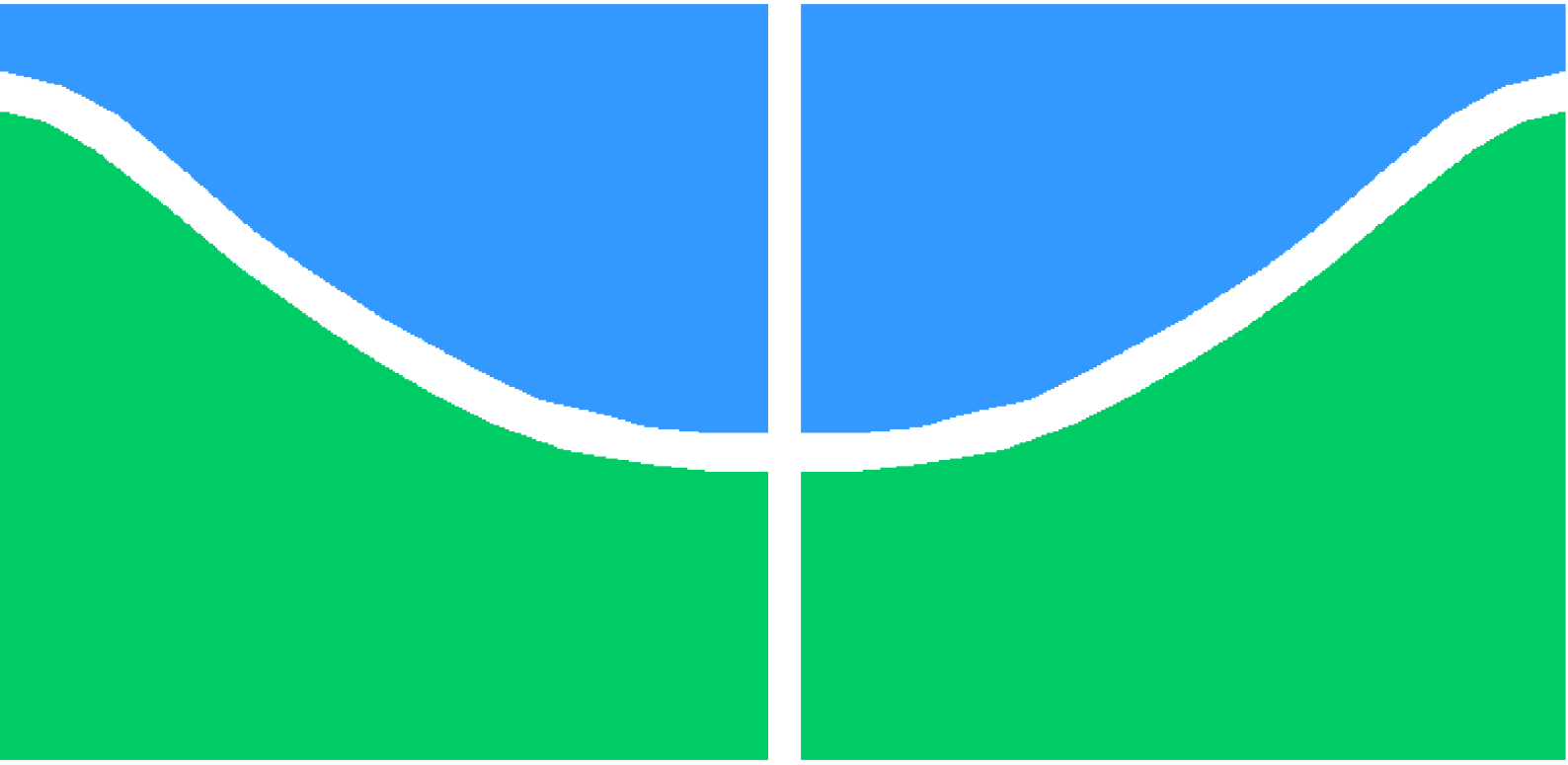

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Engenharia de Software

Comparativo entre as metodologias atômica e tradicional no desenvolvimento de CSS

Autor: Rafael dos Santos Rabetti

Orientador: Prof. Dr. Fábio Macêdo Mendes

Brasília, DF

Rafael dos Santos Rabetti

Comparativo entre as metodologias atômica e tradicional no desenvolvimento de CSS

Monografia submetida ao curso de graduação em (Engenharia de Software) da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em (Engenharia de Software).

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Orientador: Prof. Dr. Fábio Macêdo Mendes

Brasília, DF

Rafael dos Santos Rabetti
Comparativo entre as metodologias atômica e tradicional no desenvolvimento de CSS/ Rafael dos Santos Rabetti. – Brasília, DF, -
55 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Fábio Macêdo Mendes

Trabalho de Conclusão de Curso – Universidade de Brasília - UnB
Faculdade UnB Gama - FGA , .

1. CSS atômico. 2. metodologia. I. Prof. Dr. Fábio Macêdo Mendes. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Comparativo entre as metodologias atômica e tradicional no desenvolvimento de CSS

CDU 02:141:005.6

Rafael dos Santos Rabetti

Comparativo entre as metodologias atômica e tradicional no desenvolvimento de CSS

Monografia submetida ao curso de graduação em (Engenharia de Software) da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em (Engenharia de Software).

Trabalho aprovado. Brasília, DF, 22 de Agosto de 2019:

Prof. Dr. Fábio Macêdo Mendes
Orientador

Prof. Joenio Marques da Costa
Convidado 1

Rodrigo Maia Pereira
Convidado 2

Brasília, DF

Sumário

Lista de ilustrações

Lista de tabelas

Resumo

A organização do CSS em grandes projetos de desenvolvimento *web* é um desafio. As ferramentas e processos disponíveis não são tão maduros quanto em outras áreas mais convencionais da Engenharia de Software. O crescimento constante do CSS ao longo do desenvolvimento de projetos web aumenta a dificuldade de manutenção e refatoração desse código e, ainda, compromete a eliminação de código obsoleto. O CSS atômico propõe uma abordagem em que as classes devem ser pequenas, com um propósito único e com nomes baseados na variação visual, evitando a redundância e reduzindo o tamanho, migrando a compexidade do CSS para o HTML. Esse trabalho tem como objetivo analisar e quantificar aspectos do uso das metodologia atômica e tradicional no desenvolvimento de CSS em repositórios de código aberto.

Palavras-chave: CSS, CSS atômico, metodologia, análise

Introdução

Contextualização

O CSS é uma linguagem de estilo usada para descrever a apresentação de documentos escritos normalmente em HTML e XML. Essa apresentação é definida por estilos que são aplicados a elementos específicos. O CSS detalha como os elementos devem ser renderizados na página.

Ao passo que os grandes sites evoluem, O CSS tem a tendência de se tornar cada vez mais difícil de se gerenciar. O LinkedIn, por exemplo, possui mais de 1100 arquivos de Sass (pré processador de CSS), sendo 230 mil linhas de SCSS e mais de 90 desenvolvedores escrevendo Sass todos os dias (EPPSTEIN, 2013).

O reaproveitamento de código e estilos, especialmente em grandes projetos, é difícil e trabalhoso. A adição de novos estilos e novas linhas de código se torna uma opção máis fácil e mais rápida. Desse modo, é comum que um código antigo se torne obsoleto, causando um inchaço no CSS.

Abaixo vemos uma lista que apresenta o número de declarações únicas de CSS em grandes *sites*:

- GitLab: 402 cores de texto, 239 cores de fundo, 59 tamanhos de fonte
- Buffer: 124 cores de texto, 86 cores de fundo, 54 tamanhos de fonte
- HelpScout: 198 cores de texto, 133 cores de fundo, 67 tamanhos de fonte
- Gumroad: 91 cores de texto, 28 cores de fundo, 48 tamanhos de fonte
- GitHub: 197 cores de texto, 177 cores de fundo, 51 tamanhos de fonte
- ConvertKit: 128 cores de texto, 124 cores de fundo, 70 tamanhos de fonte

Esse número elevado de declarações únicas reduz a performance do time de desenvolvimento e também do site em questão, já que arquivos maiores levam mais tempo para carregar. Além disso, ao passo que mais declarações surgem, mais difícil se torna a refatoração desse código. Quanto maior o arquivo de CSS, maior será a dificuldade de gerenciamento do mesmo e a sua manutenção se torna muito onerosa.

As regras no CSS possuem escopo global. Alterar uma regra pode gerar erros, portanto, é comum que novas regras sejam criadas e as regras antigas acabam ficando obsoletas porque, muitas vezes, evita-se editá-las ou removê-las.

Além disso, mundaças simples de requisito podem exigir mudanças extensas no CSS, criando novas regras a partir de novos seletores. Essas mudanças criam arquivos grandes e complexos e acabam causando um inchaço no CSS (KOBLENTZ, 2013).

Arquivos grandes resultam em um tempo de carregamento maior e um mau uso do *cache* do navegador (ARDELJAN, 2016). Além disso, a dificuldade de gerenciamento desses projetos aumenta gradativamente. Por isso, várias metodologias foram e estão sendo criadas para evitar esses problemas.

Este trabalho discute a metodologia atômica como uma forma de mitigar esses problemas e a compara com outras abordagens mais tradicionais como o BEM e o SUIT CSS.

Objetivo

Geral

Analisar e quantificar aspectos do uso das metodologias atômica e tradicional no desenvolvimento de CSS em repositórios de código aberto.

Específicos

- Criar comparações entre o uso das metodologias atômica e tradicional em relação a quantidade de linhas de estilo
- Validar o uso da metodologia atômica e verificar se ela realmente reduz o esforço de desenvolvimento do CSS

Organização do documento

Além do capítulo de introdução, este trabalho está organizado em mais 3 capítulos:

- Capítulo 2 - (Referencial Teórico) Contém o referencial teórico sobre CSS. Além disso possui um conteúdo sobre as ferramentas que são utilizadas dentro do projeto.
- Capítulo 3 - (Resultados) Apresenta as comparações realizadas e os resultados obtidos.
- Capítulo 4 - (Conclusão) Apresenta uma conclusão a este trabalho e propostas para trabalhos futuros.

Referencial teórico

Este capítulo discute conceitos importantes para compreender e definir recomendações arquiteturais para criação de folhas de estilo CSS (do inglês, *Cascading Style Sheets*). Mais especificamente, estamos interessados na abordagem de CSS atômico, que será discutida adiante.

CSS

Funcionamento do CSS

O CSS é a linguagem que define como os documentos são exibidos aos usuários em uma página web. É no CSS em que se define várias características visuas que um elemento pode possuir, como sua altura, largura, cor e seu posicionamento, entre outras.

Um documento é normalmente um arquivo estruturado utilizando uma linguagem de marcação, sendo o HTML a mais comum. Esse documento é apresentado pelo navegador na forma de páginas usáveis para os usuários.

