CSS MODULAR

Uma abordagem na metodologia BEM

Jackson Camara

1ª edição

CSS MODULAR

Jackson Neri da Camara

Dedico este trabalho a minha amada filha Louise e esposa Daiana por todo amor e carinho. Vocês deram um sentido especial a minha vida, me proporcionando grandes momentos de alegria.

As únicas limitações são aquelas que estabelecemos em nossa mente.
Napoleon Hill.

Sumário

Isenção de Responsabilidade

Este livro é baseado na metodologia BEM, porém algumas abordagens podem ser diferentes de acordo com a visão do autor desta obra. Essas diferenças são para simplificar a escrita de código CSS, resolver alguns problemas não abordados e simplificar algumas complexidades da metodologia BEM. Caso você queira seguir à risca a metodologia BEM, aconselho você ir direto na documentação.

Não há uma solução mágica que resolva todos os problemas para estruturar o seu código CSS de forma organizada. Mas a metodologia BEM aplicada juntamente com algumas abordagens apresentada neste livro, podem resolver a maior parte dos seus problemas ao estruturar o seu código CSS.

Este livro está na fase de desenvolvimento, portanto, não está completo e também não foi revisado. A distribuição desta obra é gratuita, com o intuito de ajudar e receber feedbacks dos nossos leitores. Fique atento em nosso site e verifique sempre que possível seu e-mail, que em breve teremos mais versões atualizadas. Caso você queira baixar este e-book atualizado ou assinar a newsletter acesse www.jacksoncamara.com.br.

Para quem foi escrito este livro?

Este livro foi escrito para os desenvolvedores front-end de websites e aplicativos que utilizam HTML e CSS. Principalmente para aqueles que desejam criar interfaces escaláveis para qualquer tamanho da equipe.

Este livro parte do princípio que o leitor possui conhecimentos básicos em HTML e CSS. E deseja, aperfeiçoar a estrutura dos seus projetos através de técnicas modulares. Os conceitos e o entendimento das técnicas modulares para o desenvolvimento de interfaces são uma poderosa ferramenta para criação de interfaces manutenível.

Este livro vai ensinar você a estruturar seu código CSS de forma modularizada. Tenho certeza que num futuro próximo você vai me agradecer pelas técnicas apresentada aqui. Seja bem-vindo e boa leitura!!!

Sobre o Autor

Jackson Neri da Câmara

é graduado em Ciência da Computação pela Universidade do Vale do Itajaí – UNIVALI. É proprietário do portal de conteúdo **www.jacksoncamara.com.br** onde são divulgados cursos e artigos sobre programação e outros temas relacionados a Ciência da Computação.

Começou sua jornada profissional, trabalhando em gráficas e Agências de Publicidade quando tinha 12 anos. Mas descobriu sua verdadeira paixão no ano de 2012, quando fez seu primeiro site utilizando a plataforma Magento (se vale dizer isso) com ajuda de muitos plug-ins para criar um e-commerce para sua esposa. Embora o site tenha funcionado, ocorreram uma série de problemas pela falta de conhecimento em programação. Desde então, percebeu que o conhecimento em programação seria necessário para desenvolver um website profissional, que não poderia continuar usando somente plataformas e plug-ins.

Seus estudos começaram na área de desenvolvimento de websites com as linguagens HTML, CSS, JavaScript, PHP e MySQL. Após dois anos estudando através de cursos online, começou a faculdade de Ciência da Computação, aprendendo linguagens como Java, C, C++, muito cálculo e algoritmo. A faculdade de Ciência da Computação proporcionou-lhe um conhecimento amplo sobre a área, porém, seu foco sempre foi a programação.

Embora todo aprendizado que a faculdade propicia, não te ensina profundamente as linguagens de programação e outras tecnologias para o mercado de trabalho. Paralelamente a faculdade se especializou em Angular, .net Core, C#, Web API, PostgreSQL e AWS. Atualmente, são as principais ferramentas que utiliza para desenvolver projetos profissionais.

Por saber o quanto é difícil achar conteúdos que ensinam a criar projetos profissionais, sentiu-se obrigado a compartilhar seus conhecimentos com aqueles que almejam criar aplicações reais. Com este objetivo resolveu criar o portal www.jacksoncamara.com.br para compartilhar seus conhecimentos. Jackson também é um grande entusiasta em microsserviços, DDD, SaaS, meta-heurística e apaixado por fotografia, sua filha e esposa.

Introdução

Como você estrutura seu código determina se você pode fazer alterações com segurança no futuro sem efeitos colaterais indesejados. Isso começa com uma compreensão do CSS modular, que possui uma coleção de princípios para escrever código com desempenho e manutenção escalável.

A ideia de escrever código modular não é nova na ciência da computação, mas os desenvolvedores só começaram a aplicá-lo ao CSS nos últimos anos. Como os sites e aplicativos da Web ficaram maiores e mais complexos, tivemos que encontrar maneiras de gerenciar a crescente complexidade das folhas de estilo.

Á medida que as folhas de estilo crescem com o código CSS acoplado ao contexto, cria-se os problemas compostos. Estes problemas são conhecidos pelo alto grau de acoplamento entre códigos, causando uma série de dificuldades durante o desenvolvimento e a manutenção do código. Uma simples exclusão do código antigo torna-se insegura porque é difícil mensurar até onde aquele código está acoplado com outro. Além do mais, código acoplados são difíceis de entender, até mesmo em projetos pequenos.

Para aplicar a modularização ao código CSS, precisamos encapsular o código CSS em folhas de estilo. De modo que o código de uma folha de estilo não possa interferir em outra folha de estilo.

A metodologia BEM é utilizada nesta obra como a metodologia principal para criar componentes modulares. Esta, é amplamente utilizada pela comunidade e possui uma excelente documentação.

Não podemos prever, o que nossas páginas precisarão no futuro. Mas com módulos reutilizáveis, você deixa a porta aberta para misturar e combinar novas possibilidades com uma aparência familiar e consistente.

1. Conceitos Básicos

Neste capítulo é apresentado alguns conceitos básicos que são utilizados na computação em diversas áreas. Entender esses conceitos é primordial para desenvolver códigos concisos em diferentes linguagens de programação, que também se aplica perfeitamente a linguagem de estilização CSS (Cascading Style Sheets). Não será feito um estudo aprofundado nesses conceitos, aconselho que você busque materiais alternativos em relação a estes assuntos.

1.1. Metodologias Modulares CSS

A modularização torna o código legível, onde qualquer pessoa inserida no projeto poderá entendê-lo. Com esta abordagem códigos legados podem ser facilmente removidos sem causar maiores dores de cabeça.

