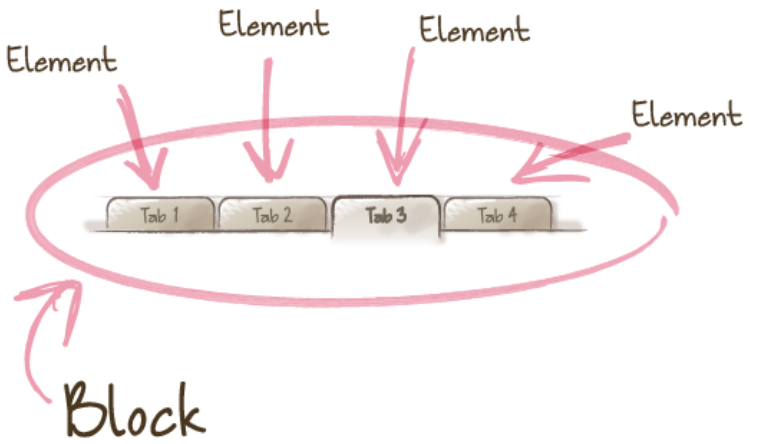


Fig. 5.5: Bloco de abas mentalmente separado em rosa

Nada impede que haja Blocos dentro de Blocos:

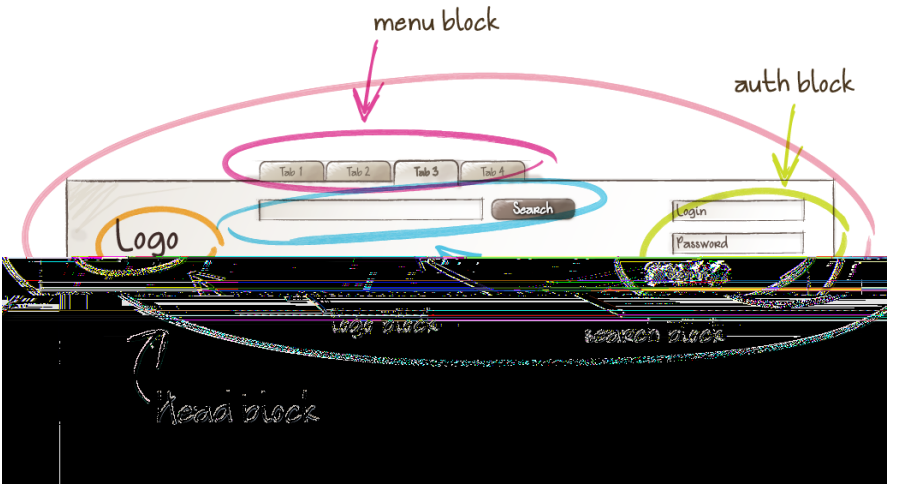


Fig. 5.6: Blocos dentro de Blocos

Na verdade, eles já estavam lá antes de você ver as imagens com as divisões e mesmo nos projetos em que você participou antes de ter lido qualquer coisa sobre tudo isso! Você ainda só não tinha aprendido a **visualizar BEM**! ;)

Por isso, a partir do momento em que se decide trabalhar com ele, a própria maneira de enxergar as estruturas de uma página se altera, uma vez que se começa a visualizá-las conforme os Blocos, Elementos e Modificadores, automaticamente.

Você vai saber o que é isso depois de usar BEM em alguns projetos, tenha certeza!

5.2 Escrevendo BEM

Agora que você já consegue "visualizar BEM" pode demorar um pouco para que isso aconteça, então, treine! , é hora de aprender a "escrever BEM". Quer dizer, tomar conhecimento de como é a convenção da sua sintaxe.

É simples:

```
.block {}
```

```
.block__element {}

.block--modifier {}
```

Nas convenções de sua nomenclatura, Blocos são classes com o próprio nome da estrutura; Elementos são delimitados com `__`; e Modificadores delimitados com `--`.

Pegando o exemplo mostrado antes do Bloco de busca:

```
.search {} /* Bloco */

.search__field {} /* Elemento */

.search--searching {} /* Modificador */
```

Essa convenção garante que todos os desenvolvedores que conheçam BEM identifiquem a estrutura em que o estilo atuará instantaneamente. O oposto também vale: ao se olhar o HTML, pelo nome das classes, já se sabe o que esperar do CSS que estiliza aquele pedaço de código.

Voltando à metaexemplos, veja alguns seletores para ajudar a fixar o conceito:

```
.person {}

.person__hand {}

.person--female {}

.person--female__hand {}

.person__hand--left {}
```

O Bloco `person` tem vários Elementos, como `hand`, e pode ter Modificadores, como `female`. Por sua vez, os Elementos também podem ter modificadores, tais como `left`.

Só de ter olhado para este metacódigo, você já foi capaz de saber que algumas das estruturas HTML correspondentes poderiam ser:

```
<div class="person">
    <div class="person__hand">
        [...]
    </div>
</div>

<div class="person--female">
    <div class="person__hand">
        [...]
    </div>
</div>

<div class="person">
    <div class="person__hand--left">
        [...]
    </div>

    <div class="person__hand">
        [...]
    </div>
</div>
```

E a recíproca é verdadeira: se tivesse olhado primeiro o HTML, já poderia inferir como seria o CSS.

Outra grande vantagem é que, usando BEM, você consegue escrever seletores menores e não conflitantes, evitando muitos dos problemas de especificidade CSS (2.1) que poderiam acontecer, fazendo seu código seguir aquelas boas práticas mencionadas.

Em outras palavras, **cada classe BEM é única e autossu□ciente!** Afinal, um seletor como este:

```
html #content > .person .hand {}
```

É muito mais verboso e menos eficiente (pensando em manutenibilidade e performance) do que este:

```
.person__hand {}
```

Isso sendo que ambos serviriam para estilizar exatamente os mesmos elementos!

Note que não é preciso que toda a estrutura do DOM esteja representada nos seletores. Mesmo que o HTML tivesse mais níveis, não seria preciso uma classe `.person__body__arm__hand`; uma `.person__hand` já conseguiria estilizar o que fosse necessário, e tornar a classe única e a estrutura inteligível.

> Idealmente, cada Bloco deve ser absolutamente independente; ou seja, deve ser possível mudá-lo de posição na página ou de página, sem que nenhuma alteração de CSS precise ser feita.

Lembra do objeto Media do capítulo sobre CSS orientado a objetos (3)? Usando BEM, seus seletores CSS poderiam ser escritos mais ou menos assim:

```
.media {}

    .media__img {}

        .media__img--rev {}

    .media__bd {}
```

Perceba que muitos também gostam de colocar níveis de indentação extras para ajudar ainda mais a identificar a estrutura de códigos. Com ou sem esse plus, são poucas as alterações que trazem muitas vantagens!

5.3 Conclusão

Como você viu, a convenção de nomenclatura de BEM tem muito de OOCSS: garante desacoplamento e reúso intenso de código por meio de uma série de padrões preestabelecidos; padrões, estes, facílimos de serem aprendidos e fixados.

Alguns desenvolvedores têm uma reclamação: "BEM é feio!". Parece que ter vários `__` e `--` por todo o HTML e CSS (e, inevitavelmente, JavaScript) não agrada a todos. Em minha opinião, isso não atrapalha em absolutamente nada e, de certa forma, é até divertido.

Mas se, para você, usar essa sintaxe for menos compensatório que:

- Código desacoplado;
- Reúso automático de código;
- Menos repetições;
- Rápida identificação de estruturas HTML através do CSS e vice-versa;
- Independência absoluta de classes;
- Seletores menores e mais performáticos;
- CSS mais manutenível.

A decisão fica por sua conta. Esperamos que escolha BEM! ;)

Entretanto, será que, depois de conhecer os conceitos sobre OOCSS (3), SMACSS (4) e BEM, ainda é possível aprimorar seus códigos e *workflow* ao desenvolver CSS? É o que você vai descobrir no próximo capítulo, sobre **pré-processadores CSS**.

Capítulo 6

Pré-processadores CSS

Se você estiver lendo este livro na sequência dos capítulos, neste ponto você já conhece sobre CSS orientado a objetos (3), SMACSS (4) e BEM (5). Para os acostumados a fazer só o feijão com arroz de CSS, são evoluções bastante consideráveis!

Porém, mesmo que você já tenha visto algo a respeito dessas técnicas ou tenha pesquisado e se aprofundado mais, ainda é possível aprimorar seus códigos e workflow ao desenvolver CSS! Existe uma maneira de usar características com que só pessoas que conhecem linguagens de programação estão acostumadas, e aplicar conceitos e técnicas de programação a suas folhas de estilo!

Respire fundo, acomode-se e seja bem-vindo ao mundo dos **pré-processadores CSS**!

6.1 O que são pré-processadores CSS?

Consultando a Wikipédia que está longe de ser uma fonte de consulta fidedigna, mas, para este caso, serviu , é possível saber que **um pré-processador é um programa que recebe texto e efetua conversões léxicas nele. As conversões podem incluir substituição de macros, inclusão condicional e inclusão de outros arquivos.**

Quer dizer, você escreverá código na sintaxe do pré-processador de sua preferência e, através de alguns processos de conversão, ele transforma (ou *compila*) essa sintaxe específica e própria em código CSS, a que todos já estamos habituados. Caso contrário, não se teria nenhuma estilização, já que, para este fim, os navegadores precisam de arquivos `.css`.

Aquela velha história de `Input -> Magic -> Output`. No caso específico dos pré-processadores de CSS, o `Input` é o código do pré-processador que você estiver usando e, o `Output`, o bom e velho CSS às vezes, chamado de *CSS puro*.

Neste momento, se você ainda não tinha ouvido falar (ou lido) sobre pré-processadores CSS, isso pode parecer exótico e/ou desnecessário. Afinal, para que incluir mais uma etapa no processo de escrita de CSS se já se pode escrever diretamente, como sempre se fez? É justamente ao responder essa questão que o poder dos pré-processadores se mostra!

Se não houvesse quaisquer vantagens em usá-los, certamente milhares de desenvolvedores pelo mundo não o fariam. Eles fazem-no pelo simples motivo de que os pré-processadores CSS contam com *features* que extrapolam as capacidades nativas do CSS! Algumas possibilidades com pré-processadores são:

- Usar variáveis;
- Usar funções;
- Aninhamento de regras;
- Operações aritméticas;
- Estender regras a partir de regras pré-existentes.

Se você já tem alguma vivência com programação, o simples vislumbre de poder usar este tipo de coisa em CSS já deve parecer fantástico; se não, o fato de isso parecer fantástico para quem já programa deve parecer fantástico! Vamos combinar o seguinte: pré-processamento de CSS é fantástico de qualquer jeito!

Através dessas e outras facilidades que eles oferecem, muitos desenvolvedores pelo mundo inteiro muitos, mesmo! estão adotando-os para melhorar seus fluxos de criação de estilo e organização de projetos. Continue lendo para também fazer parte dessa galera!