O navegador também é responsável a aplicar as regras de estilo do CSS ao documento para afetar como este será apresentado. Um conjunto de regras contidas em uma folha de estilo determina a aparência de uma página.

Abaixo segue um exemplo da interação de um arquivo HTML com um arquivo de CSS complementar:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Meu Exemplo</title>
    <link rel="stylesheet" href="main.css">
  </head>
  <body>
    <h1>Olá Mundo!</h1>
    <p>Este é um exemplo de HTML com CSS</p>
  </body>
</html>
```

CSS complementar:

```
h1 {
  color: blue;
  background-color: yellow;
  border: 1px solid black;
}

p {
  color: red;
}
```

A primeira regra do arquivo de CSS começa com um seletor `h1`, isso significa que as regras definidas nesse seletor serão aplicadas a todos os elementos `<h1>` do documento. Esses elementos terão o texto com a cor azul, cor de fundo amarela e com borda de 1 pixel, sólida e de cor preta.

Do mesmo modo, a segunda regra possui um seletor `p` que aplica a regra `color: red` a todos os elementos `<p>` do HTML.

Então, entende-se que o CSS é uma linguagem de busca de elementos através de seletores, para aplicar o estilo associado, e assim alterar a apresentação do documento.

No exemplo abaixo, é definida uma regra para todas as marcações `<p>` em um documento.

```
p{
    color: red;
}
```

Essa regra é composta por:

- **Seletor** O elemento HTML que será estilizado, neste caso o elemento `<p>`.
- **Declaração** A declaração é o conjunto da propriedade com o valor, neste caso `color: red;`
- **Propriedade** Qual propriedade será afetada por essa declaração, neste caso é a propriedade `color`.
- **Valor** Define uma entre várias possibilidades de aparência para uma dada propriedade, existem vários valores para `color` além de `red`.

A regra acima seleciona apenas um único elemento e modifica apenas uma propriedade, porém é possível selecionar vários elementos e também modificar várias propriedades

em uma única regra. Além disso, é possível ser mais específico ao selecionar elementos, podendo selecionar através de classes ou identificadores de elementos, por exemplo.

As regras em CSS são sempre globais, podendo afetar qualquer página em que forem aplicadas, o que pode ocasionar problemas, como veremos nos tópicos posteriores.

Especificidade

A noção de cascata é fundamental para o entendimento do CSS (*Cascading Style Sheets*). É a partir dessa noção que se determina qual propriedade modificará um elemento (FRIEDMAN, 2007). A cascata segue três passos para determinar essa modificação: importância, especificidade e fonte. Ao final desse processo, é assinalado um peso para cada regra. Esse peso determina qual regra terá precedência e, sendo assim, qual regra será aplicada, no caso de haver mais de uma regra para um elemento específico.

A cascata recebe uma lista não ordenada de valores declarados para uma propriedade específica de um dado elemento, e a ordena pela precedência da declaração, retornando um único valor de cascata. O resultado da cascata é uma lista ordenada de valores declarados para cada propriedade em cada elemento e segue os critérios abaixo.

1. Especificidade

Toda regra CSS tem um peso relacionado à ela, regras de peso maior tem maior importância e, são escolhidas em detrimento à regras de menor importância. Esse peso é o que define qual regra será aplicada ao elemento quando se tem regras conflitantes.

Ao se obter a importância de uma regra, é atribuida uma especificidade. Caso uma regra seja mais específica que outra, ela a sobrescreve.

Para calcular a especificidade, temos as seguintes ordens lexicográficas:

- **Elemento ou pseudo-elemento**: [0,0,0,1]
- **Atributo**: [0,0,1,0]
- **Classe ou pseudo-classe**: [0,0,1,0]
- **Identificador único (id)**: [0,1,0,0]
- **Estilo em linha**: [1,0,0,0]
- **!important**

Assim podemos calcular a especificidade dos seletores contando o número de elementos de cada tipo.

- `p.note`: 1 classe + 1 elemento = [0,0,1,1]
- `#sidebar p[lang="en"]`: 1 ID + 1 atributo + 1 elemento = [0,1,1,1]
- `body #main .post ul > li:last-child`: 1 ID + 1 classe + 1 pseudo-classe + 3 elementos = [0,1,2,3]

Se dois seletores aplicam estilo a um elemento, apenas o estilo do seletor mais específico será aplicado. A especificidade está relacionada ao peso em que um seletor possui em relação a outro. A variação desse peso entre vários seletores pode ocasionar em um estilo não aplicado.

Logo, sabendo que a especificidade é, basicamente, a ordem lexicográfica de um seletor, então um seletor de ordem [0,1,0,0] prevalece sobre um seletor de ordem [0,0,100,1000]. Se dois seletores possuirem a mesma especificidade, a ordem de escrita prevalece, o último sobrescreve os anteriores.

Finalmente, a declaração `!important` sobrescreve qualquer regra não importa sua especificidade. É a declaração mais forte.

2. Importância

Uma folha de estilo pode ter diferentes fontes:

1. **Usuário agente** - folha de estilo padrão do navegador
2. **Usuário** - opções do navegador do usuário
3. **Autor** - O CSS fornecido pela página (em linha, embutido ou externo)

Por padrão, essa é a ordem em que esses estilos são processados, então a regra definida no estilo do autor tem mais peso que as outras.

Além disso precisamos levar em conta a declaração `!important`. Se no nível do usuário for utilizada essa declaração, ela terá precedência sobre qualquer outra, mesmo se no nível do autor também for usada. Levando isso em consideração, temos, finalmente, a ordem crescente de precedência:

1. Usuário agente
2. Usuário
3. Autor
4. Animações
5. Autor `!important`

6. Usuário `!important`
7. Usuário agente `!important`
8. Transições

É importante destacar que essa ordenação de precedência é apenas utilizada para elementos e propriedades com a mesma especificidade, visto que esta tem mais importância.

3. Herança

Além dos tópicos citados acima, a herança também afeta os elementos da cascata. Porém, a herança não tem efeito nenhum sobre a especificidade ou a importância de uma regra. A herança em CSS é, simplesmente, uma forma de propagar uma propriedade de um elemento pai para os seus filhos.

Algumas propriedades CSS são herdadas pelos elementos filhos por padrão. Por exemplo, se atribuirmos uma fonte específica à *tag* `body`, essa fonte será herdada por outros elementos filhos, como `h1` e `p`. Essa herança acontece sem que nenhuma linha de código a mais tenha que ser escrita. Isso já acontece naturalmente porque toda propriedade CSS possui um valor inicial padrão que já é herdado aos seus elementos filhos.

As regras de herança são complicadas pelo fato de que é possível alterar as regras de herança utilizando propriedades CSS. Por exemplo, `width: inherit;` faz um elemento herdar a propriedade `width` do elemento pai.

4. Ordem de aparição

A última declaração em um documento tem maior precedência se todos os critérios acima forem iguais. Por isso, as declarações de folhas de estilo importadas são ordenadas como se as regras de `@import` fossem substituídas pela respectiva folha de estilo .

Metodologias de desenvolvimento

Em projetos grandes e complexos, a forma de organização do código é crucial para a eficiência. Diminuir o tempo de escrita de código, diminuir a quantidade de código a ser escrito e diminuir a quantidade de informações que o navegador terá de carregar são metas importantes em projetos de alta performance. Por isso, várias metodologias

de desenvolvimento de CSS foram criadas a fim de reduzir a complexidade e facilitar a manutenção desses arquivos.

As metodologias estabelecidas possuem o objetivo comum de evitar conflitos de especificidade e aumentar o reuso do código.

BEM

A principal finalidade do BEM é desacoplar o CSS da estrutura da DOM, o tornando mais independente. Isso é obtido por meio da adição de mais classes ao HTML para acessá-las diretamente, mantendo assim a especificidade baixa.

A estrutura do BEM (*Blocks, Elements and Modifiers*) segue a seguinte lógica:

- ***blocks*** são entidades que fazem sentido por si só:
  - `cabeçalho`, `recipiente`, `menu` , `lista`
- ***elements*** são parte de um bloco que apenas fazem sentido de ligadas à um bloco:
  - `menu item`, `list item`
- ***modifier*** é um modificador em um bloco ou elemento a fim de alterar aparência ou comportamento:
  - `disabled`, `checked`, `color yellow`

Nessa metodologia existe uma regra de nomeação que segue um padrão.

- A primeira classe sempre será um *block*: `.lista`
- Para criarmos os *elements*, utilizamos dois sublinhados (`__`) após o nome do nosso *block*: `.lista__item .lista__titulo`
- Para criarmos os *modifiers* utilizamos dois traços (`--`) após um *block* ou *element*: `.lista__item--destacado .lista__autor--ativo`

```
<div class="autor-bio">
    <img class="autor-bio__imagem autor-bio__imagem--grande"  src="http" alt="">
    <div class="autor-bio__conteudo">
        <h2 class="autor-bio__nome">Meu nome</h2>
        <p class="autor-bio__corpo">
            Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nam pretium.
        </p>
```

```
    </div>
</div>
```

Este trecho poderia ser suplementado pelo seguinte CSS:

```
.autor-bio {
    background-color: white;
    ...
}
.autor-bio__imagem {
    display: block;
    ...
}
.autor-bio__imagem--grande {
    width: 100%;
    ...
}
.autor-bio__conteudo {
    padding: 1rem;
}
.autor-bio__nome {
    font-size: 1.25rem; ;
}
.autor-bio__corpo {
    font-size: 1rem;
}
```

As mudanças de atualizações visuais ficam restritas apenas ao CSS e não há problemas de especificidade de seletor. O HTML continua fazendo sentido semanticamente sem haver nenhum estilo em linha. Já que o BEM não utiliza seletores filhos, os possíveis problemas de especificidade são evitados.

SUITCSS

O SUITCSS é uma metodologia que foca em melhorar a experiência de escrever o código CSS em projetos de desenvolvimento baseado em componentes.

Essa metodologia se assimila ao BEM em vários aspectos, já que podemos considerar um componente como um bloco e seus filhos como elementos. Porém a convenção de

nomes é diferente. Além disso, o SUIT aceita um nível a mais de precedência para representar o estado de um componente. O estilo dessa classe nunca deve ser definido diretamente, ela deve ser usada como uma classe adjacente ao componente, como por exemplo, `.AutorBio-corpo.is-expanded`.

```
/** CSS */

.AutorBio{ /* ... */ }
.AutorBio--grande{ /* ... */ }
.AutorBio-imagem{ /* ... */ }
.AutorBio-conteudo{ /* ... */ }
.AutorBio-nome{ /* ... */ }
.AutorBio-corpo.is-expanded{ /* ... */ }
```

```
<!-- HTML -->

<div class="AutorBio">
    <img class="AutorBio-imagem" src="https://..." alt="">
    <div class="AutorBio-conteudo">
        <h2 class="AutorBio-nome">Meu nome</h2>
        <p class="AutorBio-corpo is-expanded">
            Lorem ipsum dolor sit amet, consectetur adipiscing elit.
        </p>
    </div>
</div>
```

Além disso, a componentização também é usada a nível de variáveis.