Quando se trata de CSS a escrita parece ser muito simples, mas há uma série de armadilhas. A experiência nos mostra que uma das tarefas mais árduas da computação é a de nomear as coisas. Definir quais serão nossos seletores é o que fazemos o tempo todo quando escrevemos CSS. Esta tarefa envolve organização, padronização, reutilização e uma série de outras disciplinas.

Existem várias metodologias que visam reduzir os problemas na estruturação do código no CSS, que organizam a cooperação entre programadores e mantem grandes bases de códigos CSS. Algumas dessas metodologias são: OOCSS, SMACSS, SUITCSS, Atomic e BEM. A seguir abordaremos a metodologia BEM juntamente com outros princípios para estruturar nosso código.

1.2. Princípio da Responsabilidade única

Em seu livro, *Código Limpo*, Robert C. Martin diz: “A primeira regra das classes é que devem ser pequenas. A segunda regra das classes é que devem ser menores que isso”. Ele estava falando das classes em programação orientada a objetos na época, mas o mesmo princípio se aplica muito bem aos módulos em CSS.

Seus módulos devem ser responsáveis por uma coisa. Quando um módulo tenta fazer mais de uma coisa, você deve considerar dividi-lo em módulos menores.

Ao descrever a responsabilidade de um módulo, considere se está potencialmente descrevendo mais de uma responsabilidade. Se você estiver, defina blocos separados para cada responsabilidade, aplicando o *Princípio da Responsabilidade Única*. Assim você manterá cada módulo pequeno, focado e mais fácil de entender.

1.3. Princípio do Encapsulamento

O princípio do encapsulamento trata do agrupamento de funções e dados relacionados para compor um objeto. Um dos benefícios do encapsulamento é a ocultação do estado ou valor de um objeto para que as partes externas não possam operá-lo. A única parte que fica visível é apenas a interface do componente, os detalhes da implementação do módulo não são visíveis.

Além do benefício de ocultar os detalhes dos objetos, outro benefício é a modularidade. A modularidade permite que o código possa ser mantido independentemente do código de outros objetos (desacoplamento). Como não dependem de outros objetos, cada objeto pode ser utilizado livremente no sistema.

Com o encapsulamento um objeto torna-se isolado de tal forma, que um objeto não consegue interferir em outro objeto, podendo ser inserido em qualquer contexto sem maiores problemas. Este conceito é muito utilizado na programação orientada a objeto, mas devemos utiliza-lo ao implementar código CSS modular.

Ao implementar este princípio em nossos projetos CSS, criamos módulos que não possam interferir em outros módulos, ou seja, folhas de estilos que não interferem em outras folhas de estilos. Sem dúvida esta abordagem torna a aplicação mais flexível, facilitando futuras modificações e implementações.

2. Metodologia BEM

Quando você cria sites menores, a forma como você organiza suas folhas de estilos CSS geralmente é um pequeno problema. No entanto, quando se trata de projetos maiores e mais complexos, a forma como você organiza seu código se torna crucial. A estrutura do código é ainda mais importante se você estiver trabalhando em uma equipe composta por vários desenvolvedores. Para resolver este problema muitos desenvolvedores utilizam a metodologia BEM.

BEM é a sigla para Bloco, Elemento e Modificador que é uma metodologia que permite criar uma arquitetura para seu projeto baseada em módulos independentes para desenvolvimento web. A abordagem em módulos independentes divide a interface do usuário em vários componentes pequenos para montar uma estrutura maior. Esta abordagem torna interfaces complexas em interfaces simples, possibilitando reutilização de código existente sem copiar e colar. A principal ideia por trás disso é acelerar o processo de desenvolvimento e facilitar o trabalho em equipe dos desenvolvedores.

Esta metodologia foi criada pela Yandex com diretrizes simples para organizar um projeto que precisa ser desenvolvido rapidamente e mantido a longo prazo. Os principais objetivos da metodologia são:

- **Desenvolvimento rápido e resultados duradouros de padronização: o**s projetos devem ser criados rapidamente, utilizando uma arquitetura que garante sustentabilidade e longevidade para o desenvolvimento.
- **Equipes escaláveis:** adicionando pessoas na equipe, o desempenho deve melhorar. O código deve ser cuidadosamente estruturado para garantir a sua sustentabilidade ao longo do tempo e através das mudanças da equipe.
- **A reutilização de código:** código não deve ser dependente do contexto, deve ser fácil de reutilizar em outros lugares.

Se você já viu algum componente HTML desta maneira <div class= “cabecalho__pesquisa-formulario pesquisa-formulario--tema--gelo”> é BEM em ação. Realmente, as classes podem ter nomes muito longos, mas são todas legíveis e compreensíveis. Não se assuste com toda essa verbosidade, pode ter certeza que seu código ficará muito mais compreensível.

Um nome de classe BEM é representado por até três partes:

- **Bloco:** é o componente mais externo, pode conter elementos e outros blocos.
- **Elemento:** é o componente que pode existir apenas dentro do bloco, pode haver um ou mais elementos num bloco.

- **Modificador:** um bloco ou elemento pode ter uma variação, que é realizada por um modificador.

Se todos os três forem usados em um nome, seria algo como isto:

[nome-bloco]__[nome-elemento]--[nome-modificador]--[valor-modificador]

BEM divide sua aplicação em módulos organizados, baseado na estrutura de bloco, elemento e modificador que são representados pelas classes no CSS. Observe que a melhor prática é usar o BEM apenas com classes, e não Ids. As classes permitem repetir nomes quando necessário e criar uma estrutura de codificação mais consistente.

3. BEM - Bloco

Até agora falamos sobre CSS modular, mas usamos diferentes nomenclaturas para se referir a um módulo. Imagino que você já deve ter visto vários nomes, como: módulo, componente, objeto ou bloco. Na verdade, esta confusão ocorre por causa das diferentes abordagens existentes. Mas quando falamos em CSS modular, um módulo pode ser um objeto, componente ou bloco. Ou seja, todos são a mesma coisa. Mas, em alguns momentos um elemento pode ser chamado de componente. Para não haver qualquer confusão vamos chamar o módulo de bloco conforme a metodologia BEM.