Os pré-processadores CSS mais conhecidos

Agora que já foi entendido que essa história toda de pré-processadores resume-se a escrever um pseudo-CSS que, por meio de processos específicos, será transformado em CSS de fato; saiba que, dentre os pré-processadores existentes atualmente, três deles são mais usados:

- Sass (http://sass-lang.com/) ;
- Less (http://lesscss.org/) ;
- Stylus (http://learnboost.github.io/stylus/) .

Fig. 6.1: Os pré-processadores CSS mais usados (e o Stylus)

Em função do uso global, o Stylus consta como menção honrosa. Nas rodinhas de boteco de programadores e noites do pijama para maratonas de

Senhor dos Anéis, quando o assunto for pré-processadores, certamente se falará sobre Sass e Less.

> Este capítulo reserva-se a explicar e demonstrar as principais características de pré-processadores CSS. Para instruções sobre como instalá-los em seu sistema, configurá-los e as maneiras possíveis de se compilar código, consulte os respectivos sites oficiais.

Basicamente, Sass e Less fornecem quase o mesmo acervo de funcionalidades. Há várias controvérsias e discussões acalouradas entre os adeptos de um e outro, mas a verdade é que, nessa quase conversão de características, ao optar por usar Sass ou Less, você estará bem servido e conseguirá melhorar **muito** a organização e workflow de desenvolvimento de estilos!

Pequena observação sobre a sintaxe Sass

Tenho uma preferência pessoal por **Sass**; todos os exemplos de códigos e features que serão mostrados a seguir serão feitos com código dessa sintaxe. Como acabou de ser explicado, Less também provê as maneiras de se fazer igual, de modo que não haverá prejuízo no entendimento.

A pequena observação é que Sass oferece **2 sintaxes diferentes** para se trabalhar:

- SCSS;
- Sass.

SCSS (Sassy CSS) que tem extensão de arquivos `.scss` pode ser considerada como uma extensão de CSS, valendo-se da mesma sintaxe-base e, praticamente, a mesma forma de escrever regras. Basicamente, se você abrir um arquivo `.scss` e escrever CSS puro ali, será um arquivo Sass válido.

A sintaxe Sass o mesmo nome do pré-processador , cuja extensão dos arquivos é `.sass`, já é bem diferente do CSS com que a maioria dos webdevs está acostumada, primando pela concisão e rapidez na escrita. Demora-se um pouco mais para aprender a escrevê-la, mas alguns garantem que, uma vez que se domina, é difícil voltar atrás.

Este capítulo propõe-se a apresentar a linguagem. Portanto, a sintaxe SCSS será usada por ser mais parecida com CSS puro, facilitando a assimilação e fixação dos conceitos.

6.2 Aninhamento de regras

Logo no primeiro dia de estudos de HTML, é fácil constatar que se trata de uma linguagem que usa elementos aninhados para funcionar: o aninhamento confere estrutura e organização hierárquica ao documento. CSS, por sua vez, não possui tal característica.

O poder de Sass começa a se mostrar já aqui: é possível **aninhar regras** CSS para escrever um código mais conciso e menor, conferindo hierarquia visual às regras e facilitando da leitura!

Por exemplo, veja este trecho de CSS:

```
nav ul {
    list-style: none;
    margin: 0;
    padding: 0;
}

nav li {
    display: inline-block;
}

nav a {
    display: block;
    padding: 6px 12px;
    text-decoration: none;
}
```

Em SCSS, é possível escrever assim:

```
nav {
    ul {
        margin: 0;
        padding: 0;
        list-style: none;
```

```
    }

    li {
        display: inline-block;
    }

    a {
        display: block;
        padding: 6px 12px;
        text-decoration: none;
    }
}
```

Não é incrível? Agora você pode aninhar regras de estilos e, ao compilar o código, será gerado o mesmo CSS mostrado anteriormente. A questão é que você poupou escrita e a declaração está visualmente mais inteligível, evidenciando, em um só golpe de vista, a hierarquia intrínseca das regras.

Referência ao ascendente

Em algumas situações, é preciso fazer referência ao ascendente para se montar uma regra apropriadamente. Por exemplo, dentro do aninhamento permitido por Sass, como seria para colocar uma regra com `:hover`?

Se tentar algo como:

```
.selector {
    :hover {
    }
}
```

O CSS compilado será `.selector :hover {}` (note o espaço), o que não é o necessário.

Para esses casos, Sass provê uma característica muito interessante, tornando possível fazer referência ao elemento ascendente imediatamente superior: o caractere `&`.

Neste exemplo do `hover`, o código correto seria:

```
.selector {
    &:hover {
```

```
    }
}
```

Isso geraria o CSS: `.selector:hover {}`, exatamente o que está se buscando.

Sempre que, em Sass, você deparar-se com `&`, já sabe que a intenção é repetir o seletor ascendente imediato na regra, independente de que parte dela esteja. Veja:

```
.selector {
    .parent-selector--modified & {

    }
}
```

Isso vai compilar para:

```
.parent-selector--modified .selector {}
```

O seguinte código:

```
#custom-checkbox {
    & + label {
    }

    &:checked + label {
    }
}
```

compilará o CSS:

```
#custom-checkbox {
}

#custom-checkbox + label {
}

#custom-checkbox:checked + label {
}
```

Sass e BEM

O padrão de nomenclatura BEM traz diversas vantagens no desenvolvimento de front-end, como explicado no capítulo 5. Porém, conforme foi mostrado até agora sobre o caractere de referência ao ascendente imediato, um trecho Sass do tipo:

```
.selector {
    &--modifier {
    }
}
```

Compilaria para:

```
.selector .selector--modifer {}
```

o que, definitivamente, não é o esperado.

Felizmente, desde a versão Sass 3.4, os adeptos ao BEM contam com uma grata característica: uma compilação especial quando se usa o padrão de nomenclatura!

Sass identifica automaticamente os padrões de escrita BEM e gera CSS da maneira que se espera! Caso em que se pode escrever tranquilamente o seguinte:

```
.selector {
    &--modifier {
    }

    &__subelement {
        &--submodifier {

        }
    }
}
```

E obter o compilado:

```
.selector--modifier {
}
```

```
.selector__subelement {
}

.selector__subelement--submodifier {
}
```

Também é possível escrever tantas referências mais aos respectivos elementos ascendentes imediatos quanto sejam precisas, já que, a partir dessa versão, o Sass compreende a sintaxe BEM e gera CSS corretamente, respeitando-o e garantindo regras com boa performance de seletores-chaves (2.2)!

6.3 Variáveis

Se o poder dos pré-processadores de CSS limitasse-se somente a permitir o uso de **variáveis**, ainda assim seriam tão famosos e usados como são atualmente! Poder usar variáveis é, simplesmente, um incrível avanço na organização e manutenibilidade de um conjunto de folhas de estilo!

Variáveis em Sass são como variáveis em qualquer linguagem de programação: referências nominais capazes de armazenar valores que podem ser chamadas em trechos de código subsequentes para resgatar e usá-los conforme necessário. Para declarar uma variável, basta dar um nome qualquer (sem espaços e/ou caracteres especiais) precedido de `$`, usar `:` e dar o valor que se queira, como em:

```
$mainColor: #c0ffee;
```

Isso feito, basta usar a variável à vontade! Por exemplo:

```
$mainColor: #c0ffee;

header {
    background-color: $mainColor;
}

a {
```

```
    color: $mainColor;
}

footer .highlight {
    border: 2px solid $mainColor;
}
```

que compilaria:

```
header {
    background-color: #c0ffee;
}

a {
    color: #c0ffee;
}

footer .highlight {
    border: 2px solid #c0ffee;
}
```

A imensa vantagem é poder substituir absolutamente todas as referências àquela cor, alterando somente uma linha de código. Mesmo nesse exemplo com somente 3 regras, já seria muito bom. Imagine em projetos do mundo real, em que não é raro precisar de alterações dessa natureza em dezenas de regras!

Variáveis podem conter quaisquer valores usados em CSS, como em:

```
$font-stack: Helvetica, sans-serif;
$width: 5em;
```

Quando não são declaradas fora de regras como em todos os exemplos mostrados até agora , a variável fica disponível em todo o código. Mas também é possível limitar seu escopo de atuação ao declará-la dentro de uma regra:

```
.l-main {
    $width: 5em;
    width: $width;
}
```

Se tentássemos usar `$width` em qualquer outro lugar fora de `.l-sidebar`, isso resultaria em erro.

A não ser que seja usado o modificador `!global`, que faz com que ela esteja disponível também em qualquer ponto do código. Nesse caso, seria possível algo como:

```
.l-main {
    $width: 5em !global;
    width: $width;
}

.l-sidebar {
    width: $width;
}
```

que seria compilado para:

```
.l-main {
    width: 5em;
}

.l-sidebar {
    width: 5em;
}
```

Interpolação de variáveis

Sass permite que seja possível interpolar variáveis. Em termos simples, seria como juntar o valor de variáveis com outras variáveis ou valores.

Para se fazer isso, referencia-se a variável usando `#{VARIAVEL}` e, automaticamente, a interpolação acontece. Para ficar mais claro, veja o exemplo:

```
$vert: top;
$horz: left;
$radius: 5px;

.rounded-#{$vert}-#{$horz} {
    border-#{$vert}-#{$horz}-radius: $radius;
}
```

que gera o CSS:

```
.rounded-top-left {
    border-top-left-radius: 5px;
}
```

6.4 Mixins

Mixins permitem que se façam agrupamentos de declarações CSS para serem reusados onde se queira. Certamente, são uma das features mais poderosas de Sass! Se você já brincou com programação, mixins lembram bastante funções.

Para se trabalhar com eles, o par `@mixin`/`@include` sempre estará presente. O primeiro para definir o mixin, em si; o segundo, para indicar em qual ponto do código se quer usá-lo.