```
/** @define AutorBio*/

:root {
  --AutorBio-border-width: 5px;
}

.AutorBio{
  border-width: var(--AutorBio-border-width);
}
```

Evita-se criar dependência entre os componentes, mesmo que o seu código não fique totalmente "DRY" (*don't repeat yourself*). Esse isolamento de componentes previne um aumento de complexidade e é chave para o reuso.

OOCSS

O OOCSS tenta transpor conceitos de orientação a objetos para o CSS. Seu objetivo é tornar o CSS mais gerenciável por meio da aplicação dos princípios do desenvolvimento orientado a objeto, que foram amplamente difundidos por linguagens de programação como o Java e o Ruby. Nessa metodologia, vários conceitos da programação orientada a objeto são incluídos, como o princípio da responsabilidade única e a separação de interesses.

O objeto no OOCSS é qualquer padrão visual que pode ser especificado em trechos de código. Elementos de uma página ou até grupos de elementos podem ser tratados como entidades únicas na folha de estilo.

Duas grandes regras definem o OOCSS, a primeira é estabelecer uma divisão clara entre estrutura e estilo. A estrutura de uma aplicação diz respeito a aspectos que não são diretamente visíveis ao usuário, como posição e tamanho de um elemento. O estilo de uma aplicação diz respeito aos apectos visuais de um elemento, como cores e fontes.

A segunda regra é separar o contêiner do conteúdo. Como regra geral, estilos nunca podem estar restritos a um conteúdo específico. Desse modo, a reutilização de estilos é facilitada. Além disso, para evitar que existam padrões de estilo repetidos no código, o OOCSS recomenda que uma nova classe seja criada a partir desse padrão.

Essa metodologia tem como premissa evitar o uso de seletores filhos, sendo uma boa estratégia para separar contêiner de conteúdo e, assim, aplicando classes únicas a elementos únicos.

ITCSS

Essa metodologia usa a analogia de um triângulo invertido com camadas para explicar os seus conceitos. Diferentemente das outras metodologias, o ITCSS não utiliza nenhuma convenção de nomeclatura, mas é uma metodologia de organização dos módulos e arquivos de CSS (GRUIJS, 2017). Existem três princípios:

1. **Do genérico ao explícito** Inicia-se pelo pelo mais genérico e baixo nível dos estilos, que tem uma alta abrangência. Podendo ser uma configuração de fonte ou variáveis da paleta de cores.
2. **De baixa a alta especificidade** Os seletores de baixa especificidade aparecem no começo do seu projeto. E gradativamente vão aparecendo os seletores com especificidade maior, evitando conflitos e o uso do `!important`.
3. **De longo alcance a localizado** No começo do projeto aparecem os seletores que afetam vários elementos ao mesmo tempo, e gradativamente vão aparecendo seletores mais localizados que afetam apenas um componente.

Esses três princípios são aplicados nas camadas do triângulo invertido de modo que cada camada é uma progressão lógica da camada anterior. Aumenta a especificidade, estreita-se o alcance e aumenta a explicitude a cada camada avançada para baixo. As camadas são as seguintes:

1. **Configurações** - Variáveis de configuração globais como cores, espaçamentos, fontes, etc.
2. **Ferramentas** - Funções ou mixins que ficarão disponíveis globalmente.
3. **Genérico** - Estilos de alto nível como Normilize.css, CSS resets, box-sizing, etc.
4. **Elementos** - Estilos de elementos puros e sem classe como underline em hover de link e font-size de headers.
5. **Objetos** - Elementos com classes como containers e rows. A grid também é definida aqui.
6. **Componentes** - A maioria da estilização acontece aqui, componentes como card ou navbar.
7. **Trunfos** - Caso seja necessário sobreescrever algum estilo e usar o `!important`.

Ao passo que essa metodologia define as 7 camadas, não é necessário o uso de todas elas, porém se o uso da sobreposição de estilos está se fazendo necessária, é quase certo que a estrutura de camadas utilizada não está adequada.

Vale ressaltar ainda que o ITCSS pode ser usado com conjunto de metodologias como o BEM ou o SUITCSS, visto que a ITCSS define uma organização de arquivos com base na precedência e não uma convenção de nomeclatura para classes de CSS.

CSS Atômico

CSS Atômico, também conhecido como CSS Funcional, é uma abordagem de escrita CSS que estabelece que as classes devem ser pequenas, com um propósito único e com nomes baseados na função visual. Ao mapear as classes para um estilo único, tem-se

um conjunto de regras mais granular e, com isso, ganha-se em reusabilidade (KOBLENTZ, 2013).

Abaixo segue um exemplo de uma regra CSS aplicando o princípio Atômico:

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

Uma das premissas dessa metodologia é que a complexidade deve estar no HTML e não no CSS. Com isso, é possível reutilizar as classes, resultando em arquivos de CSS mais simples e de tamanho reduzido e limitado.

O CSS atômico, diferentemente das outras metodologias, apresenta uma solução não apenas para o gerenciamento de folhas de estilo e nomeclaturas de classe. O CSS atômico, além de atender esses quesitos, propõe uma metodologia que impacta no projeto como um todo, visto que temos como resultado folhas de estilo minimalistas e decisões de estilo centradas no HTML.

Essa abordagem é constrastante se compararmos à metodologia tradicional de separação de interesses. No CSS atômico, as classes têm responsabilidade única e não há nenhuma separação entre conteúdo e as decisões de estilo. Entende-se que a possibilidade da reutilização de classes é mais valiosa do que um HTML independente das decisões de estilo.

Abordagens comuns

O termo CSS atômico se aplica à filosofia arquitetural e não à uma variação específica(POLACEK, 2017). Abaixo temos uma lista de variações do uso do CSS atômico e alguns termos relacionados.

As Variações do uso do CSS Atômico

Estático

O método estático é a forma padrão de se escrever CSS. Esse método é bastante utilizado e normalmente se usa em conjunto com um pré-processador para gerar a biblioteca de classes.

Nesse méotodo, o nome das classes está diretamente relacionado com a variação visual que aplicam. As classes declaram a propriedade que será modificada, podendo conter uma progressão numérica para as propriedades dimensionáveis. Uma das vantagens de utilizá-lo é por ele ser familiar e, por isso, a sua curva de aprendizado é pequena.

```
.f1 { font-size: 3rem; }
.f2 { font-size: 2.25rem; }
.i {          font-style: italic; }
.b {          font-weight: bold; }
.underline { text-decoration: underline; }
.strike {    text-decoration: line-through; }
.ttc {        text-transform: capitalize; }
.ttu {        text-transform: uppercase; }
```

Programático

O método programático envolve alguma ferramenta para gerar os estilos que são escritos diretamente no HTML.

```
<!-- Exemplo de CSS Atômico Programático -->
<div class="Bgc(#0280ae) C(#fff) P(20px)">
    Lorem ipsum
</div>
```

Geraria as seguintes declarações:

```
.Bgc\(#0280ae\) { background-color: #0280ae; }
.C\(#fff\) { color: #fff; }
.P\(20px\) { padding: 20px; }
```

Essa abordagem usa a sintaxe de uma chamada de função com parâmetros. Utilizando uma ferramenta para gerar os estilos, não é necessário escrever nenhuma linha de CSS. Toda a folha de estilo é gerada durante o processo de *build* e é totalmente otimizada, sem nenhum estilo não usado ou algo do tipo.

Longhand(Legível)/*Shorthand*(Abreviado)

Outra decisão arquitetural importante consiste na metodologia para definir os nomes das classes. Aqui temos dois outros métodos, o *Longhand* tem como preferência a

escrita legível e por isso os nomes das classes normalmente são longos. Já o *Shorthand* tem como preferência a escrita mais concisa e objetiva.

```
/* Exemplo de CSS Atômico Shorthand */

.bg-blue { background-color: #357edd; }
.f1 { font-size: 3rem; }
.ma0 { margin: 0; }

/* Exemplo de CSS Atômico Longhand */

.background-blue  { background-color: #357edd; }
.text-h1 { font-size: 3rem; }
.text-3rem { font-size: 3rem; }
.text-huge { font-size: 3rem; }
.fontsize-1 { font-size: 3rem; }
.margin-0 { margin: 0; }
```

É comum que abordagens *Longhand* favoreçam classes que espelhem as propriedades CSS correspondentes. Assim o nome da classe é semelhante, ou muitas vezes igual, ao nome da propriedade que ela define.

Componentes

Além dessas outras abordagens, na metodologia atômica, existe a possibilidade de criação de componentes a partir de estilos que se repetem, criando uma combinação de classes de utilidade para serem utilizadas juntamente sem a necessidade de repetí-las sempre.

Termos Relacionados

Classes de Utilidade

Classes de utilidade são classes de fácil entendimento, por serem objetivas, e de responsabilidade única que aplicam alguma regra de estilo.

Várias arquiteturas de CSS se apoiam no uso de algumas classes de utilidade para algumas funcionalidades, porém não adotam o conceito de CSS atômico, como é o caso do Bootstrap.

CSS Imutável

Um aspecto do CSS atômico é que as classes de utilidade nunca devem ser modificadas, o que resulta em resultados altamente confiáveis. A imutabilidade é essencial para a execução adequada da arquitetura do CSS Atômico. Normalmente as classes de utilidade são fornecidas por uma biblioteca e o usuário somente compõe estas classes no HTML.

Uma das principais motivações do CSS atômico é mover a complexidade das folhas de estilo para o HTML. Quando ocorrem mudanças no design, é melhor termos um HTML bem estruturado para que possamos trocar de `.background-blue` para `.background-black` rapidamente.

Breakpoint Prefixing

É uma técnica que permite que as classes de utilidade usem da sobreposição de estilos em *breakpoints* diferentes para que a implementação de telas responsivas seja simples e eficaz.