Um bloco possui sua própria lógica e funcionalidade independente, não deve ser dependente do contexto. Pense em blocos como pedaços estruturais maiores do seu código. Os blocos mais comuns em todos os sites são cabeçalho, seção, artigo, barra lateral e rodapé. Mas não se deixe enganar, um rodapé pode ter diversos blocos dentro dele, como media rede social, newsletter, lista de navegação ou informações para contato. Tudo aquilo que puder ser reaproveitado em outro lugar fora do contexto rodapé, deve ser um bloco. Caso contrário deverá ser um elemento.

A parte mais difícil de escrever CSS modular é quando precisamos escolher nomes apropriados aos módulos, mas isso não é exclusivo do CSS modular, é uma tarefa árdua em toda computação. Você precisa dar ao módulo um nome que seja significativo, o contexto a qual você vai usá-lo não deve interferir no nome. Você também deve evitar nomes que simplesmente descrevam a aparência visual ou a sua função. Chamar o bloco de “caixa cinza” parece mais versátil. Porém, se mais tarde você decidir mudar a cor do plano de fundo para azul, esse nome não seria mais aplicável, você teria que renomeá-lo e atualiza-lo em todos os locais em que ele aparece no HTML.

Os blocos devem ser representados por classe, para serem reutilizáveis. Ids não devem representar os blocos por não serem reutilizáveis. O nome do bloco deve descrever sua finalidade, por exemplo: menu, botão, cabeçalho, rodapé, seção, artigo, media e etc. O nome do bloco não deve ser sobre seu estado ou aparência, por exemplo: desabilitado, vermelho, grande. Nomes compostos devem ser separados por hífen, por exemplo: pesquisa-formulario, sobre-autor-secao. Os blocos não devem influenciar sua geometria (posicionamento externo) através das propriedades margin, position, transform e etc.

Tabela 1- Exemplo de nomes para blocos

Nomenclatura	Exemplos de Nomes
nome-bloco	formulario
	newsletter
	rodape
	botao
	menu
	rede-social-media
	contato-secao
	sobre-autor-secao
	cabecalho
	logo
	barra-lateral
	alerta
	snackbar
	banner
	acordeao
	tabela-preco
	navegacao
	dropdown
	paginacao
	galeria
	thumbnail

	media
	video
	grafico

Exemplo de código para nomear blocos

▶ CSS cabeçalho.css

.cabecalho {…}

▶ HTML arquivo.html

…

Geralmente cria-se blocos pais (blocos maiores) como cabeçalho, barra-lateral, seção e rodapé conforme mostrado na figura 1. Dentro desses blocos pais podemos colocar os blocos filhos (blocos menores), isto é chamado de aninhamento de blocos. Na figura 2 temos este exemplo, os blocos menu, auth, search e logo são blocos filhos do bloco pai head.

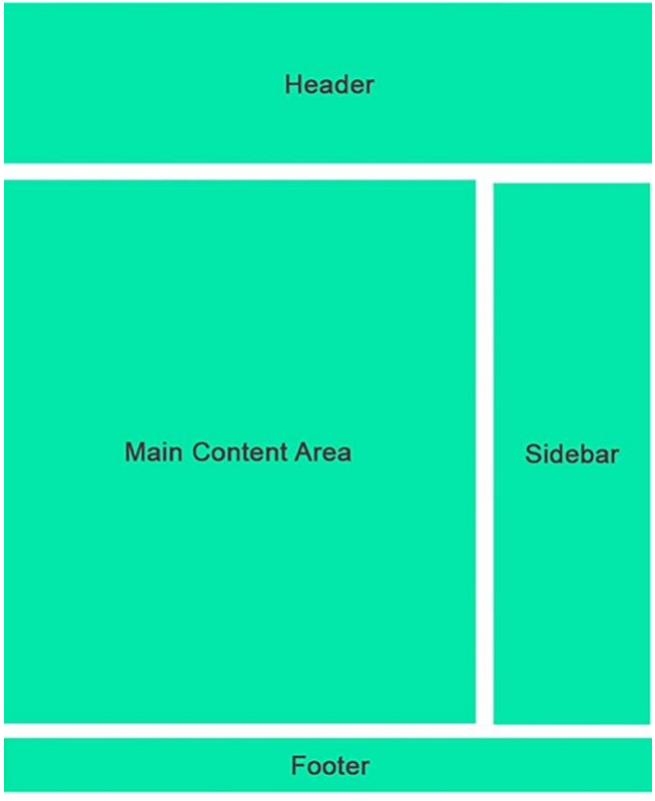

Figura 1 - Metodologia BEM blocos pais (blocos maiores)

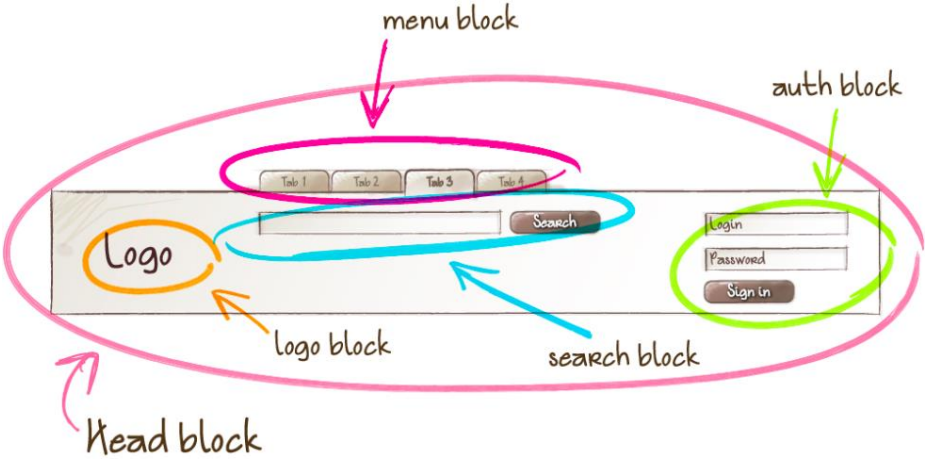

Figura 2- Metodologia BEM blocos aninhados

4. BEM - Elemento

Elementos são as partes constituintes de um bloco que não podem ser usados fora dele, funcionam como peças internas do bloco. Na figura 3 temos os elementos input e button que fazem parte do bloco search.