Veja este trecho de código Sass com um mixin para arredondamento de bordas sendo definido e usado:

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

.box {
    @include border-radius(10px);
}
```

Isso gerará o seguinte CSS:

```
.box {
    -webkit-border-radius: 10px;
       -moz-border-radius: 10px;
        -ms-border-radius: 10px;
            border-radius: 10px;
}
```

Com 6 linhas de código, você agora pode arredondar bordas em qualquer regra que quiser, apenas usando um `@include`! Como plus, ainda consegue especificar qual será o raio na própria chamada ao mixin!

Essa possibilidade de passar o valor que se quer para atuar em mixins (*argumentos*) certamente faz deles mais poderosos. É possível passar quantos quiser, que nem as funções em linguagens de programação. Caso se passe um argumento já especificando um valor, este será o valor padrão usado desde que ele seja omitido ao se usar o `@include`. Veja:

```
@mixin rounded($vert, $horz, $radius: 10px) {
    -webkit-border-#{$vert}-#{$horz}-radius: $radius;
       -moz-border-radius-#{$vert}#{$horz}: $radius;
            border-#{$vert}-#{$horz}-radius: $radius;
}

.l-footer {
    @include rounded(top, left, 5px);
}

.l-sidebar {
    @include rounded(top, left, 8px);
}

.navbar li {
    @include rounded(top, left);
}
```

Isso compila para:

```
.l-footer {
    -webkit-border-top-left-radius: 5px;
       -moz-border-radius-topleft: 5px;
            border-top-left-radius: 5px;
}

.l-sidebar {
    -webkit-border-top-left-radius: 8px;
       -moz-border-radius-topleft: 8px;
            border-top-left-radius: 8px;
}

.navbar li {
```

```
    -webkit-border-top-left-radius: 10px;
       -moz-border-radius-topleft: 10px;
            border-top-left-radius: 10px;
}
```

Eu quase consigo ouvir o som de suas lágrimas chocando-se com o solo!

6.5 Extensão/Herança

Indubitavelmente, Extensão/ Herança é uma das principais características de pré-processadores CSS!

No caso específico de Sass, é possível usar `@extend` para compartilhar uma série de propriedades/valores de várias regras diferentes em uma mesma regra! Isso mantém o código dentro do princípio da não repetição (*DRY*), faz com que seja preciso escrever bem menos código e, de quebra, ainda garante que o CSS compilado seja mais lógico e eficiente!

Como exemplo, suponha que você tenha esta regra:

```
.default-box {
    background-color: #efefef;
    border: 1px solid #000;
    color: #333;
}
```

Caso seja preciso criar variações disso com Sass, uma das maneiras possíveis seria estendê-la em outra regra! Assim:

```
.alert-box {
    @extend .default-box;
    font-size: 2em;
}
```

o que compilaria para:

```
.default-box, .alert-box {
    background-color: #efefef;
    border: 1px solid #000;
    color: #333;
```

```
}

.alert-box {
    font-size: 2em;
}
```

Percebe o poder que isso oferece? Você pode ter um arquivo com definições genéricas de regras que podem ser estendidas em quaisquer outras do projeto! Por exemplo, regras do tipo:

```
[...]

.bold {
    font-weight: bold;
}

.italic {
    font-style: italic;
}

.display-block {
    display: block;
}

.display-none {
    display: none;
}

.underline {
    text-decoration: underline;
}

.no-underline {
    text-decoration: none;
}

.text-left {
    text-align: left;
}
```

```
.text-center {
    text-align: center;
}

.text-right {
    text-align: right;
}

.text-justify {
    text-align: justify;
}

[...]
```

Se quiséssemos uma regra para um módulo que precise ter apresentação em bloco, negrito e texto justificado, só seria necessário estender regras já especificadas anteriormente!

```
.my-module {
    @extend .display-block, .bold, .text-justify;
    border: 1px solid #ccc;
}
```

que compilaria para:

```
[ ... ]

.bold, .my-module {
    font-weight: bold;
}

.display-block, .my-module {
    display: block;
}

.text-justify, .my-module {
    text-align: justify;
}
```

```
[ ... ]

.my-module {
    border: 1px solid #ccc;
}
```

É ou não é uma economia tremenda de digitação com um compilado condizente ao melhor que CSS tem a oferecer?

Seletores placeholder

No exemplo anterior, foi possível observar que todas as regras especificadas no Sass estão presentes no CSS gerado. É até conveniente que classes, como `.bold`, `.display-block` etc. sejam geradas, já que podem ser úteis ao se trabalhar com JavaScript, adicionando e excluindo classes conforme eventos e ações.

Entretanto, pode haver situações em que determinadas regras que serão estendidas só precisem existir para isso e não precisem estar presentes no CSS compilado. Para isso, saiba que existem o **seletores placeholder** (*placeholder selectors*).

Criar um placeholder selector é como uma regra comum, com a diferença de que não se coloca um elemento, classe ou ID, mas sim um %. Por exemplo, se fosse preciso usar um seletor placeholder para estender nos estilos mostrados antes, ficaria:

```
[ ... ]

%bold {
    font-weight: bold;
}

%italic {
    font-style: italic;
}

%display-block {
    display: block;
}
```

```
%display-none {
    display: none;
}

%underline {
    text-decoration: underline;
}

%no-underline {
    text-decoration: none;
}

%text-left {
    text-align: left;
}

%text-center {
    text-align: center;
}

%text-right {
    text-align: right;
}

%text-justify {
    text-align: justify;
}

[ ... ]
```

Para indicar a extensão:

```
.my-module {
    @extend %display-block, %bold, %text-justify;
    border: 1px solid #ccc;
}
```

A grande diferença está no CSS compilado, que seria somente:

```
.my-module {
    font-weight: bold;
}

.my-module {
    display: block;
}

.my-module {
    text-align: justify;
}

.my-module {
    border: 1px solid #ccc;
}
```

Observando o compilado, parece que foi um desperdício definir várias regras com o mesmo seletor com somente uma propriedade definida. Entretanto, lembre-se de que os placeholder selectors podem ser estendidos em qualquer lugar, consequentemente alterando a compilação de folhas de estilo conforme seu uso.

Outra ocasião bastante útil para eles é quando se tem que usar fontes personalizadas em projetos. Dá pra fazer algo como:

```
@font-face {
    font-family: 'custom_font';
    src: url('font/my-custom-font.eot');
    src: url('font/my-custom-font.eot?#iefix')
            format('embedded-opentype'),
         url('font/my-custom-font.woff2') format('woff2'),
         url('font/my-custom-font.woff') format('woff'),
         url('font/my-custom-font.ttf') format('truetype');
    font-weight: normal;
    font-style: normal;
}

%custom-font {
    font-family: 'custom_font';
}
```

Sempre que precisar que algum elemento, classe ou ID seja agraciado com a fonte diferenciada, estenda o seletor placeholder.

Em suma, **placeholder selectors não aparecem no CSS compilado**, servindo exclusivamente para propósitos de Extensão/ Herança.

6.6 Importação

Levando em conta o que se aprendeu no capítulo (4) sobre SMACSS e as vantagens do Sass ao permitir Extensão/ Herança, não surpreenderia se você já tivesse chegado à conclusão de que, em projetos mais profissionais, é preciso que se tenha diversos arquivos diferentes, cada um com seu propósito e contendo suas respectivas regras de estilo.

Para essas ocasiões, o Sass oferece um recurso de importação, usando `@import`.

No exemplo de placeholder selectors mostrado agora há pouco, certamente haveria um arquivo Sass com todos eles e, conforme a necessidade e a organização do projeto por exemplo, o uso da metodologia SMACSS , a criação de mais arquivos para conterem as regras devidas. Em uma demonstração simples, 2 arquivos:

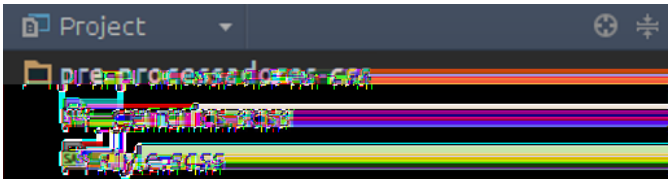


Fig. 6.2: Demonstração simples com 2 arquivos

Logo no início do arquivo `style.scss`, bastaria colocar:

```
@import 'generics';
```

Tudo o que está dentro de `_generics.scss` poderia ser acessado. Não é preciso colocar o `_` nem `.scss` na sintaxe de importação. Também é possível e, quase sempre, necessário importar vários arquivos, separando-os por vírgula na declaração do `@import`.

Neste ponto, cabem duas observações importantes.

A primeira é que se deve ter atenção ao nome do arquivo que foi importado, que começa com um subtraço (*underline*). Dentro da dinâmica do Sass, isso é chamado de **partial** e significa que, na compilação, somente será gerado um arquivo CSS de `style.scss`. Em outras palavras, na compilação, partials não geram sua contraparte `.css`.

A segunda é que é importante não confundir esse `@import` com a importação de arquivos que já existe em CSS puro. Esta faz uma requisição extra no servidor, chamando um arquivo à parte; aquela junta todos os arquivos importados no importador, garantindo a geração de um arquivo único com todas as regras.

> Por padrão, Sass não permite importar arquivos `.css` diretamente.

Para finalizar, é possível a importação de arquivos em subdiretórios. Em um projeto com estrutura Sass, por exemplo, poderia haver um arquivo único que importasse todos os partials necessários:

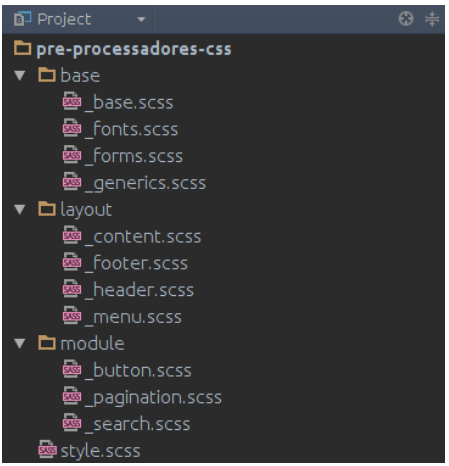


Fig. 6.3: Exemplo de estrutura com arquivos .scss

No início do arquivo `style.scss`, colocaríamos:

```
@import
    'base/base',
    'base/fonts',
    'base/forms',
    'base/generics',
    'layout/content',
    'layout/footer',
    'layout/header',
    'layout/menu',
    'module/button',
    'module/pagination',
    'module/search'
;
```

Cada arquivo importado poderia também ter declarações de `@import`, o que causaria um efeito bola de neve de importação e, como o esperado, ainda continuaria compilando somente o arquivo `style.css`.

6.7 Mais superpoderes de Sass

Sass é uma linguagem poderosíssima, que facilmente encheria um livro inteiro, além do que já foi apresentado:

- Operadores aritméticos (`+`, `-`, `*`, `/`);
- Estruturas de controle (`if`, `while`, `for`, `each`);
- Diferentes resultados de compilação (*nested*, *expanded*, *compact*, *compressed*);
- Funções personalizadas.

Tudo isso você encontra no site oficial do Sass (http://sass-lang.com/) . Para já ir brincando um pouquinho sem ter que instalar a ferramentaria necessária para se trabalhar, acesse o SassMeister (http://goo.gl/aQYSpg) . Depois disso, para continuar os estudos, há artigos de grande utilidade no *The Sass Way* (http://goo.gl/GEjBGe) .

Para aprimorar ainda mais a Arquitetura CSS de seus projetos, de uma maneira que permita uma melhor identificação da sua estrutura e manutenção, no próximo capítulo, veja sobre **CSS Namespaces**.

Capítulo 7

CSS namespaces

Como visto, SMACSS (4) e BEM (5) podem fazer maravilhas pela arquitetura e organização de seus projetos. Porém, é possível ir mais além! Espero que não fique surpreso!

Em diversas linguagens de programação bastante usadas no mundo inteiro, existe o conceito de **namespaces**. Algumas o suportam nativa e explicitamente, como, por exemplo, a maioria das linguagens de *backend* modernas; mas, outras, como JavaScript, não o aceitam. Entretanto, é possível fazer algumas marotices no código (leia-se: gambiarras) para que namespacing possa ser aplicado e conseguir seus benefícios intrínsecos obtidos.

Porém, o que são namespaces e qual é a sua finalidade?

Tentando explicar de maneira simples e sucinta, namespaces são como regiões no código, nas quais nomes de variáveis, de funções etc. são válidos dentro destas linguagens de programação. Apesar de não ter certeza quanto à

acuidade da tradução, em terras tupiniquins, alguns costumam usar o termo **espaços de nomes**.

Na prática, isso quer dizer que, dentro de determinado "espaço de nome", é possível atribuir nomes a instâncias e entidades sem que ocorra algum conflito caso termos idênticos sejam escolhidos em outros espaços de nomes.

Ou seja, torna-se possível usar termos idênticos dentro de namespaces diferentes sem que haja confusão. Adicionalmente, essa sua diferenciação traz a vantagem extra de ser possível identificar imediatamente a qual espaço de nome tal entidade pertence com um só golpe de vista.

Evidentemente, no momento, um CSS que não seja uma linguagem de programação não pode valer-se dos benefícios completos de namespacing. Mas, tal como acontece com OOCSS (3), por meio de algumas convenções bastante interessantes, é possível trazer parte dos seus benefícios para a linguagem e tentar encontrar alguma vantagem nisso.

É mais simples do que você poderia imaginar.

> Este capítulo não é sobre o Namespace Module da W3C (http://goo.gl/EoQjvt) ; mas sobre convenções de nomes de classes para ajudar na leitura e compreensão de códigos.

7.1 Os CSS namespaces mais comuns

Como você deve saber, por não se tratar de uma linguagem de programação, aplicar namespacing em CSS é possível através de **convenções de nomes** assim como muito de bom que acontece em CSS. Na prática, trata-se apenas de colocar um prefixo nos nomes dos seletores para conseguir atribuir um namespace e começar a se valer dos benefícios de imediata identificação e melhor compreensão do código.

Os namespaces de CSS mais comuns mostrados por **Harry Roberts** em um conhecido artigo a respeito (http://goo.gl/917hCW) são:

- **o-**: objeto;
- **c-**: componente;

- **u-**: utilitário;
- **t-**: tema;
- **s-**: escopo (*scope*);
- **is-**/**has-**: estado/ condição;
- _ : hack (!);
- **js-**: JavaScript;
- **qa-**: quality assurance (QA).

Só de observar essa lista, já é possível perceber o quão mais comunicativo seu código pode ser ao, simplesmente, prefixar os nomes de suas classes! Veja mais detalhes sobre cada um dos namespaces a seguir.

Além disso, há uma vantagem extra: usar namespaces em CSS combina perfeitamente com BEM (5)! Quer dizer, você pode continuar escrevendo código CSS com Bloco, Elemento e Modificador, aprimorando a arquitetura, legibilidade e entendimento do código, somente ao usar essa convenção de prefixos!

Objeto: o-

Formato:

```
.o-object-name[<element>|<modifier>] {}
```

Exemplo:

```
.o-layout {}

    .o-layout__item {}

    .o-layout--fixed {}
```

O namespace **Objeto** é especialmente útil para pessoas que se valem do OOCSS, que, a esta altura, você já deve conhecer depois de ter lido o capítulo 3.

Dado um projeto fictício que segue algumas das boas práticas para CSS eficiente já mostradas, imagine que seja preciso que se altere o `padding` das recomendações dadas por pessoas, por exemplo. Encontra-se o seguinte trecho de código:

```
<blockquote class="media testimonial">
</blockquote>
```

Em relação ao CSS correspondente, foi decidido colocar o `padding` em `.testimonial` para resolver a questão. Daria certo, mas, fazendo alguns testes, opta-se por fazer a adição da propriedade em `.media`, já que isso traz o resultado esperado.

A grande questão nesse exemplo é que `.media` é somente uma abstração de OOCSS. Ele tem de manter suas características de ser reusável e de não receber "cosméticos" que poderiam alterar seu visual dependendo de onde seja colocado. Mesmo que o código mostrado já tenha a aplicação de algumas técnicas de CSS eficiente, é possível melhorá-lo e torná-lo mais claro e comunicativo.

Bastaria fazer:

```
<blockquote class="o-media testimonial">
</blockquote>
```

Somente com a adição do prefixo do namespace Objeto, agora o código comunica a qualquer outro desenvolvedor sobre sua natureza de objeto/ abstração e também que ele não deve ser estilizado. Isso evitaria alterações nefastas nele. Logo, vemos que:

- Objetos são abstratos;
- Objetos podem ser usados em qualquer número ao longo do projeto;
- Tenha cuidado e atenção redobrada quando se deparar com elementos `o-`.

Componente: c-

Formato:

```
.c-component-name[<element>|<modifier>] {}
```

Exemplo:

```
.c-modal {}

    .c-modal__title {}

    .c-modal--gallery {}
```

O namespace **Componente** identifica a maior parte dos elementos de UI que qualquer um pode olhar e identificar imediatamente, por exemplo: "Isso é um botão", "Isso é um modal" etc.). Ao contrário de Objetos, alterações de estilo em Componentes não afetam nada além do componente em si, como vemos nesse último exemplo, nada além do modal. Enquanto objetos são abstrações, componentes são implementações específicas.

Para aprimorar o exemplo dado anteriormente, bastaria:

```
<blockquote class="o-media c-testimonial">
</blockquote>
```

Se quando a solicitação de aumento de `padding` tivesse sido feita com essa marcação presente, imediatamente já se teria percebido que ela precisaria ser feita a `.testimonial`, dada sua prefixação de componente. Teríamos a garantia de que toda e qualquer modificação realizada afetaria somente componentes `.testimonial` e nada mais! Assim:

- Componentes são implementações específicas de UI;
- Podemos modificá-los de maneira segura;
- Qualquer coisa com prefixo `c-` é algo específico.

Utilitário: u-

Formato:

```
.u-utility-name {}
```

Exemplo:

```
.u-clearfix {}
```

O namespace **Utilitário** é usado para regras que têm responsabilidade única, servindo a propósitos muito específicos e diretos e fazendo uma, e somente uma, coisa. Por exemplo, `.u-text-center` para alinhar ao centro um texto. Não é incomum que tenham `!important` para garantir que ganharão na batalha da especificidade CSS (2.1).

Devido à sua natureza de estilização focada, excepcional, única, direta e que também permite sua utilização globalmente no projeto, é muito importante usar esse prefixo quando este vale-se de namespaces CSS.

Por exemplo, veja este trecho de código:

```
.footer .text-center {
    font-size: 75%;
}
```

O problema é que `.text-center` agora tem duas responsabilidades quando aparece em qualquer lugar dentro de `.footer`. Isso causa efeitos colaterais, algo que um utilitário jamais jamais! pode ter.

Isso nos traz a uma conclusão óbvia a respeito do uso do namespace Utilitário: se começa com `u-`, **jamais reatribua!** Utilitários devem ser definidos uma vez para nunca mais precisarem ser alterados.

Veja este exemplo de HTML:

```
<div class="font-size-large">
    <blockquote class="pullquote">
    </blockquote>
</div>
```

Somente olhando para o código, pode ser confuso para um desenvolvedor perceber o porquê de o tamanho da fonte do `blockquote` ser diferente do

que estava sendo esperado. O motivo é que ele está herdando esse tamanho de `.font-size-large`.

Veja como é possível clarear o código e garantir que este tipo de dúvida não ocorra, somente usando prefixos de namespaces de CSS:

```
<div class="u-font-size-large">
    <blockquote class="c-pullquote">
    </blockquote>
</div>
```

Logo, vimos que:

- Utilitários são estilos da pesada!
- Alerte a outros desenvolvedores sobre eles, colocando o prefixo `u-`.
- Nunca os reatribua a qualquer elemento que tenha um prefixo `u-`.

Tema: t-

Formato:

```
.t-theme-name {}
```

Exemplo:

```
.t-light {}
```

O namespace **Tema**, como sugere o próprio nome, é para auxiliar na marcação HTML e na criação de seletores CSS em projetos que suportam temas.

A maioria dos projetos que suportam temas trabalham com o conceito de **Stateful □ emes** (http://goo.gl/Vrpyto) . Quer dizer, temas que podem ser habilitados e desabilitados; por exemplo, pense em sites que em cada página ou seção tem uma paleta de cores ou um seletor de temas num painel de controles. Geralmente, isso é feito colocando-se alguma classe que indica que o tema está no elemento `body`.

Valendo-nos do que foi visto sobre pré-processadores CSS (9), é possível pensar em códigos como:

```
.c-btn {
    background-color: #333;
    color: #e4e4e4;
    display: inline-block;
    padding: 1em;

    .t-light & {
        background-color: #e4e4e4;
        color: #333;
    }
}
```

Ou seja, o elemento `.c-btn` possui determinado estilo por padrão, mas, quando o tema light está ativo a classe `.t-light` é colocada no elemento `body`, as cores de background e foreground invertem-se. Como mostrado, a vida ainda fica muito mais fácil ao utilizar pré-processadores. Caso se precise editar o aspecto de algum elemento em um tema específico, já se sabe onde encontrar sua definição, uma vez que ficam todas vinculadas ao elemento padrão (no mesmo arquivo). Assim:

- Namespaces de tema são de alto nível;
- Eles proveem contexto/ escopo para diversas outras regras;
- São úteis para sinalizar a condição atual da UI.

Escopo: s-

Formato:

```
.s-scope-name {}
```

Exemplo:

```
.s-cms-content {}
```

O namespace **Escopo** (*scope*) deve ser usado somente em alguns casos bastante específicos. Preste muita atenção para não usá-lo de maneira inapropriada.

Na maioria das vezes, escopo pode ser utilizado para especificar um contexto para alguma seção particular da UI. Um exemplo seria o código HTML gerado por um CMS que, sabemos, nem sempre é da melhor qualidade. Você pode utilizar um namespace de escopo como um contêiner e, com a porção de elementos contextualizada, estilizar a partir daí.

Uma porção de código de exemplo:

```
<nav class="c-nav-primary">
    ...
</nav>

<section class="s-cms-content">
    <h1>...</h1>

    <p>...</p>

    <p>...</p>

    <ul>
        ...
    </ul>

    <p>...</p>
</section>

<ul class="c-share-links">
    ...
</ul>

<a href="#" class="c-btn c-btn--primary">
    Pr&oacute;ximo artigo
</a>
```

Veja que `.s-cms-content` serve como um contêiner para o conteúdo automático gerado e, com isso, abre-se a possibilidade de uma estilização fina para adequá-lo às diretrizes de UI do projeto.

Um exemplo (com Sass):

```
.s-cms-content {
```

```
    font: 16px/1.5 serif;

    h1, h2, h3, h4, h5, h6 {
        font: bold 100%/1.5 sans-serif;
    }

    a {
        text-decoration: underline;
    }
}
```

Como você já deve saber a esta altura, estilizar elementos diretamente não é lá considerado uma boa prática para se ter CSS eficiente. Porém, em casos como esse, não deixa de ser uma solução aceitável para conseguir-se uma estilização apropriada.

Em virtude disso e dos problemas de CSS que podem aparecer por consequência, é importante lembrar que o namespace Escopo deve ser usado com cautela e somente quando realmente necessário. Portanto:

- Escopos devem ser raramente usados, logo, tenha certeza absoluta de que precisa deles;
- Eles dependem inteiramente de aninhamento; esteja ciente disso.

Estado/Condição: is-/has-

Formato:

```
.[is|has]-state {}
```

Exemplos:

```
.is-open {}

.has-dropdown {}
```

O namespace **Estado/ Condição** advém do SMACSS (4) e, como explicado, serve para indicar estados temporários da UI que precisam ser estilizados de acordo com essa condição.

Não raramente, ao usar um inspetor de código (por exemplo, Chrome DevTools) para saber mais sobre algum elemento interativo de UI, como um modal, é comum vermos classes/ elementos aparecendo e desaparecendo, "ligando" e "desligando". Ao inspecionar-se um código, é bastante esclarecedor ver uma classe como `is-open` aparecer e desaparecer, pois informa de maneira muito clara o que está acontecendo naquele trecho específico da interface.