```
/* Exemplos de breakpoint prefixing */

.grid-12   /* Tamanho tela cheia padrão */
.m-grid-6  /* 2 colunas em telas médias */
.l-grid-4  /* 3 colunas para telas grandes */
```

Frameworks Existentes

Normalmente, a utilização do CSS atômico se dá a partir do consumo de folhas de estilo de frameworks já existentes, abaixo são citados dois frameworks que são comumente utilizados na comunidade.

Tailwind

O Tailwind CSS é um framework que segue o padrão legível na nomeclatura das classes, é escrito em PostCSS e é configurado em Javascript.

Tachyons

O Tachyons é um framework que segue o padrão abreviado na nomeclatura das classes e é bem compacto. A premissa do Tachyons é que 15kb de CSS após a compressão e a minificação é o necessário para desenvolver qualquer aplicação.

Justificativa

CSS é simples e deve ser responsável por ações simples (ARDELJAN, 2016). Ao se utilizar classes em CSS com responsabilidade única, é possível reutilizá-las infinitamente e isso resulta em arquivos de CSS pequenos e objetivos, o que traz benefícios à perfomance dos projetos.

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

Considerando que performance é um requisito, optar por essa abordagem sucede em um carregamento mais rápido da página, o que afeta positivamente os usuários. 300kb de CSS pode demora tipicamente 10 segundos para se carregar em uma conexão 3G, dependendo da sua velocidade (ARDELJAN, 2016).

Toda nova linha de CSS é como um quadro branco, nada impede que se use valores arbitrários. Toda nova linha de CSS é uma nova oportunidade para um aumento de complexidade, escrever mais CSS nunca deixará o CSS mais simples (WATHAN, 2017).

Porém, é importante ressaltar que não existe desacoplamento completo entre CSS e HTML. Ao escrever CSS semântico, ainda assim, não há uma separação total de interesses. O HTML pode estar livre de decisões de estilo porém, o CSS fica dependente da estrutura de marcação feita pelo HTML. Do mesmo modo, ao escrever CSS atômico, o CSS se torna independente da estrutura do HTML, porém o HTML é inteiramente afetado pelas decisões de estilo.

Além disso, é importante ressaltar que CSS atômico não é a mesma coisa que CSS *inline*. Com o CSS *inline* existe o mesmo problema da metodologia tradicional, em que não há regras para os valores escolhidos, podendo escolher qualquer valor arbitrário. E, além disso, no CSS *inline*, não é possível fazer as *media-queries* para manter a responsividade do design.

Ao seguir o princípio da separação de interesses, de acordo com os proponentes do CSS atômico, é inevitável que não ocorra um inchaço do arquivo de CSS, além de causar redundância e um mau gerenciamento de *cache* do navegador. Mudanças simples no estilo de um elemento muitas vezes resultam em novas regras na folha de estilo incentivando a duplicação de código (KOBLENTZ, 2013).

CSS reutilizável é a melhor opção, principalmente em aplicações grandes (GALLAGHER, 2012). Nessas aplicações percebe-se que ao passo que o código cresce, mais difícil é a sua manutenção. Utilizando componentes CSS reutilizáveis reduz o tempo de escrita e edição do código. Já que, nesse caso, é necessário aplicar mudanças nas classes

do HTML. É uma prática que beneficia os desenvolvedores visto que mudanças no HTML são mais literais (GALLAGHER, 2012).

Tecnologias e Ferramentas

Esta seção descreve as tecnologias e ferramentas utilizadas neste projeto.

Cloc

Ferramenta de código aberto utilizada para contar as linhas de código de um determinado projeto. O CLOC conta as linhas de código, dividindo o resultado por tipo de arquivo, mostrando assim a quantidade de linhas de cada tecnologia.

Ainda, é possível mostrar esses resultados em diferentes formatos, podendo ser gerado um arquivo *txt* ou *csv*, por exemplo.

Abaixo, temos um exemplo do uso do CLOC.

```
prompt> cloc gcc-5.2.0/gcc/c
      16 text files.
      15 unique files.
       3 files ignored.

https://github.com/AlDanial/cloc v 1.65
T=0.23 s (57.1 files/s, 188914.0 lines/s)
-------------------------------------------------------------------------------
Language             files         blank        comment             code
-------------------------------------------------------------------------------
C                       10          4680          6621            30812
C/C++ Header             3            99           286              496
-------------------------------------------------------------------------------
SUM:                    13          4779          6907            31308
-------------------------------------------------------------------------------
```

CSS Stats

O Css Stats é uma ferramenta de código aberto que gera uma análise das folhas de estilo de determinado projeto. Para realizar essa análise, é necessária a *URL* do projeto em funcionamento. Assim, a partir do dados de estilo obtidos, é possível diagnosticar áreas de alta complexidade, reastrear a perfomance e melhorar a consistência do *design*.

O Css Stats gera uma página com vários dados sobre as folhas de estilo do projeto. Nessa página, é possível visualizar o tamanho do arquivo antes e depois da compressão

e ainda analisar vários resultados sobre a consistência do design. Por exemplo, quantas declarações únicas de tamanho e cor de fonte.

Abaixo temos uma imagem de exemplo com alguns resultados.

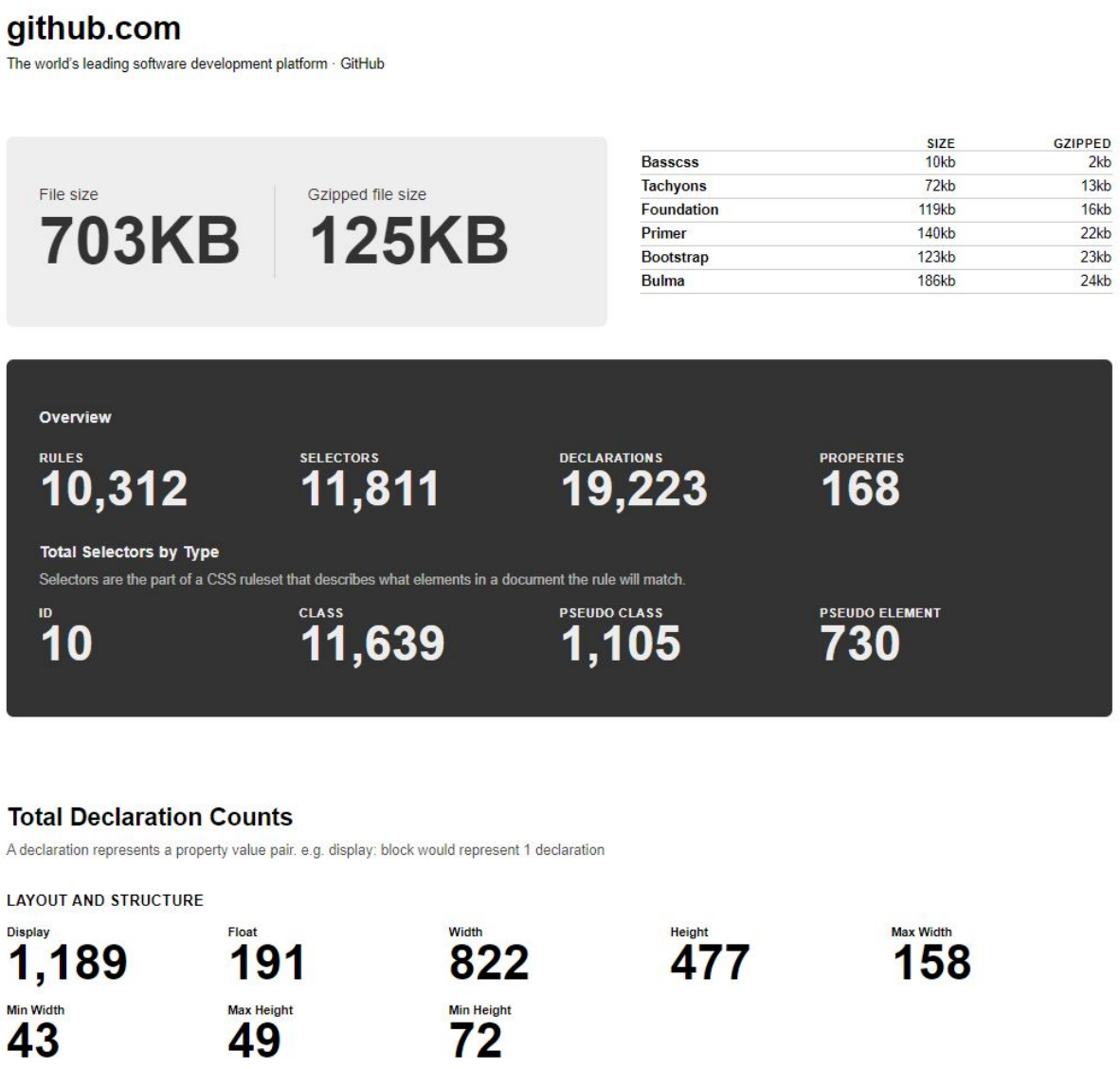

Figura 1 – Dados de estilo da página do Github.com

Git Walk

O Git Walk é um *script* de código aberto no qual é possível navegar entre os *commits* de um projeto. Por meio de comandos simples, é possível acessar os commits que foram realizados na linha do tempo de desenvolvimento de um projeto.

Para visualizar o útlimo commit, rodamos o seguinte comando:

`git walk (last|latest)`

Para visualizar o primeiro commit, rodamos o seguinte comando:

`git walk (last|latest)`

Para visualizar o próximo commit, rodamos o seguinte comando:

`git walk next [n_commits]`

Para visualizar o commit anterior, rodamos o seguinte comando:

`git walk prev [n_commits]`

Seaborn

O Seaborn é uma biblioteca de visualização de dados em Python baseado na biblioteca do Matplotlib. Com o Seaborn é possível criar gráficos atrativos e informativos por meio de uma interface de alto nível.

Abaixo temos uma imagem que mostra um exemplo de um gráfico gerado na ferramenta Seaborn.

```
In [87]: SeabornDataTudo = [
             [11132, 87.73, 12.27,"Goodwork", "Atômico"],
             [9867,90.08, 9.92,"Statusfy", "Atômico"],
             [287621, 87.96, 12.04, "Speedcube", "Atômico"],
             [2992, 100, 0, "Developmint", "Atômico"],
             [1275, 94.35, 5.65, "Litcher", "Atômico"],
             [115097, 70.79, 29.21, "Gitlab", "Tradicional"],
             [3071, 22.08, 77.92, "Casper", "Tradicional"],
             [8749, 29.99, 70.01, "Electronjs.org", "Tradicional"],
             [17655, 58.31, 41.69, "Tildes", "Tradicional"],
             [2673, 77.52, 22.48, "Tusk", "Tradicional"]
         ]

         dfTudo = pd.DataFrame(dataTudo, columns = [
                                      "Total de linhas do projeto",
                                      "Porcentagem de linhas de código de programação",
                                      "Porcentagem de linhas de estilo",
                                      "Nome do Projeto",
                                      "Metodologia"])

         b = sns.scatterplot(
             x="Total de linhas do projeto",
             y="Porcentagem de linhas de estilo",
             size="Total de linhas do projeto",
             sizes=(60, 600),
             alpha=0.5,
             hue="Metodologia",
             data=dfTudo
         )
         b.set_ylim(-1,101)
         b.legend(loc='center left', bbox_to_anchor=(1.25, 0.5), ncol=1)