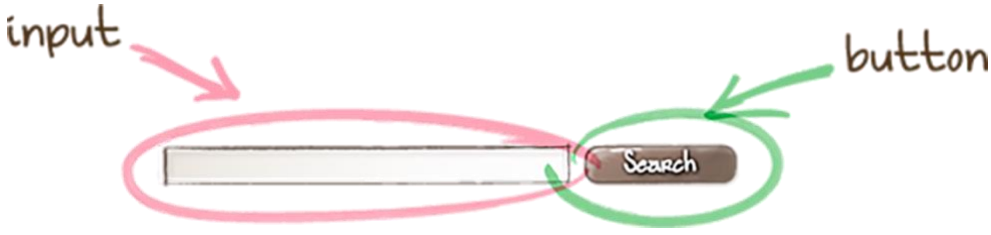

Figura 3 - Metodologia BEM elementos do bloco search

Na figura 4 temos o menu, os itens que ele contém não fazem sentido fora do contexto do menu. Portanto, você não pode definir um bloco para um item de menu, porque este item é dependente do contexto, sua existência fora do bloco não se justifica. Você deve ter um bloco para o próprio menu e os itens de menu devem ser elementos.

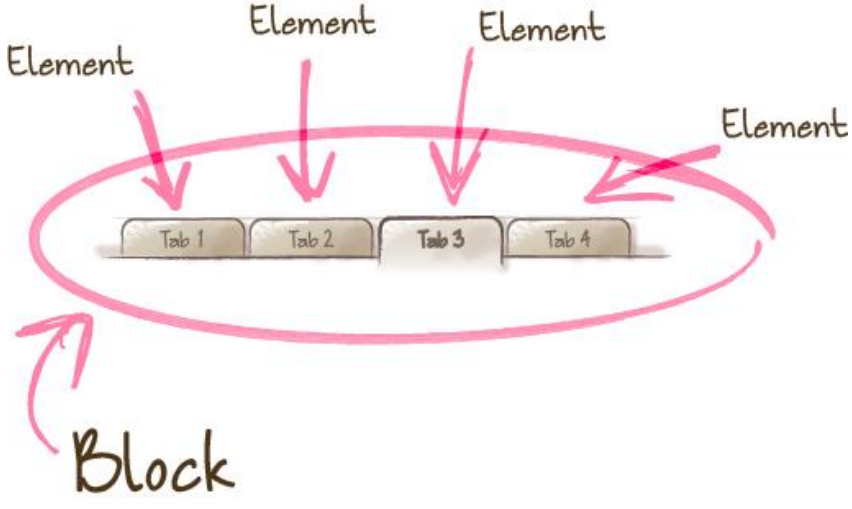

Figura 4 - Metodologia BEM elementos do bloco menu

Os elementos devem ser descritos em classes CSS, seu nome deve descrever sua finalidade, por exemplo o que o elemento é: item, texto, input, ícone, título, subtítulo e etc. Não deve descrever seu estado, por exemplo como se parece: desabilitado, vermelho ou grande.

Um elemento deve ser nomeado primeiramente com o nome do bloco a qual pertence, seguido de dois underscores (__) com o nome do elemento. Elementos com nomes compostos devem ser separados por traço.

Tabela 2 - Exemplo de nomes para elementos

Nomenclatura	Exemplos de nomes
nome-bloco__nome-elemento	menu__item
	menu__texto
	formulario__input
	formulario__textarea
	botao__icone
	sobre-autor-secao__titulo
	artigo__paragrafo

Exemplo de código do bloco menu

▶ CSS menu.css

```
.menu{...}
.menu__item {…}
```

▶ HTML arquivo.html

```
<div class=“menu>
    <ul>
        <li class=“menu__item”>Tab 1</li>
        <li class=“menu__item”>Tab 2</li>
        <li class=“menu__item”>Tab 3</li>
        <li class=“menu__item”>Tab 4</li>
    </ul>
</div>
```

Se o seu componente tiver elementos filhos em vários níveis, não tente representar cada nível no nome da classe. BEM não se destina a comunicar profundidade estrutural. Nomes de classe BEM devem incluir apenas o nome do bloco e o nome do elemento. Caso você represente de forma estrutural criará o Problema dos Seletores Netos (9.1). Em resumo um elemento faz parte de um bloco não de outro elemento.

5. BEM - Modificador

Consistência é boa, mas às vezes podemos precisar que o bloco pareça diferente em determinadas circunstâncias. Por exemplo, se você precisar exibir um botão excluir, talvez deva ser vermelho, em vez de cinza. Você faz isso definindo modificadores.

Para criar um modificador é necessário definir uma nova classe dentro do mesmo módulo. O nome do modificador deve descrever sua aparência, comportamento ou estado. O tipo estado faz parte das classes de estado que são chamados de modificadores booleanos, essa abordagem é diferente do tipo aparência e comportamento. O tipo estado é visto em detalhes em Modificadores Booleanos e Classes de Estado (5.1)

Tabela 3- Exemplo tipos de modificadores

Tipo	Modificador
aparência	tamanho
	tema
	cor
comportamento	direcao
estado	desabilitado
	focado

Os modificadores de aparência e comportamento possuem modificador e valor. Se o bloco botão possui uma classe modificadora de tamanho, deve ser representado como grande, médio, pequeno. Nunca devemos representa-lo diretamente pelo valor da sua propriedade, por exemplo 30px, 2em, 3vw ou qualquer outra unidade de medida.

Tabela 4 - Exemplo valores de modificadores

Modificador	Valor
tamanho	pequeno
	medio
	grande
cor	excluir
	salvar
	cancelar
direção	topo
	base
	esquerda
	direita

A definição da classe deve começar com o nome do bloco seguido de dois traços com o nome do modificador mais dois traços com o valor do modificador. Se for um modificador de elemento deve ser inserido o nome do elemento logo após o bloco.

Tabela 5 - Exemplo nomes de modificadores de blocos

Nomenclatura	Exemplo de nomes
nome-bloco--nome-modificador--valor-modificador	botao--tamanho--grande
	menu--tema--pilula
	selecao--direcao--topo

Tabela 6 - Exemplo nomes de modificadores de elementos

Nomenclatura	Exemplo de nomes
nome-bloco__nome-elemento--nome-modificador--valor-modificador	menu__item--tema--pilula

Os estilos de modificadores não precisam redefinir o bloco ou elemento inteiro, só precisam substituir as partes que eles mudam. Geralmente, os modificadores fazem com que algumas propriedades dos blocos/elementos sejam complementadas ou alteradas, através das propriedades background-color, font-size, borders, opacity, display, position e etc.

Para usar um modificador, adicione a classe principal do bloco e a classe modificadora a um elemento HTML. Isso aplica o estilo padrão do módulo e permite que o modificador substitua seletivamente onde necessário. O modificador não pode ser usado isoladamente do bloco ou elemento. Um modificador deve altera-lo e não substituí-lo.