```
<div class="c-modal is-open">
    ...
</div>
```

Também não somente quando se olha para o HTML. Ao olhar para um arquivo de estilos com esse namespace, tudo fica muito mais claro.

```
.c-modal {
    &.is-open { ... }

    &__content {
        &.is-loading { ... }
    }
}
```

Algo importante a saber é que um estado/ condição diferencia-se de um Modificador de BEM (5) devido à característica de ser temporário. Ou seja, um estado/ condição pode mudar de um momento para o outro, devido a alguma ação tomada pelo visitante (por exemplo, `.is-expanded`) ou alguma interação com o servidor, como `.is-updating`. Logo, vimos que:

- Estado/ condição é temporário;
- Certifique-se de que sejam facilmente percebidos e entendidos no HTML;
- Nunca escreva uma classe de estado/ condição "pura". Ele é sempre atrelado a outra.

Hack: _

```
._<namespace>hack-name {}
```

Exemplo:

```
._c-footer-mobile {}
```

O namespace **Hack**, devido a seu próprio nome e objetivo que decorre daí, deve ser usado somente em casos raros. Ter essa marcação no código ajuda todos a identificar que se trata de um hack que (teoricamente) deve ser resolvido de maneiras convencionais, assim que possível.

Exemplo:

```
@media screen and (max-width: 30em) {
    // É preciso forçar o footer a ter uma largura fixa
    // em viewports menores
    ._c-footer-mobile {
        height: 80px;
    }
}
```

Resumindo:

- Hacks são feios;
- Hacks devem ser temporários, portanto não faça reúso ou vincule-os a suas classes;
- Fique atento ao número de hacks no código.

JavaScript: js-

Formato:

```
.js-component-name {}
```

Exemplo:

```
.js-modal {}
```

O namespace **JavaScript**, como você deve imaginar, indica que há algum tipo de interação com JavaScript associada ao elemento em que se encontra. Em teoria, não se deve ter estilo e comportamento associados ao mesmo *hook* no código, logo, geralmente esse namespace é usado para fazer a **separação de preocupações** (*Separation of Concerns*).

Usá-lo também traz uma tranquilidade extra ao projeto, já que desenvolvedores podem mexer no CSS à vontade sem se preocupar com alguma possível quebra em JS e vice-versa.

Portanto, na prática, classes como `.js-modal` pouco têm a ver com CSS e estilização, servindo mais como hooks JavaScript, propriamente ditos. Dependendo do guia de estilos adotado e convenções da equipe, nada impede que sejam `.jsModal`. Assim:

- JavaScript e CSS têm preocupações diferentes (*Separation of Concerns*), então use hooks diferentes;
- Tendo hooks diferentes para CSS e JS, a codificação em cada um deles torna-se mais segura.

Quality Assurance (QA): qa-

Formato:

```
.qa-node-name {}
```

Exemplo:

```
.qa-error-login {}
```

O namespace **Quality Assurance (QA)**, apesar de não ser o mais usado da lista, pode ser extremamente útil e valioso quando se está trabalhando com testes de UI e com ferramentas de automatização como **Selenium** (http://goo.gl/N7523e) . Ou até mesmo com um navegador *headless*, em que é extremamente comum scripts do tipo:

1) Visite `site.dev/login`;
2) Insira um nome de usuário incorreto;

3) Insira uma senha incorreta;

4) Espere (*Expect*) por um erro no DOM.

Até aqui, nenhuma novidade. Mas, pense bem: se os testes são feitos usando como hooks nomes de classes CSS e é isso mesmo o que geralmente acontece , uma simples troca de nomes pode fazer com que eles comecem a falhar!

Fazendo um paralelo com o namespace `js-`, aqui também seria conveniente que acontece-se uma separação de preocupações. Isso permitiria que diferentes pessoas ou equipes pudessem atuar de maneira independente, fazendo o que quer que fosse preciso.

Para melhorar a situação, uma medida interessantíssima a ser tomada é começar a usar o namespace QA e, para os que fazem os testes, a utilizar essas classes como hooks; em vez das normais presentes no código, ficando com classes como neste exemplo:

```
<strong class="c-message c-message--error qa-error-login">
```

Resumindo:

- Usar hooks de estilo para testes é um tanto obscuro, então, não faça isso;
- Use-os como hooks classes específicas para testes;
- Garanta que qualquer refatoração de UI não afete os hooks de QA.

7.2 Conclusão

Como muitas das boas práticas de CSS, **CSS Namespaces** nada mais são do que convenções de codificação que, se bem usadas, trazem numerosas vantagens aos projetos que se valem delas, almejando um CSS eficiente. Tecnicamente, são apenas pseudonamespaces, mas, de qualquer forma, dentro das atuais limitações de CSS, ajudam bastante.

Existem até vantagens não explícitas, como o fato de conseguir-se mais produtividade com editores/ IDEs com *autocomplete*.

Fig. 7.1: Por essa você não esperava!

Se, por qualquer motivo, seja preciso destacar visualmente um ou mais namespaces, isso torna-se possível com uma simples regras, como:

```
[class^="c-"],
[class*=" c-"] {
    outline: 5px solid red;
}
```

A lista de namespaces apresentados não é fixa e nem imutável, só reflete o que é mais comumente usado por aí, enquanto você está lendo estas palavras. Apesar de não ser única, como se pode ver em **FUN** (http://goo.gl/GocWkX) e outras propostas existentes.

Devido à facilidade de aprendizado e de uso de CSS Namespaces, somados aos imensos benefícios e ganhos que são trazidos aos projetos, realmente não há motivos para que os namespaces também não façam parte do seu rol de convenções CSS, apesar de ser possível usar somente outras das técnicas de CSS eficiente mostradas no livro.

Automatizar tarefas é sempre algo bom a se fazer e certamente usar **Task Runners** em seus projetos é um passo inteligente a ser dado, o que veremos a seguir.

Capítulo 8

Task Runners

Se você estiver lendo este livro sequencialmente, você já viu CSS orientado a objetos (3), SMACSS (4), BEM (5), pré-processadores CSS () e CSS namespaces (7). Um belo caminho rumo à escrita de CSS mais eficiente!

Algo que pode passar essa eficiência a um outro nível é a **automatização**. É bastante comum os desenvolvedores terem de fazer tarefas repetitivas, seja ao iniciar projetos ou durante todo seu ciclo de desenvolvimento. Muitas delas são recorrentes, tendo de ser executadas a todo momento, por exemplo, compilar Sass para CSS.

Felizmente, é possível dar um passo além na eficiência e produtividade front-end por meio dos automatizadores de tarefas, mais conhecidos como **Task Runners**.

8.1 O que são Task Runners?

Como já citado, task runners são ferramentas que se destinam a automatizar tarefas variadas que devem acontecer em determinados momentos do desenvolvimento. Isso vale também para o back-end, apesar de não ser o foco deste livro.

Tarefas como compilar CSS de pré-processadores, verificar erros de JavaScript, combinar e minificar *assets*, otimizar imagens, e muitas outras podem ser automatizadas! Sim, elas podem ser executadas pelos task runners de maneira transparente e indolor, fazendo com que a produtividade das pessoas envolvidas no projeto atinja limites nunca antes imaginados!

Eles permitem que se criem tarefas que podem ser chamadas e executadas através da linha de comando, além de ficarem rodando em background, só esperando você salvar determinado tipo de arquivo, para dar início à execução de uma série de instruções automatizadas, previamente programadas!

Parece bom demais para ser verdade, mas realmente é!

Grunt e Gulp

De maneira semelhante aos pré-processadores CSS (), no mundo da automatização de tarefas, também há dois task runners que são usados pela maioria das pessoas:

- Grunt (http://gruntjs.com/) ;
- Gulp (http://gulpjs.com/) .

Na prática, ambos podem executar as mesmas tarefas e contam com quase a mesma gama de possibilidades. O que realmente muda é a maneira como se escreve e como se dão as instruções, para que cada um faça o que deve fazer.

Grunt tem sintaxe mais parecida com um arquivo de configuração, sendo mais verboso e, consequentemente, gerando arquivos maiores. **Gulp**, por sua vez, remete mais a quem gosta de dar instruções, de programar mesmo, tendo como características principais uma sintaxe mais enxuta e performance espantosa. Estamos falando de **microssegundos** e, caso não saiba, **1s = 1.000.000s**!

O Gulp baseia-se em alguns princípios básicos:

- Fácil de usar: ao preferir código, em vez de configuração, ele mantém simples o que é simples e faz tarefas complexas bem mais administráveis;
- Eficiente: Gulp usa o poder de *streams* de `Node.js`, o que permite builds muito mais rápidos e dispensa a necessidade de gravação de arquivos intermediários em disco;
- Alta qualidade: as diretrizes estritas para plugins do Gulp garantem que eles permaneçam simples e trabalhem da maneira que se espera;
- Fácil de aprender: com uma API mínima, aprende-se a trabalhar com ele quase que na hora!

Código em vez de configuração (*code over configuration*) significa que realmente é preciso programar, montar scripts.

A eficiência do Gulp muito se dá em função de este usar streams do Node (http://goo.gl/gGcxVW) . Tecnicamente, stream é uma interface abstrata implementada por vários objetos. Porém, pense neles como os pipes de sistemas `*NIX`, em que é possível jogar o resultado de um comando diretamente em outro.

Sua alta qualidade e facilidade de aprendizado garantem plugins igualmente eficientes. Também, devido ao próprio modo como o Gulp funciona, essa facilidade de aprender e usar são absurdas; é uma curtíssima (quase inexistente) curva de aprendizado para começar a mexer com uma das mais poderosas ferramentas de desenvolvimento front-end que surgiram nos últimos tempos!

Em virtude dessas características e do meu gosto pessoal, até o fim do capítulo, será focado no Gulp. Entretanto, lembre-se de que é possível realizar tudo o que será mostrado também com o Grunt, alterando e adaptando uma coisa aqui e ali.

8.2 Como instalar o Gulp

Gulp é JavaScript com Node. A primeira coisa que você precisará é instalar o `Node.js` instruções em http://nodejs.org/. Tomando por base que você está

mexendo em um sistema de gente grande, isso é tão simples quando digitar uma linha de comando no terminal (caso já até não o tenha instalado). Depois disso, instale o Gulp globalmente no sistema com:

```
npm install -g gulp
```

Já que estamos lidando com Node, um arquivo `package.json` na raiz do projeto faz-se necessário para indicar quais módulos serão usados. É possível que o Node o crie automaticamente com `npm init`, o que iniciará um prompt interativo que perguntará um monte de coisas (tais como: nome do projeto, versão, descrição, licença, autor etc.). Na verdade, para começar a adicionar módulos, só é preciso que esse arquivo seja um JSON válido (mais sobre JSON em http://goo.gl/bofIoU) . Um JSON válido mínimo não passa de `{}`. Caso queira ser mais prático, basta executar:

```
echo "{}" > package.json
```

Já com o arquivo `package.json` na raiz do projeto com um JSON válido, execute:

```
npm install --save-dev gulp
```

Com isso, o Gulp já está instalado e pronto para ser usado no projeto.

Revisando:

- `npm install -g gulp`
- `echo "{}" > package.json`
- `npm install --save-dev gulp`

Depois disso, seu `package.json` deve estar assim (logicamente, a versão pode se alterar):

```
{
  "devDependencies": {
    "gulp": "~3.8.10"
  }
}
```

Como você viu, para instalar módulos, o comando é `npm install --save-dev [MÓDULO]`. Na verdade, a instalação propriamente dita dispensa o uso de `--save-dev`, mas usar esse parâmetro garante que a definição do módulo seja gravada no `package.json`. Isso torna as coisas mais fáceis ao se iniciar um projeto que usa Gulp.

> Caso esteja usando controle de versão, não é preciso incluir o diretório `node_modules`, já que, ao se clonar um projeto que tenha um `package.json`, basta executar `npm install` para que ele seja criado automaticamente junto com os módulos/ dependências que serão usados.

gulp□le.js

Agora que o básico para se mexer com Gulp está preparado, é hora do `gulpfile.js`. Nesse arquivo, que também deve ficar na raiz, é onde os scripts das tarefas instaladas constam. É aqui, efetivamente, que os comandos sobre o que fazer e como fazer são passados para o Gulp.

Um *boilerplate* para o `gulpfile.js` poderia ser:

```
var gulp = require('gulp');

gulp.task('default', function() {
    // Tarefas
});
```

Primeiramente, requere-se o módulo `gulp`. Isso deve ser feito com todos módulos que você pretende usar. Depois, cria-se uma tarefa `default` com as instruções que serão realizadas com sua execução. Do jeito que está, você já pode executar no terminal: `gulp`.

Apesar de ser uma tarefa nada útil, ela já deve retornar algo como:

```
[gulp] Using file [...]/gulpfile.js
[gulp] Working directory changed to [...]
[gulp] Running 'default'...
[gulp] Finished 'default' in 85 s
```

Atente-se à maneira de executar uma tarefa, que é usar `gulp` seguido de seu nome. Por exemplo, se você tivesse criado uma tarefa `concatAssets`, você a executaria com `gulp concatAssets`. No caso, funcionou executando somente `gulp`, porque foi usado um nome de tarefa especial. Quando uma tarefa `default` está definida, é ela que será executada caso nenhum outro nome seja passado.

8.3 Gulp API

Já está claro que um dos princípios norteadores do Gulp é a simplicidade. Indo ao encontro disso, sua API é extremamente simples e enxuta, limitando-se a quatro funções principais:

- `src()`: arquivo(s) que entrará(ão) na sequência de pipes para serem tratados/manipulados;
- `task()`: define tarefas no Gulp;
- `dest()`: destino(s) do(s) arquivo(s) que passou pelos pipes;
- `watch()`: observa arquivo(s) e faz alguma coisa quando esse(s) é(são) alterado(s).

Obviamente, cada um tem suas opções, possibilidade e detalhes técnicos. Apesar de essas breves palavras sobre cada já darem uma noção sobre as respectivas funções, seria interessante olhar a página da API do Gulp (http://goo.gl/7cviok) .

8.4 Exemplo de tarefas com Gulp

Basicamente, você já sabe como trabalhar com Gulp! A partir de agora, é usar os módulos Gulp disponíveis (e, quem sabe, até criar alguns) e conhecer as peculiaridades de uso de cada um. Um exemplo vai ajudar a clarear as coisas e fixar o conhecimento.

O Gulp, para este exemplo, vai trabalhar com os módulos:

- `gulp-ruby-sass`: para compilar Sass (http://goo.gl/Qbfohq) ;

- `gulp-autoprefixer`: para colocar automaticamente *vendor prefixes* em seletores (http://goo.gl/eCrNp5) ;
- `gulp-minify-css`: para minificar CSS (http://goo.gl/Ln6JL3) ;
- `gulp-livereload`: para recarregar a página automaticamente quando um arquivo for salvo (http://goo.gl/tK9Lak) .

Para descobrir mais plugins e módulos, acesse http://gulpjs.com/plugins/.

Usando a função `watch()` do Gulp, tudo isso vai acontecer **automaticamente** ao se salvar algum arquivo Sass.

Instalando módulos Gulp

Para instalar os módulos necessários ao exemplo:

```
npm install --save-dev gulp-ruby-sass gulp-autoprefixer
gulp-minify-css gulp-livereload tiny-lr
```

Vale lembrar que, para usar o `gulp-ruby-sass` que é só um pouquinho mais lento que `gulp-sass` (http://goo.gl/I8D18V) , mas possui opções mais interessantes , é preciso que você tenha Ruby instalado.

Também é preciso ter a extensão LiveReload para Chrome (http://goo.gl/FXPwAL) , pois o `gulp-livereload` usa `tiny-lr`. Por isso, ele consta na lista de módulos a serem instalados.

Fig. 8.1: É preciso habilitar o LiveReload no Chrome

Depois de rodar o comando, seu `package.json` deve estar parecido com:

```
{
  "devDependencies": {
    "gulp": "~3.8.10",
    "gulp-ruby-sass": "~0.7.1",
    "gulp-autoprefixer": "2.1.0",
    "gulp-minify-css": "~0.4.3",
    "gulp-livereload": "~3.6.0",
    "tiny-lr": "0.1.5"
  }
}
```

Estrutura do projeto-exemplo

Para esse projeto-exemplo, considere a seguinte estrutura:

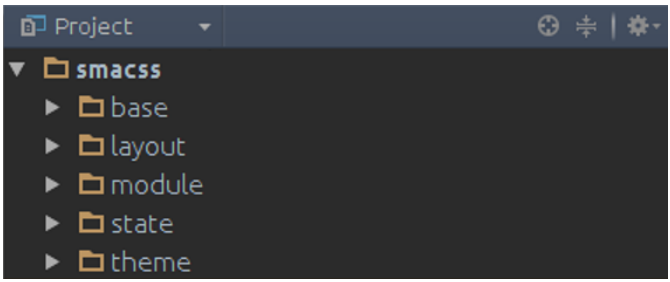


Fig. 8.2: Exemplo de estrutura de diretórios

`index.html`:

```
<!doctype html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Gulp test</title>
        <link href="style.css" rel="stylesheet">
    </head>

    <body>
        <p>Texto do teste.</p>
```

```
    </body>
</html>
```

`_variables.scss`:

```
$backgroundColor: tomato;
$defaultTextColor: #fff;
```

`style.scsss`:

```
@import 'variables';

body {
    background-color: $backgroundColor;
    color: $defaultTextColor;
}
```

Editar o gulp□le.js

Agora será preciso editar o `gulpfile.js` para que fique adequado aos propósitos do projeto-exemplo. Para começar, é preciso requerer os módulos e os atribuir a suas respectivas variáveis:

```
var gulp      = require('gulp'),
    sass      = require('gulp-ruby-sass'),
    prefix    = require('gulp-autoprefixer'),
    minifycss = require('gulp-minify-css'),
    refresh   = require('gulp-livereload'),
    server    = require('tiny-lr')();
```

Repare que, para o `gulp-livereload` e `tiny-lr`, foram dados nomes de variáveis diferentes, embora seja uma convenção nomear uma variável com o nome do módulo que ela recebe; daqui a pouco você vai entender.

Agora, chegou o momento de indicar ao Gulp o que fazer, efetivamente. Lembre-se dos pipes: o(s) arquivos(s) indicado(s) em `src()` passa(m) por um comando e o resultado disso é jogado para o próximo, e assim por diante.

```
gulp.task('compileStyles', function() {
    gulp.src('src/style.scss')
```

```
        .pipe(sass({
            noCache      : true,
            precision    : 4,
            unixNewlines : true
        }))
        .pipe(prefix('last 3 version'))
        .pipe(minifycss())
        .pipe(gulp.dest('dist'))
        .pipe(refresh(server));
});
```

Perceba que há opções definidas para `gulp-ruby-sass`. Isso é perfeitamente normal para vários módulos Gulp. Geralmente o que está disponível e quais as opções possíveis são informações disponibilizadas na própria página do módulo. Também, veja o motivo de aquelas variáveis terem recebido nomes diferentes do nome do módulo: fica mais entendível quando se lê `refresh(server)`, não é verdade?

Finalmente, a parte mágica que automatiza inclusive o recarregar de página, poupando até de ter que dar o foco na janela do navegador e apertar `F5` ou pior: clicar no botão `Recarregar`.

```
gulp.task('watch', function() {
    server.listen(35729, function( err ) {
        if ( err ) { return console.log( err ); }

        gulp.watch('src/**/*.{sass,scss}', [
            'compileStyles'
        ]);
    });
});
```

A tarefa chamada `watch` inicia um servidor LiveReload, escutando na porta 35729 (a porta padrão desse server) e colocando no console algum acontecimento inesperado. O filé mignon é a parte que indica que, caso qualquer arquivo Sass seja alterado em qualquer parte do diretório `src`, a tarefa `compileStyles` deve ser executada e a página recarregada. Afinal, é para isso que o LiveReload serve.

O arquivo `gulpfile.js` final deve parecer-se com:

```
var gulp      = require('gulp'),
    sass      = require('gulp-ruby-sass'),
    prefix    = require('gulp-autoprefixer'),
    minifycss = require('gulp-minify-css'),
    refresh   = require('gulp-livereload'),
    server    = require('tiny-lr')();

gulp.task('compileStyles', function() {
    gulp.src('src/style.scss')
    .pipe(sass({
        noCache      : true,
        precision    : 4,
        unixNewlines : true
    }))
    .pipe(prefix('last 3 version'))
    .pipe(minifycss())
    .pipe(gulp.dest('dist'))
    .pipe(refresh(server));
});

gulp.task('watch', function() {
    server.listen(35729, function( err ) {
        if ( err ) { return console.log( err ); }

        gulp.watch('src/**/*.{sass,scss}', [
            'compileStyles'
        ]);
    });
});
```

Com essas quase 30 linhas de código, agora é possível abrir `index.html`, executar `gulp watch`, ativar a extensão LiveReload no navegador, e partir para o abraço (leia-se: editar um arquivo Sass em `src`). Quase que instantaneamente, você verá as alterações aparecendo na tela de seu navegador.

Agora você entende aqueles preceitos do Gulp: **Velocidade**, **E□ciência** e **Simplicidade**.

8.5 Sugestões de plugins Gulp

Existem centenas de módulos/ plugins para o Gulp e, a cada dia, mais e mais aparecem pela web. Não seria viável manter uma lista completa, mas, certamente, algumas indicações de módulos úteis para tarefas comuns é bem-vinda.

- `gulp-util`: funções úteis variadas (http://goo.gl/fH3Qhq) ;
- `gulp-jshint`: detecta erros e problemas potenciais de JavaScript (http://goo.gl/9GG5JQ) ;
- `gulp-uglify`: minifica arquivos (http://goo.gl/iVLc8p) ;
- `gulp-imagemin`: minifica/ otimiza imagens (http://goo.gl/GT8rNO) ;
- `gulp-replace`: substitui `strings` em arquivos (http://goo.gl/tuhD8v) ;
- `gulp-concat`: concatena arquivos (http://goo.gl/UGpyfZ) ;
- `delete-files-folder`: remove arquivos e diretórios (http://goo.gl/OV2SCB) ;
- `gulp-rev`: adiciona hash aleatório em nomes de arquivos (http://goo.gl/9HTXri) .

8.6 Conclusão

Kronos é implacável, principalmente quando se trata de desenvolvimento web. Não há como evitar que novas tecnologias apareçam com novas propostas, solicitando uma nova maneira de pensar e fazer, a fim de tentar sanar problemas já conhecidos. Este é, precisamente, o caso do Gulp: com a intenção de ajudar na automação de tarefas repetitivas e liberar a mente criativa dos desenvolvedores, é uma das melhores e mais eficientes ferramentas para front-end dos últimos anos!

Aprofunde seus conhecimentos na ferramenta e certamente perceberá que, quando tiver se acostumado às facilidades e ao imenso ganho de produtividade proporcionado pelos task runners, não tem como voltar atrás!

Agora, que tal conhecer uma forma inovadora de organizar sua Arquitetura CSS por meio de uma metodologia inédita no Brasil? Conheça **ITCSS** no próximo capítulo.

Capítulo 9

ITCSS

Neste ponto do livro, você já conheceu CSS orientado a objetos (3), SMACSS (4), BEM (5), pré-processadores CSS (), CSS namespaces (7) e várias práticas e dicas sobre como escrever CSS mais eficiente.

São conhecimentos/ técnicas que podem ser usados e aplicados conforme a necessidade de cada projeto ou, melhor, como um know-how de técnicas que podem ser usadas em qualquer combinação para que você consiga levar a arquitetura CSS deles para o próximo nível.

Por falar em próximo nível, algo novo chegou ao mundo das boas práticas de CSS; algo bastante recente que, com exclusividade e em primeira mão, sendo tratada pela primeira vez em uma publicação nacional, você vai conhecer agora: **Inverted Triangle CSS** ou **ITCSS**.

Fig. 9.1: Triângulo invertido dividido em camadas

Alguns dos principais benefícios de usar ITCSS são:

- Por ser simplíssima, é uma metodologia acessível a qualquer um;
- Organiza e gerencia o código em cascata;
- Cria uma organização, na qual se sabe onde cada parte do código deve estar;
- Reduz perdas e redundância de código;
- Evita dores de cabeça com especificidade CSS (2.1).

Parece muito bom para ser verdade, mas acredite: simplesmente é!

9.2 As camadas do triângulo invertido

As 7 camadas do triângulo invertido são:

- **Con□gurações** (*Settings*): variáveis e configurações globais;

- **Ferramentas** (*Tools*): funções e mixins;
- **Genérico** (*Generic*): estilos genéricos (resets, normalizes etc.);
- **Elementos** (*Elements*): estilização de elementos HTML diretamente;
- **Objetos** (*Objects*): padrões não cosméticos (mais sobre OOCSS em 3);
- **Componentes** (*Components*): peças de UI isoladas (menu dropdown, carousel etc.);
- **Trumps**: utilitários, helpers, sobrescritas e hacks.

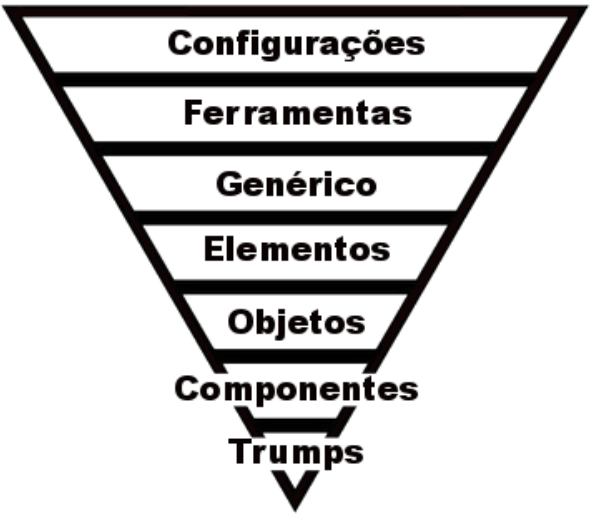


Fig. 9.2: Camadas do triângulo invertido

Perceba que *Trumps* não teve tradução. Na verdade, o próprio Harry Roberts explica que esta é uma palavra que, em inglês, é sinônimo de *beat*, que, nesse contexto, seria algo como **ultrapassar alguém ou alguma coisa por dizer ou fazer algo melhor**.

Para fins de concisão e estilo, acredito que, quando ITCSS for mais popular na Terra dos Papagaios, não haverá maiores questões em continuar usando o termo *Trumps* que é até bem simpático e agradável de se pronunciar, diga-se de passagem.

Além desta organização mais formal, ITCSS é norteado por alguns princípios gerais:

- Sem uso de IDs, somente classes;
- Criação modular de componentes, em vez de páginas;
- Uso e abuso de classes no HTML (nada de economia de caracteres aqui).

Para complementar sua robustez, ITCSS é construída em cima de 3 métricas-chave:

- **Geral para explícito:** começa-se pelos estilos mais gerais e genéricos possíveis e, a partir daí, vão-se colocando camadas de estilo adicionais;
- **Baixa Especi□cidade para alta Especi□cidade:** regras com especificidade menor aparecem em camadas mais genéricas; regras de maior, em camadas mais específicas/ focadas;
- **Muito alcance para pouco alcance:** as regras presentes em camadas mais genéricas afetam grande parte do DOM e, à medida que vão se afunilando, alcançam cada vez menos porções.

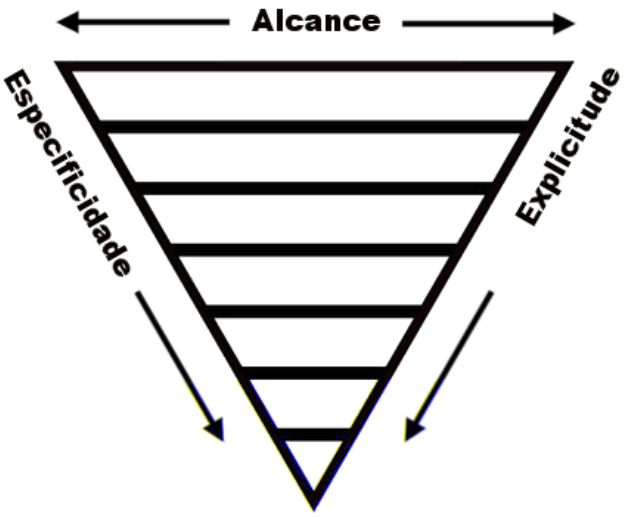


Fig. 9.3: Métricas-chave do triângulo invertido

Como será constatado a seguir, construir o CSS baseado nessas métricas-chave garante que questões com especificidade sejam mitigadas e que o código seja escrito em uma ordem lógica e progressiva. Como se verá, isso traz grande capacidade de extensão e menos redundância.

Se pudesse representar a relação **especi□cidade x localização na folha de estilos** por meio de um gráfico para um projeto comum, este seria mais ou menos assim:

Fig. 9.4: Especificidade por localização em projetos sem ITCSS

Já usando ITCSS e todas as vantagens que sua arquitetura traz:

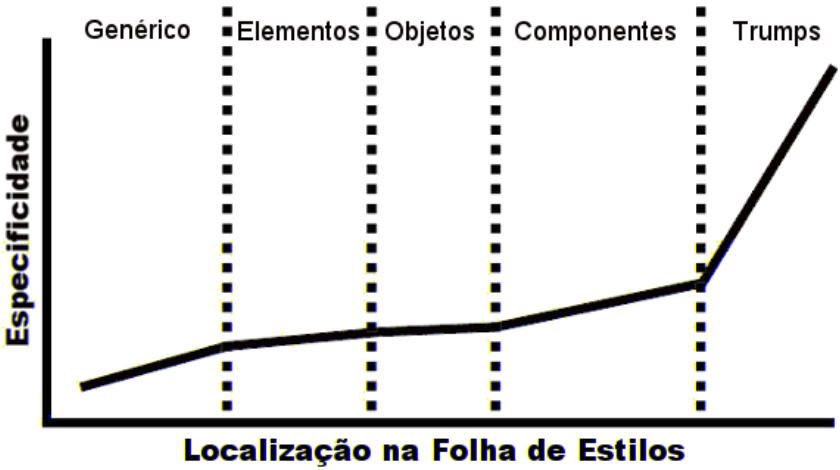


Fig. 9.5: Especificidade por localização em projetos usando ITCSS

Isso ilustra nitidamente as vantagens que estão ao seu alcance e que você aprenderá agora, sabendo mais sobre cada uma das camadas de ITCSS.

Con□gurações (settings)

Em ITCSS não é obrigatório que se faça uso de pré-processadores CSS. Porém, devido a todas as vantagens já vistas no capítulo 9, é extremamente aconselhável que você use. Neste caso, é na camada **con□gurações** que você começará o trabalho.

Quando pensar na camada Configurações, pense no termo *estilos globais*, já que é exatamente onde esse tipo de estilo deve estar.

Exemplo:

```
$environment: dev;

$rem-base: 16px !default;

$main-color: #c00fee;

$module-size: rem-calc(20px);
```

Ferramentas (tools)

A camada **Ferramentas** deve conter funções e mixins olha os pré-processadores CSS aí! relativos ao projeto. Essa camada vem logo após a de Configurações, porque pode ser necessário que alguma função ou mixin faça uso de uma configuração global qualquer.

Exemplo:

```
@function strip-unit($num) {
    @return $num / ($num * 0 + 1);
}

@function convert-to-rem($value, $base-value: $rem-base) {
    $value: strip-unit($value) / strip-unit($base-value) * 1rem;

    @if ($value == 0rem) { $value: 0; } // Turn 0rem into 0

    @return $value;
}
```

```
@function rem-calc($values, $base-value: $rem-base) {
    $max: length($values);

    @if $max == 1 {
        @return convert-to-rem(nth($values, 1), $base-value);
    }

    $remValues: ();

    @for $i from 1 through $max {
        $remValues: append(
                        $remValues,
                        convert-to-rem(
                            nth($values, $i),
                            $base-value
                        )
                    );
    }

    @return $remValues;
}
```

Genérico (generic)

Na camada **Genérico** é que o CSS que efetivamente vai entrar em ação o resultado da compilação do pré-processador, diga-se dessa forma começa a ser escrito. Aqui é onde se definirão eventuais *CSS resets*; onde se coloca um `Normalize.css` (http://goo.gl/YFPLMO) ; e onde um `box-sizing` geral é definido.

Essa camada afeta a imensa maioria do DOM, por isso, deve ser mais ampla e genérica, estando mais para a base do triângulo invertido.

Exemplo:

```
* {
    -webkit-box-sizing: border-box;
       -moz-box-sizing: border-box;
            box-sizing: border-box;
}
```

Elementos (elements)

Na camada **Elementos**, consta a estilização de elementos HTML, propriamente ditos. Seguindo na metodologia do ITCSS, é uma camada ligeiramente mais específica que comporta estilizações de elementos HTML diretamente, conforme as necessidades do projeto.

Devido ao tipo de estilização que há nela, geralmente depois de definidos os estilos, não é necessário que se mexa mais com eles, ficando essa tarefa para as camadas mais específicas do triângulo.

Exemplo:

```
img,
iframe,
object,
embed,
video {
    height: auto;
    max-width: 100%;
}
```

Objetos (objects)

A camada **Objetos** comporta estilos não cosméticos/ abstratos, remetendo mais a OOCSS (mais informações em 3), grids, contêineres e outros desta natureza.

Exemplo:

```
.media {
    margin: 10px;

    &, .bd {
        overflow: hidden;
        _overflow: visible;
        zoom: 1;
    }

    .img {
        float: left;
```

```
        margin-right: 10px;

        &--ext {
            float: right;
            margin-left: 10px;
        }
    }

    .img img {
        display: block;
    }
}
```

Componentes (components)

A camada **Componentes** é a que comporta o maior número de estilizações, já que nela devem constar os códigos dos elementos de UI do projeto (menus, carousels, slider etc.). Para facilitar o entendimento, faça um paralelo com os Módulos de SMACSS (4.5).

Seguindo a proposta do ITCSS, essa é uma camada mais específica, que faz uso de tudo o que foi definido nas anteriores, tendo um nível de especificidade maior.

Trumps

Para ter-se um código totalmente elegante e formal, a camada Componentes poderia ter sido considerada a última. Porém, como "na prática, a teoria é outra", ITCSS usa a camada **Trumps** como uma maneira de vencer (*beats*) os estilos das anteriores (se necessário), tornando-se o último bastião de estilização dentro da metodologia.

Esta camada contém utilitários, helpers, sobrescritas e hacks e, devido ao propósito pelo qual existe, não se acanhe em usar `!important` se necessário. Trumps é a ponta do triângulo e, em decorrência disso, a camada com as regras mais específicas, explícitas e focadas.

Exemplo:

```
.half-width {
    width: 50% !important;
```

```
}
```

9.3 Organização do código com ITCSS

Como foi possível perceber, a estrutura triangular invertida de ITCSS permite a organização do código, não mais por função estilos para tipografia, para formulários etc. , agora, organiza-se por **explicitude** e **especi□cidade**.

Em função da organização de ITCSS, a hierarquia, a cascata e a especificidade CSS trabalham a favor do desenvolvedor, já que cada camada somente dá prosseguimento ao que a anterior já definiu!

Como sugestão do próprio criador de ITCSS, recomenda-se que, na prática, cada camada seja formada por uma série de partials (Smacss) com a seguinte convenção de nome: `_<camada>.<partial>.scss`.

Exemplos:

```
_settings.colors.scss
_elements.headings.scss
_components.tabs.scss
```

Cada partial deve ser o mais granular e específica possível, contendo somente o código necessário para cumprir a função a que se destina. Por exemplo, um arquivo `_elements.headings.scss` deve conter tão somente regras para os títulos (h1-h6) e nada mais. Caso haja, por exemplo, uma seção em que títulos e subtítulos sejam apresentados de maneira diferente, isso poderia constar em um outro arquivo, como `_components.page-title.scss`, por meio de classes como `.page-title` e `.page-title-sub`.

Essa é a maneira como ITCSS funciona e para a qual sua estrutura de triângulo foi concebida: não se colocam todos os estilos de cabeçalhos juntos; colocam-se os estilos referentes a elementos juntos e tudo o que é baseado em classes em um outro agrupamento! Com isso, organiza-se o projeto em métricas úteis de CSS e não em blocos temáticos, que, na maioria das vezes, trazem questões de cascata e especificidade.

Na prática, usando Sass, todas as camadas funcionam juntas por meio da importação desses vários arquivos respeitando-se a ordem do triângulo invertido, obviamente:

```
@import
    'settings.colors',
    'settings.global',

    'tools.functions',
    'tools.mixins',

    'generic.box-sizing',
    'generic.normalize',

    'elements.headings',
    'elements.links',

    'objects.grid',
    'objects.wrappers',

    'components.buttons',
    'components.carousel',
    'components.main-nav',

    'trumps.clearfix',
    'trumps.ie8'
    'trumps.utilities',
;
```

Dentro da estrutura de camadas de ITCSS, você pode usar a ordem de importação que quiser, desde que sejam obedecidos os seguintes critérios de similaridade:

- **Especi□cidade:** todos os seletores de elementos ou de classes ou todos utilitários com `!important` etc.;
- **Explicitude:** estilizar todos os elementos base HTML ou de UI etc.;
- **Alcance:** capacidade de afetar todo o DOM (por exemplo, `* {}`); partes do DOM (exemplo, `a {}`); uma seção do DOM (`.carousel {}`); ou um nó específico do DOM (`.clearfix {}`).

9.4 Conclusão

Como visto, ITCSS é uma nova maneira de pensar e organizar a arquitetura CSS de seus projetos. Sua estrutura permite parar de ordená-los por temas e passar a usar métricas úteis de CSS. Para isso, ele é organizado funcionalmente em camadas que aproveitam a hierarquia, cascata e especificidade CSS.

Como qualquer metodologia, não se trata de regras rígidas nem conceitos escritos em pedra. Caso tenha gostado, mas gostaria de alterar uma ou outra coisa, sinta-se à vontade! Por exemplo, em vez de usar a estrutura de nomes de partials indicada, você pode querer organizar tudo em subpastas com o nome do arquivo simplificado:

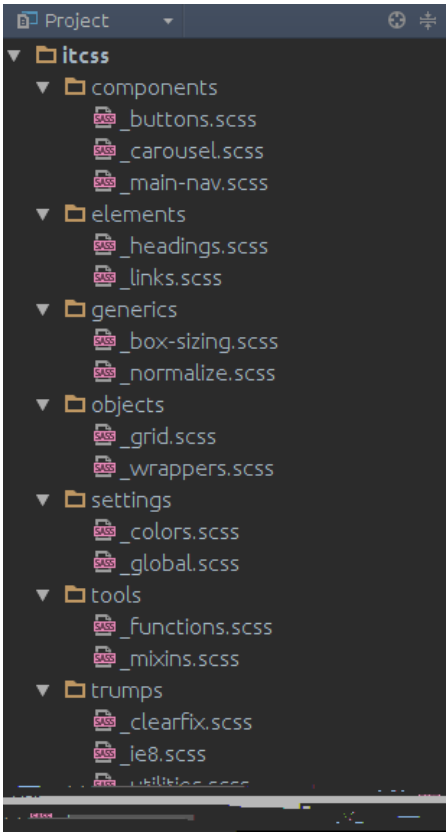


Fig. 9.6: Organize os projetos como for melhor para você e sua equipe: não há normas rígidas

Isso, obviamente, alteraria a importação dos partials em seu `style.scss`:

```
@import
    'settings/colors',
    'settings/global',

    'tools/functions',
    'tools/mixins',

    'generics/box-sizing',
```

```
    'generics/normalize',

    'elements/headings',
    'elements/links',

    'objects/grid',
    'objects/wrappers',

    'components/buttons',
    'components/carousel',
    'components/main-nav',

    'trumps/clearfix',
    'trumps/ie8'
    'trumps/utilities',
;
```

Perceba que o `generic` também foi alterado para `generics`. Uma adaptação meramente material ao gosto de quem está usando, que deixa a proposta de arquitetura CSS mais personalizada sem se abster dos benefícios que ela oferece!

A metodologia foi criada para ajudar, não para atrapalhar. Portanto, teste, experimente, use, reúse e adapte-a conforme seja necessário, sempre aproveitando o que de melhor ITCSS tem a oferecer!

Capítulo 10

O fim

Este é o fim. O fim do início de seus estudos sobre como escrever CSS melhor e começar a usar uma arquitetura CSS decente!

Se leu o livro na íntegra e prestou atenção, agora você já conhece mais a fundo sobre seletores CSS (2), CSS orientado a objetos (3), SMACSS (4), BEM (5), pré-processadores CSS (9) e o novíssimo ITCSS (9).

Para os acostumados a fazer só o feijão com arroz de CSS, são evoluções bastante consideráveis, que, sem sombra de dúvidas, farão com que você passe para o próximo nível nos conhecimentos e técnicas de front-end!

Você não precisa usar tudo o que foi visto de uma vez. Faça suas experimentações. Teste usar algumas das técnicas/ metodologias em um projeto, depois troque para outras em outro. A experimentação trouxe a humanidade até aqui e você pode ser aquele que ajudará o mundo a dar um passo além!

Aproveite que adquiriu este livro e além de sugerir aos colegas que também façam o mesmo , volte aos capítulos e faça consultas quantas vezes forem

necessárias. Siga os links e sugestões mostrados e sempre continue estudando!

Algo a se levar em consideração, que também ajuda a produzir **CSS e□ciente**, é tentar manter o código fiel a um **guia de estilos**. Isto é, uma série de orientações de escrita de código que deve ser seguida por aqueles que mexem nos estilos do projeto. Existe uma máxima quanto a isso: **O código de um projeto deve parecer que foi escrito por uma só pessoa**. Seguir um guia de estilo ajuda bastante quanto a isso, tenha certeza!

Guias de estilos facilmente poderiam ter sido um capítulo incluído neste livro, mas como são muitos e bastante variados para CSS e para pré-processadores CSS , é mais conveniente que você os encontre em uma busca e decida por aqueles que se adequem mais à sua maneira e a seus hábitos de codificação.

Para começar, dê uma olhada no Idiomatic CSS (http://goo.gl/gLJI8e) e no artigo *CSS Style Guides* (http://goo.gl/HVTY5K) , que conta com referências a alguns notórios, como GitHub, Google e WordPress.

Não deve ser nenhuma surpresa que as técnicas e conhecimentos mostrados aqui podem ser tão dinâmicos e mutáveis quanto a própria web! Ademais, que existem muitos mais conceitos/ metodologias por aí que, apesar de bem parecidos com o que foi mostrado durante todo o livro, também possam ser dignos de sua atenção como Atomic Design (http://goo.gl/d7BJQs) ou Organic CSS (http://goo.gl/PHQCJJ) .

Logo, nada de achar que agora você é o Sr. (ou Sra.) front-end; você está apenas dando continuidade à jornada. A partir dessas informações e somente com **muita prática**, você conseguirá efetivamente dar os próximos passos na melhoria e na evolução de conhecimentos sobre front-end para se tornar um profissional melhor.

Como diria *Morpheus*, **há uma diferença entre conhecer o caminho e trilhá-lo**.

Bons estudos!