Out[87]: <matplotlib.legend.Legend at 0x7f1bc3cba5c0>
```

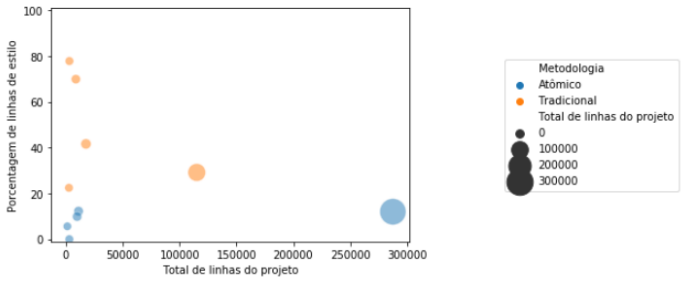

Figura 2 – Gráfico gerado com o Seaborn que foi utilizado nos resultados deste trabalho

Google Sheets

O Google Sheets é um programa de planilha incluído como parte de um pacote de escritório de *software* gratuito baseado na Web oferecido pelo Google em seu serviço Google Drive. Nestre projeto, o Google Sheets foi utilizado para estudo e visualização dos dados gerados, em *csv*, pela ferramenta CLOC. Os dados foram formatados e, assim, vários gráficos foram gerados. Alguns deles estão presentes nesse trabalho.

Resultados

Avaliação da Metodologia

O primeiro passo para a avaliação da metodologia foi buscar uma lista de projetos para realizar a análise. Essa busca foi feita nos principais sistemas de gerenciamento de projetos e versões de códigos, como o GitHub e o Gitlab. O objeto de busca estabelecido foi de projetos de desenvolvimento web.

A partir de uma lista disponível dentro de um projeto chamado *awesome-tailwindcss* no GitHub, foram selecionados 5 projetos que aplicam a metodologia atômica no seu desenvolvimento, utilizando o Tailwind CSS.

A lista de 5 projetos que seguem a metodologia tradicional foi selecionada a partir da lista de projetos do Gitlab, de forma aleatória.

Com os projetos selecionados, foi utilizado o CLOC (*Count Lines of Code*) para contar a quantidade de linhas de código de cada tecnologia em um determinado projeto. As amostras foram coletadas no último *commit* e também foi coletado o histórico dessa quantidade ao longo dos *commits* do projeto. Para isso, foi utilizado o *Git Walk*, que é um *script* em *Python* utilizado para facilitar a navegação entre os commits de um projeto, facilitando assim a coleta de dados para compor o histórico do projeto.

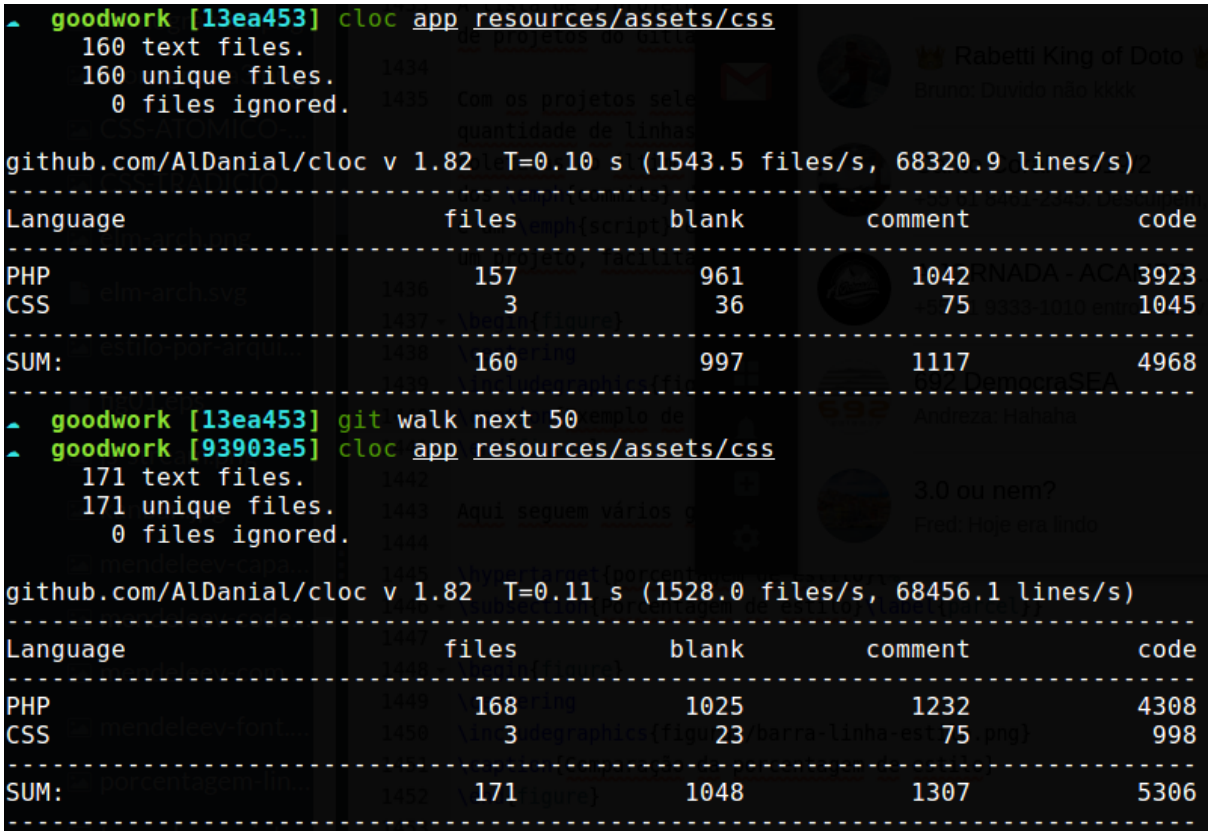

Figura 3 – Exemplo de utilização do *cloc* e do *git walk*.

Dessa forma, os dados foram obtidos e formatados para as futuras análises. Duas tecnologias foram usadas para gerar gráficos, o *Seaborn*, utilizando a linguagem *Python* e o *Google Sheets*.

A seguir, são apresentados os gráficos gerados com o foco de comparar as metodologias atômica e tradicional.

Porcentagem de estilo

Na figura abaixo, temos um gráfico que mostra o percentual final de linhas de estilo em relação ao total de linhas, tanto de estilo como de código de programação. Podemos perceber que há uma porcentagem maior de linhas de estilo na metodologia tradicional.

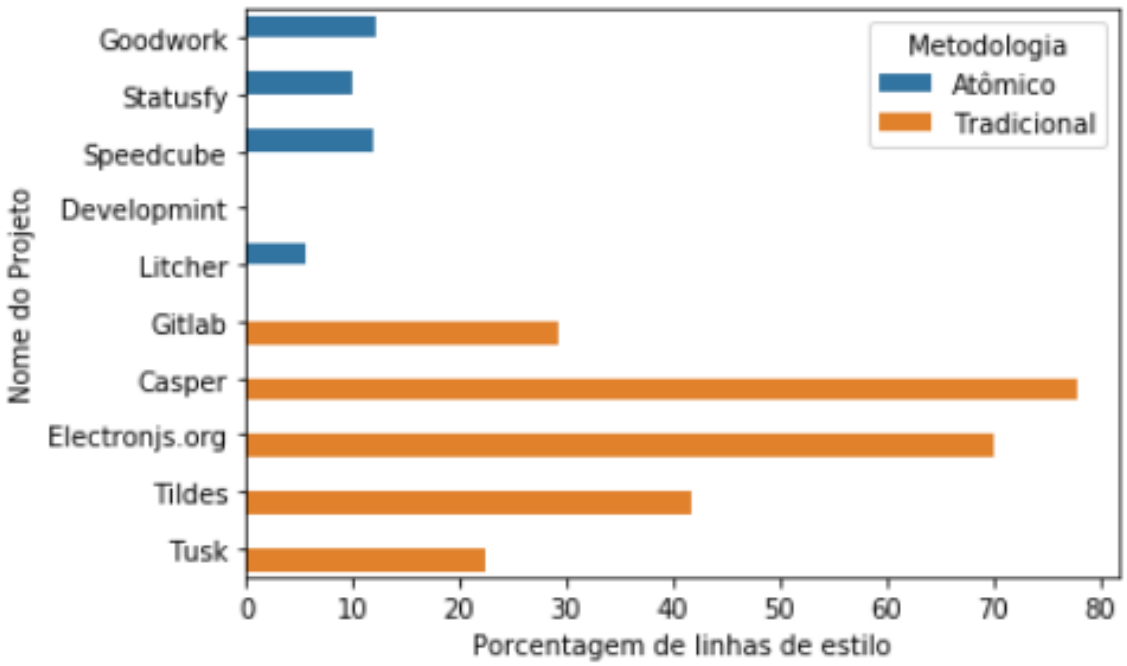

Figura 4 – Comparação da porcentagem de estilo

Porcentagem de estilo em relação ao tamanho do projeto

Abaixo, temos dois gráficos que buscam analisar a mesma situação. A partir deles, podemos concluir que as linhas de estilo tem uma porcentagem maior nos projetos que seguem a metodologia tradicional. Além disso, podemos verificar que o total de linhas de um projeto não é um fator que afeta na porcentagem de estilo.

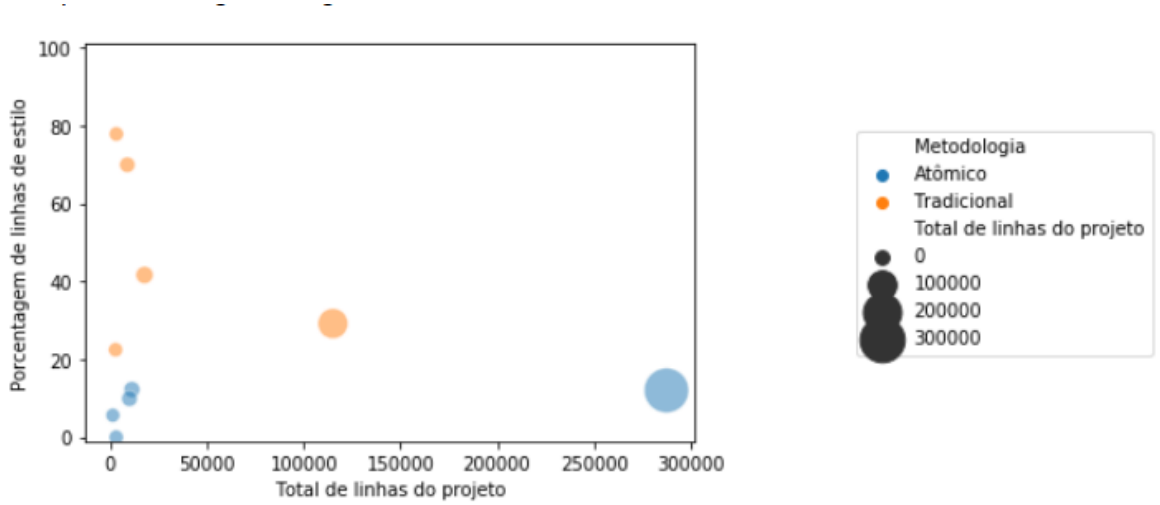

Figura 5 – Comparação da porcentagem de estilo

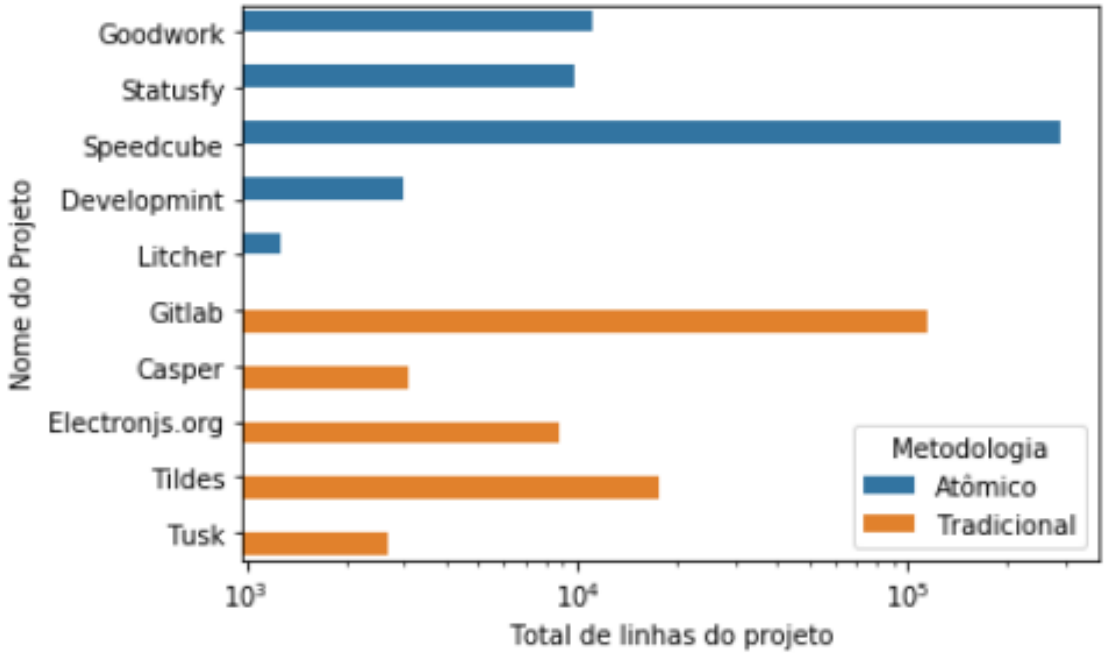

Figura 6 – Total de linhas dos projetos

Porcentagem de estilo ao longo do projeto

No gráfico abaixo, é feita uma comparação do crescimento do CSS ao longo do desenvolvimento de cada projeto, desse modo é possível analisar a porcentagem de estilo ao longo do desenvolvimento do projeto. Para a geração desse gráfico, foram alinhadas seis coletas de dados durante o desenvolvimento dos projetos, sendo a primeira no início e a última ao fim, para manter o histórico ao longo dos projetos.

Assim como nos outros gráficos, podemos tirar a mesma conclusão em relação a porcentagem de estilo e a metodologia utilizada ao longo do desenvolvimento dos projetos. Podemos concluir também que há uma variação consideravelmente maior entre os valores máximos e mínimos nos projetos de metodologia tradicional, já que não existe um padrão a se seguir. Além disso, podemos afirmar que há uma tendência de queda da porcentagem de estilo a partir de um determinado momento em ambas metodologias, sendo que na metodologia atômica essa tendência ocorre mais cedo.

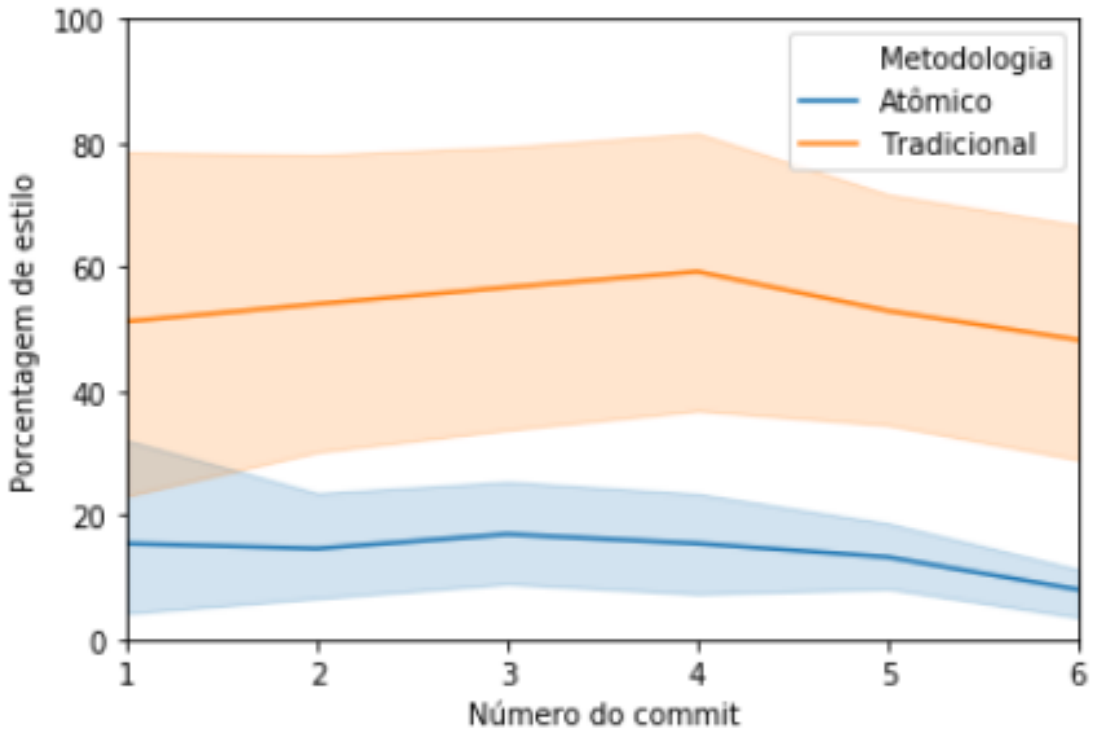

Figura 7 – Comparação da porcentagem de estilo

Quantidade de linhas de estilo por arquivo

A partir do gráfico abaixo, podemos concluir que a métrica de quantidades de linhas de estilo por arquivo não segue nenhum padrão. Não há distinção entre as duas metodologias nesse quesito.

Geralmente os arquivos de CSS são maiores, porém não há uma relação entre o tamanho dos arquivos de CSS e o tamanho dos arquivos de código de programação.

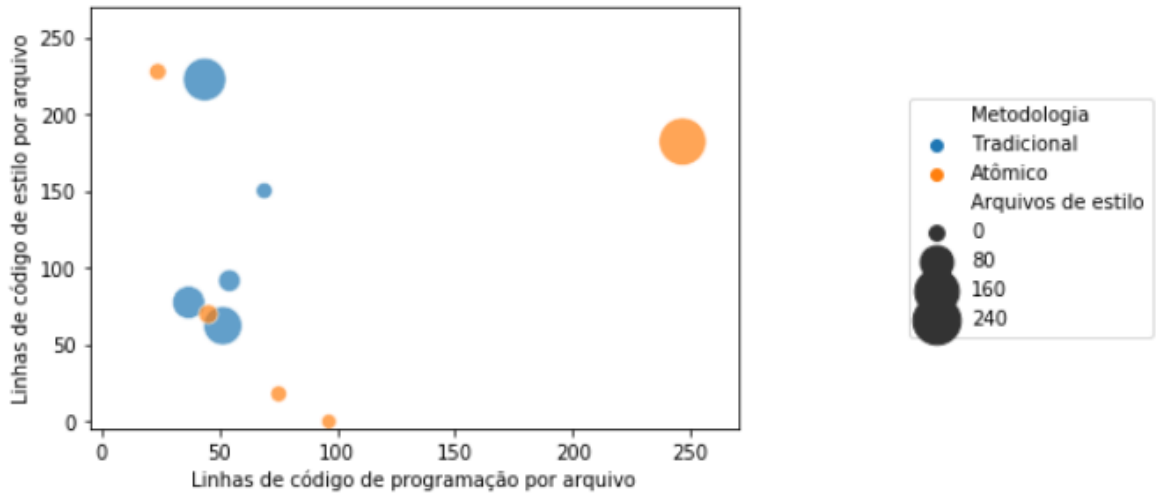

Figura 8 – Comparação da porcentagem de estilo

Histórico de commits dos projetos

Abaixo, temos um comparativo com o histórico de commits dos projetos de ambas as metodologias. Todos os projetos da mesma metodologia seguem um mesmo padrão e fica evidente que os projetos da metodologia tradicional possuem um percentual de linhas de estilo maior se comparado com os projetos de metodologia atômica. Ainda, os gráficos mostram que, com o uso da metodologia atômica, o crescimento das linhas de estilo é limitado. Já na metodologia tradicional não há um padrão, porém a tendência é um crescimento indefinido do CSS.

A partir desse resultado, poderia ser feita uma análise para calcular a economia de esforço entre as duas metodologias. O COCOMO (*Constructive Cost Model*) é um modelo de estimativa do tempo de desenvolvimento de um software, podendo assim avaliar o seu custo.

O COCOMO propõe a seguinte fórmula para um projeto do tipo orgânico (time de desenvolvimento pequeno com boa experiência trabalhando com requisitos não rígidos).

$$E = 3,2(KLOC)^{1,05}$$

E é o esforço calculado e $KLOC$ é o número de *kilo* linhas de código. Se calcularmos esse resultado levando em conta a quantidade de linhas de código de estilo, teríamos uma grande vantagem para a metodologia atômica. Entretanto, esse modelo não quantifica o esforço de refatoração e reescrita de código, que acontece com frequência maior nos projetos que seguem a metodologia tradicional.

Além disso, uma das maiores limitações desse modelo é que ele não leva em consideração o aumento da complexidade dos arquivos HTML, já que na metodologia atômica, todas as decisões de estilo são feitas nesses arquivos. Uma análise mais profunda teria que ser feita para analisar se o esforço economizado ao se escrever uma quantidade consideravelmente menor de linhas de código de estilo é vantajoso se compararmos com esforço extra empreendido ao realizar as decisões de estilo no HTML.

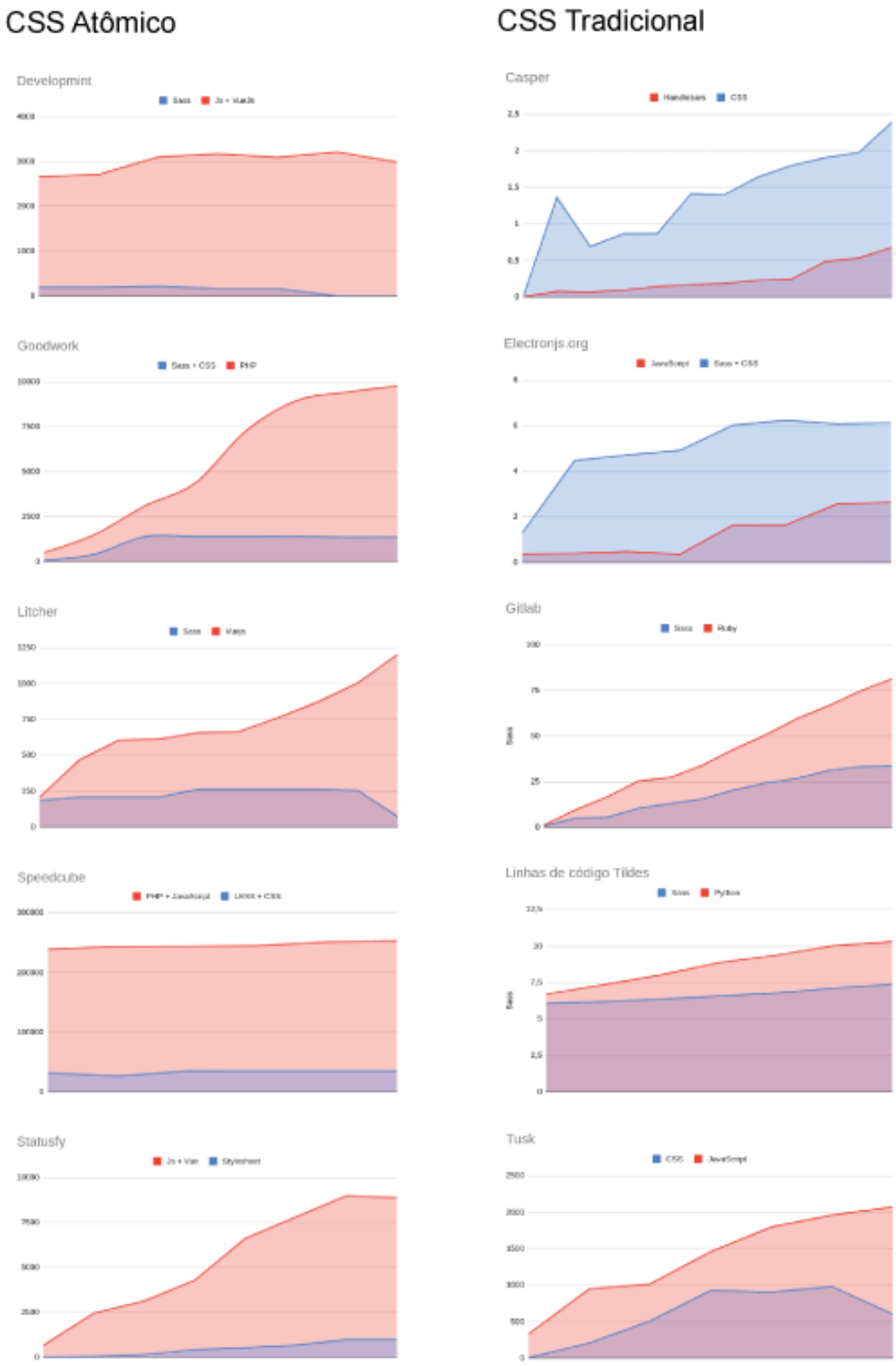

Figura 9 – Histórico de commits dos projetos