No exemplo a seguir as classes botao--cor--salvar e botao--cor--excluir não substituem a classe botao, apenas modificam a propriedade background-color para alterar o plano de fundo.

Exemplo do código CSS para modificadores de bloco:

▶ **CSS botao.css**

```
.botao {
    font-size: 1em;
    padding: 0.8em 1.2em;
    border-radius: 0.4em;
    color: white;
    background-color: grey;
}


.botao--cor--salvar {
    background-color: green;
}

.botao--cor--excluir {
    background-color: red;
}
```

▶HTML arquivo.html

```
<button class="botao botao--cor--salvar">
    Salvar
</button>

<button class="botao botao--cor--excluir">
    Excluir
</button>
```

É possível criar diversos modificadores, no exemplo anterior mostramos como alterar a cor do plano de fundo de acordo com a ação que o botão representa. Mas também podemos alterar o tamanho do botão criando outros modificadores.

Exemplo código modificador de tamanho:

▶CSS botao.css

```
.botao {
    font-size: 1em;
    padding: 0.8em 1.2em;
    border-radius: 0.4em;
    color: white;
    background-color: grey;
}

.botao--tamanho--pequeno {
    font-size: 0.8em;
}

.botao--tamanho--grande {
    font-size: 1.2em;
}
```

▶HTML arquivo.html

```
<button class="botao botao--tamanho--grande">
    Salvar
</button>

<button class="botao botao--tamanho--pequeno">
    Excluir
</button>
```

Os nomes dos modificadores devem ser menos específicos e mais genéricos. Esta abordagem reduz os conflitos de nomes ao reutilizar a classe modificadora, mas o nome não pode ser tão genérico ao ponto de conflitar com outra classe. Veja os exemplos na tabela 7.

Tabela 7 - Exemplo de nomes específicos e nomes genéricos

Nomes Específicos	Nomes genéricos
botao--cor--salvar	botao--cor--sucesso
botao--cor--cancelar	botao--cor--aviso
botao--cor--excluir	botao--cor--perigo

Podemos adicionador vários modificadores aos nossos blocos ou elementos para obter a aparência desejada.

Exemplo código vários modificadores:

▶CSS botao.css

```
.botao {
    font-size: 1em;
    padding: 0.8em 1.2em;
    border-radius: 0.4em;
    color: white;
    background-color: grey;
}
```

```
.botao--cor--sucesso {
    background-color:  green;
}

.botao--cor--perigo {
    background-color:  red;
}

.botao--tamanho--pequeno{
    font-size: 0.8em;
}

.botao--tamanho--grande {
    font-size: 1.2em;
}
```

▶HTML arquivo.html

```
<button class="botao botao--cor--sucesso botao--tamanho--grande“>
    Salvar
</button>

<button class="botao botao--cor--perigo botao--tamanho--pequeno”>
    Cancelar
</button>
```

Embora não exista um limite de modificadores que possam ser adicionados em nossos blocos ou elementos, adicionar diversos modificadores pode não ser a abordagem mais apropriada. Não existe um limite padrão de modificadores que um bloco ou elemento possa ter, mas se você tiver que adicionar três ou mais modificadores deve-se pensar em criar um modificador de tema ou criar outro bloco.

Quando houver há necessidade de criar um modificador, devemos analisar se esta modificação não pode ser inserida em um modificador já existente. Caso não seja possível, deve-se criar um novo modificador.

Em alguns casos pode haver um padrão, por exemplo: o botão salvar sempre deve ser verde e maior que os demais botões, neste caso pode-se criar um tema como botao--tema--sucesso. Veja o exemplo abaixo.

Exemplo código modificador tema:

▶CSS botao.css

```
.botao {
    font-size: 1em;
    padding: 0.8em 1.2em;
    border-radius: 0.4em;
    color: white;
    background-color: grey;
}

.botao--tema--sucesso {
    font-size: 1.2em;
    background-color:  green;
}

.botao--tema--perigo {
    font-size: 0.8em;
    background-color:  red;
}
```

▶HTML arquivo.html

```
<button class="botao botao--tema--sucesso“>
    Salvar
</button>

<button class="botao botao--tema--perigo”>
    Cancelar
</button>
```

Lembre-se, você deve ter muito cuidado para não gerar o agrupamento de modificadores, isto ocorre quando adicionamos diversas alterações dentro do mesmo modificador, por exemplo: background-color, font-size, border-radius, opacity. Mais tarde você pode querer que o seu modificador não faça tantas modificações, e isso será um problema. Se houver dúvida em relação ao agrupamento desses modificadores, mantenha-os separados.

5.1. Modificadores booleanos e classes de estado

Os modificadores booleanos na metodologia BEM possuem a mesma função da classe de estado. Para refrescar a memória, as classes de estado são aquelas que tem como convenção o prefixo “is” ou “has” na palavra que descreve o estado, são muito utilizadas por programadores javascript quando pretendem adicionar ou remover uma classe CSS dinamicamente usando javascript.

São considerados classes de estado porque elas indicam algo sobre o estado atual do bloco e espera-se que mudem. Podemos citar como exemplo is-expanded, is-loading ou has-error. Os modificadores também alteram o estado dos blocos, porém não utilizam o prefixo “is” ou “has”.

Exemplos de classes de estado:

- ativo (active)
- carregado (loaded)
- carregando (loading)
- visível (visible)
- escondido (hidden)
- desabilitado (disabled)
- expandido (expanded)
- focado (focused)
- desbotado (faded)

Tabela 8 - Exemplo de nomes para modificadores de estado de bloco

Nomenclatura	Exemplos de nomes
nome-bloco--nome-modificador	menu--expandido
	bar--escondido
	botao--desabilitado

Tabela 9 - Exemplo de nomes para modificadores de estado de elemento

Nomenclatura	Exemplos de nomes
nome-bloco__nome-elemento--nome-modificador	menu__item--focado
	sobre-autor-secao__imagem--desbotado

Os modificadores booleanos devem ser usados somente quando a presença ou ausência do modificador é importante e seu valor é irrelevante. Se um modificador booleano estiver presente, seu valor é assumido como true.

Poderíamos ser facilmente induzidos ao erro de escrever bar--visivel--true ou bar--visivel--false. Porém, os modificadores booleanos não precisam do nome do modificador, seu próprio valor deve ser suficiente para descrever a modificação que será realizada. Para esconder o bloco bar, apenas precisamos inserir a palavra escondido, “bar--escondido”. Desta forma, o bloco bar não ficará mais visível, porque a presença da palavra “escondido” é assumida como verdade. Caso o padrão seja "visível" não precisamos criar a classe “bar--visivel”. Lembre-se, se um modificador booleano estiver presente, seu valor é assumido como true. Devemos usar um modificador booleano apenas para modificar o bloco ou elemento, e não para informar o que já é o padrão.