Abaixo temos duas tabelas que comparam algumas declarações de estilo entre projetos de metodologias diferentes.

	Atômico - Speedcube	**Tradicional - Gitlab**
Linhas de estilo	34621	33622
Declarações únicas de cores de fonte	16	402
Declarações únicas de cores de fundo	16	239
Declarações únicas tamanhos de fonte	18	59

	Atômico - Litcher	**Tradicional - Eletronjs.org**
Linhas de estilo	254	6125
Declarações únicas de cores de fonte	21	153
Declarações únicas de cores de fundo	13	167
Declarações únicas tamanhos de fonte	13	50

Em ambas análises, os projetos que seguem a metodologia atômica apresentaram uma menor diferença entre o total de declarações e o número de declarações únicas, o que evidencia uma maior reutilização de código. Além disso, esses projetos também apresentaram uma média menor de especificidade dos seus seletores, se comparado aos projetos de metodologia tradicional. Esses fatores aumentam a manutenibilidade do código e tornam o *design* mais consistente.

Conclusão

Atualmente existem várias ferramentas que buscam otimizar a escrita de folhas de estilo. O uso do CSS atômico resulta em um carregamento mais rápido da página, melhor uso do *cache* e ainda elimina a redundância e o inchaço do CSS (ARDELJAN, 2016; KOBLENTZ, 2013).

Existe uma relação entre o tamanho do arquivo de CSS e a perfomance da aplicação. Arquivos menores resultam em um tempo menor de carregamento da página (ARDELJAN, 2016; BECE, 2019). Porém, é necessário relatar que existem outros processos que são utilizados para reduzir o tamanho de um arquivo de CSS, como a minificação e a compressão.

A partir dos resultados obtidos, é seguro dizer que projetos que seguem a metodologia atômica possuem um percentual de linhas de estilo reduzido e com isso, há um ganho de performance na aplicação.

Tendo isso em vista, o objetivo do trabalho foi parcialmente alcançado. As comparações entre as duas metodologias foram estabelecidas e os resultados obtidos foram claros. Tendo o percentual de linhas de estilo em um projeto como um fator de perfomance da aplicação, aplicações que seguem a metodologia atômica são mais performáticas. Além disso, a manutenibilidade de projetos que seguem a metodologia atômica é maior e esses projetos tendem a ser mais consistentes em termos de *design*.

Porém, todas as comparações foram realizadas entre projetos distintos e analisar um mesmo projeto que sofreu uma refatoração nas suas folhas de estilo poderia trazer resultados interessantes para esta análise. Além disso, para validar o uso da metodologia atômica ainda seria necessário analisar a produtividade de uma equipe de desenvolvimento ao aplicar as duas metodologias.

A seleção aleatória dos projetos pode ter causado um certo viés na validade dos resultados, visto que a comparação foi feita entre projetos de natureza e objetivos diferentes. Além disso, comparar uma métrica entre projetos diferentes também pode gerar um viés.

Para trabalhos futuros, existem duas análises que podem ser realizadas. A primeira consiste na performance da equipe de desenvolvimento ao se utilizar as duas metodologias, podendo medir a produtividade e qual a curva de aprendizado necessária para ambas. A segunda consiste na análise de um projeto que sofreu uma completa refatoração das suas folhas de estilo, passando da metodologia tradicional para a atômica, podendo assim medir quais os ganhos e perdas nesse processo.

Todos os dados que foram gerados para a realização deste trabalho, inclusive este

texto, estão disponíveis no repositório do Github e podem ser acessados por este link https://github.com/rafaelrabetti/TCC-FGA-UnB.

Referências

ARDELJAN, P. 15kb of css is all you'll ever need. 2016. Disponível em: <https://medium.com/@philipardeljan/15kb-of-css-is-all-youll-ever-need--634da7258338>. Citado 3 vezes nas páginas 14, 29 e 45.

BECE, A. Improving css performance and file size - an in-depth guide. 2019. Disponível em: <https://dev.to/prototyp/improving-css-performance-and-file-size-an-in-depth-guide-4mb5>. Citado na página 45.

BEM Introduction. Disponível em: <http://getbem.com/introduction/>. Nenhuma citação no texto.

CASCADING. Disponível em: <https://www.w3.org/TR/css-cascade-4/\#cascading>. Nenhuma citação no texto.

CSS Stats. Disponível em: <https://cssstats.com/>. Nenhuma citação no texto.

EPPSTEIN, C. Joining linkedin. 2013. Disponível em: <https://chriseppstein.github.io/blog/2013/04/22/joining-linkedin/>. Citado na página 13.

FRIEDMAN, V. Css specificity: Things you should know. 2007. Disponível em: <https://www.smashingmagazine.com/2007/07/css-specificity-things-you-should-know/>. Citado na página 17.

GALLAGHER, N. About html semantics and front-end architecture. 2012. Disponível em: <http://nicolasgallagher.com/about-html-semantics-front-end-architecture/>. Citado 2 vezes nas páginas 29 e 30.

GIT Hooks. Disponível em: <https://githooks.com/>. Nenhuma citação no texto.

GIT Walk. Disponível em: <https://github.com/nok/git-walk>. Nenhuma citação no texto.

GITHUB Awesome TailwindCSS List. Disponível em: <https://github.com/aniftyco/awesome-tailwindcss>. Nenhuma citação no texto.

GRUIJS, L. The inverted triangle architecture: how to manage large css projects. 2017. Disponível em: <https://medium.freecodecamp.org/managing-large-s-css-projects-using-the-inverted-triangle-architecture-3c03e4b1e6df>. Citado na página 23.

HOW CSS Works. Disponível em: <https://developer.mozilla.org/en-US/docs/Learn/CSS/Introduction_to_CSS/How_CSS_works>. Nenhuma citação no texto.

KOBLENTZ, T. Challenging css best practices. 2013. Disponível em: <https://www.smashingmagazine.com/2013/10/challenging-css-best-practices-atomic-approach/>. Citado 4 vezes nas páginas 14, 25, 29 e 45.

OOCSS. Disponível em: <https://www.keycdn.com/blog/oocss>. Nenhuma citação no texto.

POLACEK, J. Let's define exactly what atomic css is. 2017. Disponível em: <https://css-tricks.com/lets-define-exactly-atomic-css/>. Citado na página 25.

SUIT CSS. Disponível em: <https://suitcss.github.io/>. Nenhuma citação no texto.

WATHAN, A. Css utility classes and "separation of concerns". 2017. Disponível em: <https://adamwathan.me/css-utility-classes-and-separation-of-concerns/>. Citado na página 29.

Apêndice

APÊNDICE A - Motivação do uso de CSS Atômico

Fase 1: CSS Semântico

Ao se aprender CSS, é muito comum ouvir dizer sobre algumas boas práticas da escrita. Uma delas é sobre a separação de interesses. Essa boa prática apresenta uma ideia em que o HTML deve manter informações apenas sobre o conteúdo e todas as suas decisões de estilo devem ser escritas no CSS. Então, as classes HTML devem ser puramente sobre o conteúdo o qual elas aplicam, como, por exemplo, `class="biografia"` ou `class="noticias"`. Portanto, os estilos de centralização e margem, por exemplo, estarão no CSS.