Exemplo código modificadores booleanos:

▶ CSS bar.css

```
.bar{
    font-size: 1em;
    color: white;
}

.bar--escondido{
    visibility: hidden;
}
```

▶ HTML arquivo.html

```
<i class=“bar bar--escondido”></i>
```

6. Bloco, Elemento ou Modificador

Um desafio ao usar metodologia BEM é decidir se devemos transformar o componente em bloco, elemento ou criar um modificador. Este capítulo servirá como um guia para ajudar você a tomar a melhor decisão.

6.1. Modificador ou novo bloco?

Um dos maiores problemas é decidir onde um bloco termina e um novo começa. Eu recomendo começar com modificadores, mas se o bloco está ficando difícil de gerenciar, provavelmente é hora de transformar em bloco.

Outro bom indicador é quando você descobre que precisa redefinir todo o CSS para poder estilizar seu novo modificador, isso sugere um novo bloco. Utilizar os princípios da responsabilidade única vão te ajudar a resolver esta incógnita ao escrever seus códigos.

6.2. Bloco ou Elemento?

A metodologia BEM não estabelece regras estritas para a criação de blocos e elementos. Muito dependerá das implementações específicas e preferências pessoais do desenvolvedor.

Você deve criar um bloco se uma das suas respostas for igual:

- O componente poder ser reutilizado?
Sim.

- O elemento pode ser inserido em qualquer contexto?
Sim.

- Vai depender de qualquer outro elemento externo para ser implementado?
Não.

Você deve criar um elemento se uma das suas respostas for igual:

- Pode ser implementado fora do bloco pai?
Não.

A figura 4 é uma imagem do bloco “time-secao”, vamos identificar seus componentes internos para diferenciar o que é bloco e o que é elemento. Neste contexto o bloco “time-secao” é o bloco pai de todos outros componentes que estão inseridos dentro dele.

O componente carrossel possui diversas características intrínsecas, podemos citar a quantidade de cards que o carrossel apresenta de acordo com tamanho da tela. Um celular poderá apresentar apenas um card por enquanto um notebook de 15” polegadas poderá apresentar quatro cards. Embora possua muitas pecualiaridades, o carrossel possui apenas uma única responsabilidade, apresentar os cards aos usuários. Os cards que o carrossel irá apresentar pode ser sobre o time, artigos, clientes, escritores, cursos, preços etc. Neste exemplo o carrossel apresenta o avatar -card, mas pode ser facilmente trocado pelo artigo-card e continuará funcionando do mesmo jeito. O carrossel pode ser reutilizado em qualquer outro contexto (artigo-secao, curso-secao) e possui um alto nível de complexidade, por estes motivos esse componente deve ser um bloco.

O carrossel poderá aceitar cards de diferentes contextos e continuará funcionando. Portanto, não seria prudente tornar o card um elemento, porque ficaria fortemente acoplado ao contexto do carrossel, por exemplo: carrossel_artigo-card, carrossel_curso-card, carrossel_avatar-card. Futuramente poderemos querer usar esses cards em outros carrosséis, o que exigiria uma refatoração do código. Por estes motivos o componente avatar-card deve ser um bloco.

Definir o que é bloco, elemento ou modificador vai muito além de regras, mas se você focar na reutilização dos componentes estará no caminho certo. Voltando ao avatar-card, percebemos que possui diversos componentes, como degrade, media rede social, rodapé e texto. Esses componentes dentro do avatar-card torna-o um objeto complexo, então devemos nos perguntar: Não seria mais fácil gerenciar este componente se for um bloco? Se a resposta for sim, opte por bloco.

O componente rede-social-media também nos coloca em dúvida se deve ser um elemento ou bloco. Mas precisamos fazer apenas uma pergunta. O componente poder ser reutilizado? Logicamente a resposta é sim, podemos reutilizar em rodapés, seção sobre o autor, banner principal etc. Sendo assim, rede-social-media deve ser um bloco.

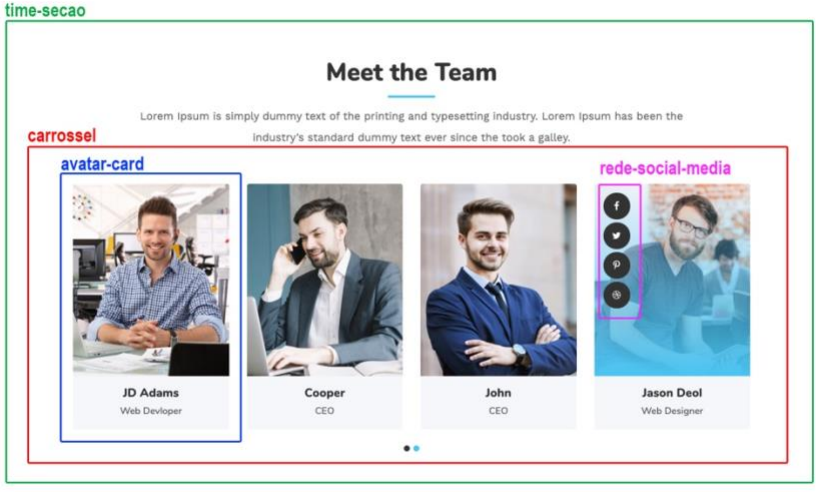

Figura 4 - Bloco time-secao

Exemplo código seção time:

▸CSS time-secao.css

.time-secao {…}
.time-secao__titulo{...}
.time-secao__barra{...}
.time-secao__paragrafo{...}

▸CSS corrosel.css

.carrossel {...}
.carrosel__ponto{...}

▸CSS avatar-card.css

.avatar-card{...}
.avatar-card__imagem{...}
.avatar-card__rodape{...}
.avatar-card__titulo{...}
.avatar-card__subtitulo{...}
.avatar-card__degrade{...}

.avatar-card__rede-social-media{...}

▶ CSS rede-social-media.css

.rede-social-media{...}
.rede-social-media__circulo{...}
.rede-social-media__icone{...}
.rede-social-media--direcao--vertical{...}

7. Convenção de nomes

O nome de uma entidade BEM é exclusivo. A mesma entidade BEM sempre tem o mesmo nome em todas as tecnologias (CSS, JavaScript e HTML). O objetivo principal da convenção de nomenclatura é dar significado aos nomes para que eles sejam tão informativos quanto possível para o desenvolvedor.