```
<div class="biografia">
    <img src="http...">
    <div>
        <h2>Title</h2>
        <p>Lorem Ipsum</p>
    </div>
</div>
```

Existem problemas com esse tipo de escrita, muitas vezes o CSS se torna um espelho do HTML, refletindo todas as estruturas com os seletores aninhados. Desse modo, apesar do HTML estar distante das estilizações, o CSS acaba se adaptando muito à estrutura de marcação. Isso mostra que os interesses não estão separados como deveriam.

Além disso, Thierry Koblentz, em seu artigo *Challenging CSS Best Practices*, expõe que o seguir o princípio da separação de interesses, leva à um inchaço do arquivo de CSS, além de causar redundância e um mau gerenciamento de *cache*. Mudanças simples no estilo do elemento resultam em novas regras na folha de estilo. Ainda, relata que o único modo de se aprimorar como autor de folhas de estilo é não seguindo esses preceitos.

Fase 2: Desacoplando Estilo da Estrutura

Para evitar esse acoplamento, recomenda-se adicionar mais classes ao HTML, como `class="biografia-corpo"` e `class="biografia-imagem"`. Desse modo, mantém-se a

especificidade dos seletores baixa e o seu CSS fica menos dependente da estrutura do HTML. Uma metodologia que utiliza esse padrão de escrita é o BEM, que já foi citado nesse documento.