Compare o mesmo nome para um seletor CSS escrito de diferentes maneiras:

- menuitemvisible
- menu-item-visible
- menuItemVisible

Para entender o significado do primeiro nome, você precisa ler cuidadosamente cada palavra. Nos dois últimos exemplos, o nome está claramente dividido em suas partes. Mas nenhum desses nomes nos ajuda a entender que menu é um bloco, item é um elemento e visible é um modificador. As regras para nomear entidades BEM foram desenvolvidas para tornar os nomes das entidades não ambíguos e fáceis de entender.

7.1. Convenção de nomes BEM - Alternativo estilo dois traços

A metodologia de BEM fornece alguns formatos de nomes para evitar as colisões de nome. A abordagem apresentada até agora é o estilo dois traços que segue as seguintes regras:

- Os nomes são escritos em letras latinas minúsculas.
- Palavras dentro dos nomes de entidades BEM são separadas por um hífen (-).
- O nome do elemento é separado do nome do bloco por um sublinhado duplo (__).
- Os modificadores booleanos são separados do nome do bloco ou elemento por um hífen duplo (--).
- O valor do modificador é separado de seu nome por um hífen duplo (--).

Alguns podem pensar que os nomes das classes ficam muito grandes. Mas na Metodologia BEM, os nomes das classes devem ser específicos, claros, fáceis de ler e principalmente descritivos.

8. Aninhamento de Blocos

O aninhamento de blocos ocorre quando temos um ou mais blocos dentro de um bloco, podendo ter vários níveis de aninhamento.

Na figura 5 os blocos menu, auth, search e logo estão aninhados dentro do bloco pai head. Os blocos não devem determinar sua própria geometria (posição, margem, etc), esta deve ser determinada pelo bloco pai.

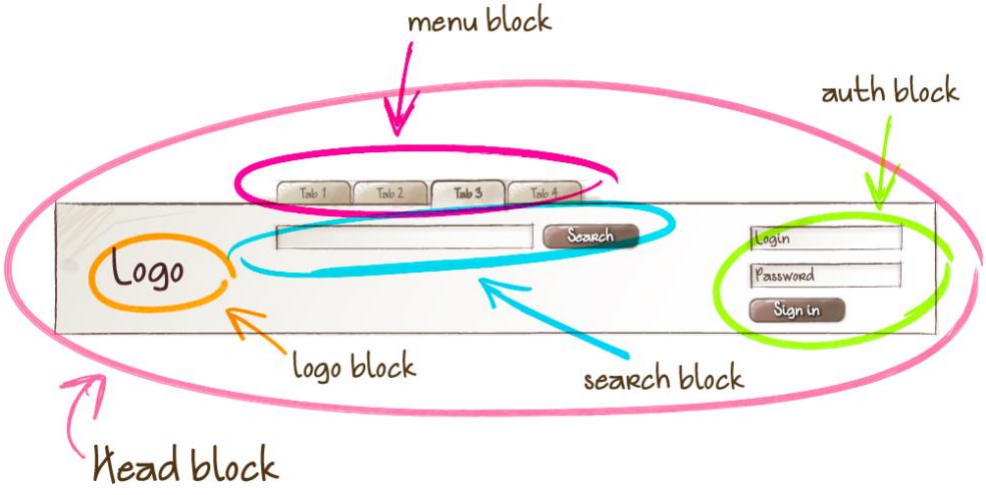

Figura 5 - Bloco head

8.1. Posicionando Blocos filhos

Na figura 5 os blocos foram definidos pelas classes menu, auth, search e logo que não especificam qualquer margem ou posição. Desta forma, podemos trocar suas posições ou reutilizá-los em outro local facilmente.

Devemos criar as classes de geometria para especificar posição e margem dos blocos, que devem ser inseridas dentro da folha de estilo do bloco pai. A nomenclatura dessas classes deve ter o nome do bloco seguido de dois underline e o nome do bloco filho.

No exemplo da figura 6 a margem e o posicionamento do bloco são definidos através das classes head__logo, head__search, head__auth e head__menu. Com esta

abordagem podemos trocar facilmente a posição do bloco logo com a posição do bloco auth através das classes head__logo e head__auth.

Figura 6 - Margem e posicionamento

Tabela 10 - Exemplo de nomes de classes para posicionamento

Nomenclaturas	Exemplo de Nomes
nome-bloco-pai__nome-bloco-filho	cabecalho__logo
	cabecalho__pesquisa
	cabecalho__autenticacao
	cabecalho__menu

Exemplo código para posicionamento de blocos filhos:

▶ CSS cabecalho.css

```
.cabecalho {...}
.cabecalho __logo{…}
.cabecalho__pesquisa{…}
.cabecalho__autenticacao{…}
.cabecalho__menu{…}
```

▶ CSS logo.css

.logo{...}

▶ CSS pesquisa.css

.pesquisa{...}

▶ CSS autenticacao.css

.autenticacao{...}

▶ CSS menu.css

.menu{...}

▶ HTML

```
<header class = “cabecalho">
   <div class = “logo cabecalho__logo”>...</div>
   <div class = “pesquisa cabecalho__pesquisa”>...</div>
   <div class = “autenticacao cabecalho__autenticacao”>...</div>
   <div class = “menu cabecalho__menu”>...</div>
</header>
```

9. Problemas conhecidos

9.1. Problema dos Seletores netos

Os componentes aninhados (componente dentro de componente) podem nos induzir a escrever nomes de classes com diversos componentes em sequência (bloco__componente__componente-aninhado__outro-componente-aninhado para representar o aninhamento desses itens no HTML. Isso pode ficar fora de controle rapidamente e quanto mais aninhado for seus componentes, mais ilegíveis se tornarão os nomes das classes.

Este problema é conhecido como seletores netos, essa abordagem pode comprometer a legibilidade do código. Para evitar este problema o padrão de sublinhado duplo deve aparecer apenas uma vez em um nome de seletor. Caso você não siga esta regra você vai acabar escrevendo seletores gigantes e confusos, evite este tipo de seletor pai__filho__neto__bisneto__tataraneto.

A abordagem de seletores netos não deve ser utilizada. O nome da classe de um componente deve apenas conter o nome do bloco pai seguido do nome do componente e não deve ser inserido outros componentes sucessores ou antecessores. Um exemplo de código errado e posteriormente um exemplo do código correto são apresentados com base na figura 7.

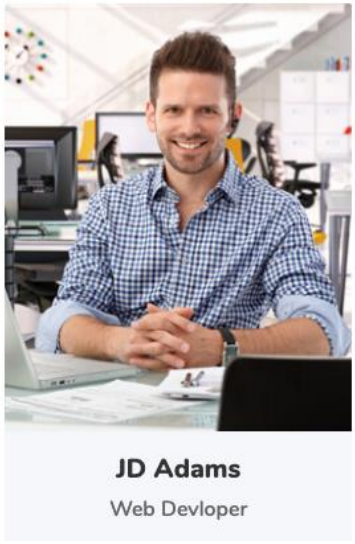

Figura 7 - Bloco avatar-card

Exemplo de código errado para criar o avatar-card com seletores netos.

▶CSS avatar-card.css

```
.avatar-card{...}
.avatar-card__imagem{...}
.avatar-card__rodape{...}
.avatar-card__rodape__titulo{...}
.avatar-card__rodape__subtitulo{...}
```

▶HTML arquivo.html

```
<div class="avatar-card ">
    <img class="avatar-card__imagem" >
    <div class="avatar-card__rodape">
        <p class="avatar-card__rodape__titulo"></p>
        <p class="avatar-card__rodape__subtitulo"></p>
    </div>
</div>
```

Exemplo de código correto para criar o avatar-card

▸CSS avatar-card.css

```
.avatar-card{...}
.avatar-card__imagem{...}
.avatar-card__rodape{...}
.avatar-card__titulo{...}
.avatar-card__subtitulo{...}
```

▸HTML arquivo.html

```
<div class="avatar-card">
    <img class="avatar-card__imagem" >
    <div class="avatar-card__rodape">
        <p class="avatar-card__titulo"></p>
        <p class="avatar-card__subtitulo"></p>
    </div>
</div>
```

9.2. Componentes Cruzados

O problema cruzado acontece quando um bloco precisa ter um comportamento diferente por causa do bloco pai. Sabendo que não podemos duplicar o código ou deixá-lo dependente do contexto, precisamos de uma abordagem um pouco diferente.

Para manter um padrão em nosso site, podemos transformar o botão em um bloco para ser utilizado em diversos lugares em nosso layout. Na figura 8 é apresentado o layout do nosso botão e na figura 9 a newsletter. Precisamos que o botão tenha algumas diferenças sutis de estilo quando estiver dentro do bloco pai “newsletter”, por exemplo, os cantos superior e inferior do lado esquerdo não pode ser arredondado. Como essas diferenças não correspondem a nenhuma das classes modificadoras já existente no botão, precisamos criar uma nova classe que seja escalável.

A primeira ideia que surgi é criar uma classe “botao__newsletter” na folha de estilo do bloco botão, mas teríamos um acoplamento com o elemento pai. Esse acoplamento ocorre, porque o bloco botão seria o responsável por mudar o

comportamento do botão quando estivesse dentro do bloco pai “newsletter”, isso deixaria o botão dependente do contexto.

Para que o bloco botão não dependa do contexto, deve-se criar a classe newsletter__botao dentro da folha de estilo do bloco newsletter, desta forma eliminaríamos o problema do acoplamento. Esta é a mesma abordagem que utilizamos para fazer a geometria dos blocos filhos.

Figura 8 - Bloco botao

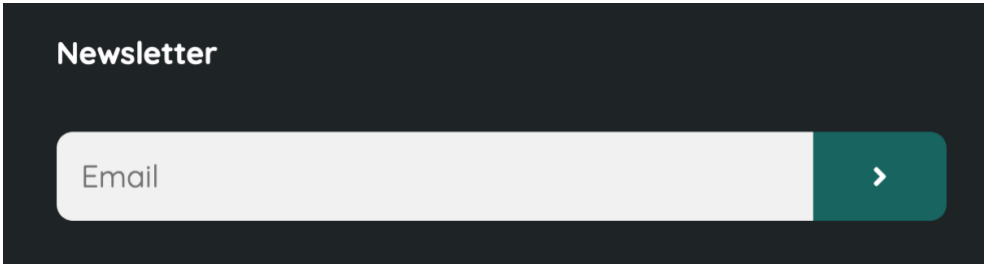

Figura 9 - Bloco newsletter

▶ CSS botao.css

.botao{…}

▶ CSS newsletter.css

. newsletter{…}
. newsletter__botao{…}
. newsletter__input{…}

▶HTML arquivo.html

```
<form class=“newsletter”>
    <input class=“newsletter__input”>
    <input class=“botao newsletter__botao”>
</form>
```

Os atributos de estilo exclusivos foram aplicados a newsletter__botao, se você decidir remover o botão da newsletter, os estilos exclusivos do botão no bloco newsletter também serão removidos. Se mais tarde durante o desenvolvimento, esse mesmo estilo de botão surgir mais duas vezes em contextos diferentes, é mais apropriado criar uma classe modificadora dentro do bloco botão. Se a classe for utilizada várias vezes, não estaríamos criando uma classe dependente do contexto, por ser usado em mais blocos.

10. Conclusão

A ideia de criar uma interface de usuário a partir de blocos padronizados e fáceis de entender, que se comportam de maneira previsível, independente do contexto, é um ótimo conceito.

O principal objetivo dessa metodologia, além de manter os códigos simples na hora da escrita e principalmente da manutenção, é fazer com que qualquer desenvolvedor possa ter total autonomia para mexer em qualquer parte do código do projeto. Seja num projeto que já conheça ou num projeto que acabou de entrar.

Aconselho fortemente que você use alguma metodologia modular em seu projeto. Particularmente eu prefiro a metodologia BEM por ter boa documentação e ser amplamente utilizada pelos demais programadores.

Neste ebook abordei a metodologia BEM, fiz algumas modificações de acordo com o meu ponto de vista. Não sou muita a favor de misturar diferentes metodologias ou qualquer outra abordagem, mas por questões práticas as vezes é necessário.

Resumo para implementar os princípios BEM em um projeto**:**

- Dívida seu CSS em pequenos blocos reutilizáveis.
- Nunca escreva estilos que atinjam outros blocos.
- Use classes modificadoras para fornecer várias versões do mesmo bloco.
- Divida as grandes construções em blocos menores;
- Construa suas páginas juntando vários blocos.
- Reutilize os blocos
- Nomeie seus seletores de maneira informativa e clara.
- Agrupe todas as regras de um bloco na sua respectiva folha de estilo.
- Use uma convenção de nomenclatura como a convenção de nomes BEM alternativo estilo dois traços