```
<div class="biografia">
    <img class="biografia-imagem" src="http...">
    <div class="biografia-conteúdo">
        <h2 class="biografia-nome">Nome</h2>
        <p class="biografia-corpo">Lorem ipsum lorem ipsum</p>
    </div>
</div>
```

Dessa forma, o HTML permanece "semântico" e ainda não contém decisões de estilo, e CSS está desacoplado da estrutura de marcação. Porém existe um problema ao lidar com componentes similares.

Foi criado um novo requisito e é necessário adicionar um componente de visualização de artigo em um estilo bem parecido, com uma imagem e um conteúdo, com título e um resumo. A melhor forma de lidar com essa situação e ainda sim separar os interesses é escrever um componente novo.

```
<div class="artigo">
    <img class="artigo-imagem" src="http...">
    <div class="artigo-conteúdo">
        <h2 class="artigo-título">Nome</h2>
        <p class="artigo-resumo">Lorem ipsum lorem ipsum</p>
    </div>
</div>
```

Porém ao escrever o CSS para esse novo componente, seria uma duplicação do estilo do componente de `.biografia`. É uma opção, porém não segue os padrões de escrita de código, como o "DRY" (*don't repeat yourself*) e isso pode levar a inconsistências de design, caso um dos componentes seja alterado sem que aconteça as mesmas mudanças no outro.

Para evitar essa repetição, usando um pré-processador, existe a função `@extend` em CSS. Assim, é possível extender as funções do componente `.biografia`no componente `.artigo`. Com isso, o problema parece estar resolvido, a duplicação do CSS é removida e o HTML ainda está livre de decisões de estilo. Porém, ainda existe outra opção.

Apesar de não conter nenhum conteúdo semântico similar, os componentes `.biografia` e `.artigo` contém muito em comum em uma perspectiva de design. Assim,

podemos criar um componente `.cartão` genérico para representar essas características comum e aplicar aos dois casos.

```
<div class="cartão">
    <img class="cartão-imagem" src="http...">
    <div class="cartão-conteúdo">
        <h2 class="cartão-título">Nome autor / Título artigo</h2>
        <p class="cartão-resumo">Lorem ipsum lorem ipsum</p>
    </div>
</div>
```

Essa abordagem remove a duplicação, porém não está de acordo com o padrão de separação de interesse. O HTML agora está escrito de forma os dois componentes com a classe `.cartão` terão o mesmo estilo, e isso, claramente, é uma decisão de estilo. Além disso, não seria possível mudar o estilo de um sem alteração no outro. Para realizar uma estilização em apenas um dos componentes teríamos que editar o HTML, o que é contra a abordagem de separação de interesses.

Entretanto, o autor defende que a não separação de interesses não é simplesmente ruim e, na verdade, existem duas formas de se escrever HTML e CSS.

- **CSS que depende do HTML** Nessa abordagem, o HTML é independente porém, o CSS precisa saber as classes do HTML e precisa achá-las na estrutura. O HTML é reestilizável mas o CSS não é reutilizável.
- **HTML que depende do CSS** Já nessa outra abordagem, O HTML não é independente e utiliza classes que são pensadas primeiro no CSS, para atingir um estilo especificado. O CSS é reutílizável e o HTML não é reestilizável.

Nicolas Gallagher, em seu artigo *About HTML semantics and front-end architecture*, defende que um CSS reutilizável é a melhor opção, principalmente em aplicações grande. Nessas aplicações percebe-se que ao passo que o código cresce, mais difícil é a sua manutenção e utilizando componentes CSS reutilizáveis reduz o tempo de escrita e edição do mesmo. Já que, nesse caso, é necessário aplicar mudanças nas classes do HTML. O autor ainda afirma que é uma prática que beneficia os desenvolvedores visto que mudanças no HTML são mais literais.

Fase 3: Componente CSS Desconectado com Conteúdo

Ao criar classes abrangentes e reutilizáveis podemos criar composições novas juntando os componentes, sem escrever nenhuma linha de CSS. Se, por acaso, o componente

não for reutilizável ou então se a criação de subcomponentes causar muita repetição, deve-se focar na criação de classes mais abrangentes de modo a desconectar o máximo possível com o conteúdo. Desse modo, é possível contemplar todas as opções e, com isso, aumentar a reutilização.

Fase 4: Componente Desconectado com Conteúdo + Classes de Utilidade

Dar nomes aos componentes pensando na sua reutilização é um trabalho que demanda tempo. Por exemplo, se temos um componente genérico que se chama `.lista-de-ações` e gostaríamos que esse componente fosse alinhado à esquerda. Podemos criar um modificador `.lista-de-ações--esquerda` e aplicar a propriedade CSS `text-align: left;` à essa classe. Porém estamos criando um novo modificador para a `.lista-de-ações` que não será reutilizado em nenhum outro lugar fora desse componente.

Para resolver esse problema, podemos criar uma classe que seja capaz de ser reutilizável e que implemente o efeito desejado no componente. Podemos nomear a classe de `.align-left` que simplesmente irá alinhar o componente à esquerda. Assim, vemos que as classes de utilidade são uma ótima solução a fim de aumentar a reutilização do CSS.

Podemos também criar uma classe `.mar-r-sm` em que irá adicionar uma `margin-right: 1rem;` ao elemento. Com a adição dessas classes, além da já dita reutilização, o CSS tende a diminuir de tamanho, já que mais classes são adicionadas no HTML. Com isso, não há necessidade de criar nenhum tipo de estruturação dentro do CSS, deixando-o compacto e resolvendo qualquer tipo de problema com especificidade.

Fase 5: Priorizar a Utilidade

Com essa nova perspectiva, podemos criar um conjunto de classes de utilidade para contemplar vários aspectos como:

- Tamanho de texto, cores e pesos
- Bordas
- Cores de fundo
- *Grid* e *flexbox*
- Preenchimento e margem

```
<div  class="py-3 px-4 border-b border-dark-soft flex-spaced flex-y-center">
    <div  class="text-ellipsis mr-4">
        <a  href="..."  class="text-lg text-medium">
            Test-Driven Laravel
        </a>
    </div>
</div>
```

É possível criar uma interface totalmente nova sem escrever nenhuma nova linha de CSS. Ao seguirmos a trilha de criar componentes focando na sua reutilização, o uso de classes de utilidade é o caminho natural.

Segundo Wathan, um dos maiores benefícios de se ter um conjunto de classes de utilidades que sejam objetivas e agrupáveis é que todo desenvolvedor do time tem que escolher sempre um valor dentro de um conjunto de opções fixas.

Essa vantagem elimina problemas de inconsistência no seu CSS. Quando se escolhe um valor dentro de opções fixas, não há espaço para a criação de valores arbitrários. O desenvolvedor fica impedido de criar um novo valor para o tamanho de uma fonte porque, no seu julgamento, considerou que aquele texto específico deveria ser um pouco menor, por exemplo.

Abaixo vemos uma lista de grandes *sites* e o quanto esse problema pode ser impactante:

- GitLab: 402 cores de texto, 239 cores de fundo, 59 tamanhos de fonte
- Buffer: 124 cores de texto, 86 cores de fundo, 54 tamanhos de fonte
- HelpScout: 198 cores de texto, 133 cores de fundo, 67 tamanhos de fonte
- Gumroad: 91 cores de texto, 28 cores de fundo, 48 tamanhos de fonte
- GitHub: 197 cores de texto, 177 cores de fundo, 51 tamanhos de fonte
- ConvertKit: 128 cores de texto, 124 cores de fundo, 70 tamanhos de fonte

Quando todos os envolvidos no projeto tomam suas decisões de estilo a partir de um conjunto limitado de opções, previne-se que o CSS cresça em proporções desnecessárias e as definições são sempre consistentes no projeto.

Porém, ainda recomenda-se que componentes sejam criados. Deve-se construir o projeto seguindo a abordagem de se priorizar a utilidade e extrair componentes quando alguma duplicação aparecer. Abaixo temos um botão criado tomando como referência um conjunto de classes de utilidade, neste caso, o Tachyons:

```
<button class="f6 br3 ph3 pv2 white bg-purple hover-bg-light-purple">
    Salvar
</button>
```

Se for necessário utilizar esse botão com a mesma combinação de classes em outras partes do código, a abordagem recomendada pela ferramenta Tachyons é criar uma abstração por meio de um *template* e não pelo CSS. Com o Mithril.js, por exemplo, podemos criar um componente para usá-lo assim:

```
m(ButtonTeste, "purple")
```

A definição do componente pode ser escrita assim:

```
export let ButtonTeste = {
    view: (vnode) => {
        var classes = "";
        switch(vnode.children[0]){
            case("purple"):
                classes = ".white.bg-purple.hover-bg-light-purple";
                break;
            case("gray"):
                classes = ".mid-gray bg-light-gray hover-bg-light-silver";
                break;
        }
        return m(`button.f6.br3.ph3.pv2${classes}`, "Salvar")
    }
}
```

Essa é uma abordagem válida para vários projetos, porém há casos em que criar um componente CSS é mais prático. Com o Less, é possível criar um componente `.btn-purpple` apenas declarando as classes na definição da nova classe, já que, nessa ferramenta, é possível usar classes existentes como *mixins*. Já no Sass ou no Stylus, é necessário criar um *mixin* separado para cada classe de utilidade e depois importá-los na classe do componente.

Adam Wathan, relata que sua experiência com a abordagem de piorizar a utilidade o levou a desenvolver designs mais consistentes, por mais atípico que pareça, se comparado a uma abordagem priorizando componentes.

Thierry Koblentz aponta que a maior vantagem no uso do CSS atômico é que, ao utilizar o HTML para criar os estilos, reduzimos o escopo drasticamente. Já não se estiliza

no escopo global, a folha de estilo, e sim no nível de um elemento ou bloco. Não é preciso adicionar regras novas à folha de estilo para alterar o estilo de um módulo e, ainda, não há a possibilidade desse estilo novo interferir sobre outro estilo já existente, já que a regra é aplicada apenas no módulo que possui determinada classe. Não há redundância, seletores não são duplicados e não há problemas com especificidade. Além disso, ao abandonar um estilo a regra não fica obsoleta.