Review the OOCSS

OOCSS をはじめよう

CSS Talk vol.01 manabu yasuda

INTRODUCTION

自己紹介

安田 学 (yasuda manabu)

株式会社 TAM
マークアップエンジニア

https://github.com/manabuyasuda

@Gaku0318

株式会社TAMのテクニカルチームがお届けする技術情報！

TipsNote by TAM

エンジニア、デザイナー、ディレクター常時募集中です。
「wantedly TAM」で検索 !!

アジェンダ

01. CSSに**設計**が求められる理由

02. **避けるべき**CSS（アンチパターン）

03. OOCSSの**概要**

04. OOCSSの**2つの原則**

05. **オブジェクト**の役割と分離する粒度

01

CSSに設計が求められる理由

・既存のスタイルの影響範囲が広すぎて追加、変更しにくい。

・既存のスタイルの全体像が把握できない。

・既存のスタイルが再利用を前提に作られていない。

・再利用できるパーツが少ないので、コードが増えてしまう。

・レスポンシブだとコードが複雑になってしまう。

代表的なCSS設計の手法

- OOCSS
- SMACSS
- BEM
- FLOCSS
- ITCSS
- Atomic Design
- EDJO
- ECSS

その他にもいろいろ。

どれを選べばいい？

大切なこと

- CSS の設計手法は大まかなルールを定義してくれるもの。
- 設計の実践には小さな決定がたくさんある。
- CSS には避けるべきアンチパターンがある。
- CSS はアンチパターンの積み重ねで破綻する。
- 万能な設計手法はないので最適な手法を選択する必要がある。

何から始めればいい？

アンチパターンを避けるためのCSS設計の原点ともいえる

OOCSSから理解するのが近道。

02

避けるべきCSS（アンチパターン）

※自分が既存の案件を引き継いだという想定で
考えてみてください。

アンチパターン

① IDセレクタを使いすぎている

② ベーススタイルの詳細度を高くしている

③ 要素セレクタに具体的すぎるスタイル

アンチパターン

❶ IDセレクタを使いすぎている

```css
#main {}
#main #content {}
#main #content p {}
#main #content ul {}
#main #content ul li {}
#main #content ul li a {}
```

IDセレクタ 2つの指定は引き継がないと上書きできない。
セレクタが長いので見通しも悪い。

```css
#main {}
#main #content {}
#main #content p {}
#main #content ul {}
#main #content ul li {}
#main #content ul li a {}
/* 追加 */
#main #content .text {}
```

アンチパターン

❷ ベーススタイルの詳細度を高くしている

どこが問題なのかわかりますか？

```HTML
<a href="#" class="link">リンク</a>
```

```CSS
a:link { color: red; }

.link { color: blue; }
```

a タグのスタイルをクラス 1 つで上書きできない。

```html
<a href="#" class="link"> リンク </a>
```

HTML

```css
a:link { color: red; }

/* 適応されない */
.link { color: blue; }
```

CSS

CSS

```
/*
 * 0,0,0,1（要素セレクタ）
 * 0,0,1,0（クラスセレクタ）
 * 0,0,1,0（擬似クラス）
 */

/* 0,0,1,0（擬似クラス）+ 0,0,0,1（要素セレクタ）*/
a:link { color: red; }

/* 0,0,1,0（クラスセレクタ）*/
.link { color: blue; }
```

アンチパターン

❸ 要素セレクタに具体的すぎるスタイル

```css
.content h2 {
  position: relative;
  margin-bottom: 25px;
  padding: 0.5em 0.8em;
  border-radius: 5px;
  color: #fff;
  font-size: 20px;
  font-weight: bold;
  background: crimson;
}
.content h2:after {
  content: "";
  position: absolute;
  z-index: 1;
  bottom: -15px;
  left: 3em;
  margin-left: -15px;
  border: 15px solid transparent;
  border-top-color: crimson;
  border-bottom-width: 0;
}
```

CSS

h2見出し

CSS

```css
.content .heading {
  padding: 0.5em;
  border: 0; /* リセット */
  border-bottom: 1px solid;
  border-left: 6px solid;
  border-radius: 0; /* リセット */
  color: #000;
  background-color: #fff; /* 上書き */
}

.content .heading:after {
  content: none; /* リセット */
}
```

h2見出し

h2見出し

.content h2 のスタイルが当たってしまうので、

上書きやリセットが必要になってしまう…

既存のCSSを消してしまうと、

どんな影響が出るのかわからないから消せない…

アンチパターンを
できるだけ避けるべき!!

デザインを実現できるだけでは充分ではない。

スタイルの変更や追加を問題なくできるようにするために、

「CSSの設計」が必要。

03

OOCSSの概要

OOCSSとは

- ページ単位ではなく、**オブジェクト（部品）**単位。
- **小さく作って組み合わせる**ことでページを作る（Legos first）。
- 基本的に**クラスセレクタだけ**を使い、
  マルチクラスで指定していく（詳細度を低く保つ）。
- アンチパターンを避ける**ベストプラクティスの詰め合わせ**。

参考リンク

- stubbornella/oocss: Object Oriented CSS Framework
- stubbornella/oocss-accessibility-guidelines: The accessibility guidelines used by the OOCSS open source project
- Object Oriented CSS | SlideShare
- Web 制作者のための CSS 設計の教科書


Web制作者のための
CSS設計
の教科書
モダンWeb開発に欠かせない
「修正しやすいCSS」の設計手法
谷 拓樹 著
あとで困るコードを書いてない?
予測しやすい
保守しやすい
再利用しやすい
拡張しやすい
コンポーネントやCSSプリプロセッサを意識した設計・管理の実践など、
「ちゃんとCSSを書く」ためのプロの考え方を徹底解説!
インプレス

まずはこれ読んで！

2009年、OOCSSの提唱者 Nicole Sullivan はこう語っている。

> CSSは壊れやすい。始めることは簡単だけど。
> ファイルサイズは確実に増えていきます。
> なぜなら、コードの再利用はほとんどされないからです。
> クリーンなコードを壊すことは簡単です。
> 始めることが重要なのではなく、クリーンに保ち続けることが重要です。
> クレバーなオブジェクトを書きましょう。さもなければ、ブロックや
> ページ、複雑なコンテンツが増えるごとにコードも増え続けます。
>
> — Object Oriented CSS

04

OOCSSの2つの原則

OOCSSの2つの原則

① 構造と見た目を分離する（Separate Structure and Skin）

② コンテナとコンテンツを分離する（Separate Container and Content）

OOCSSの2つの原則

❶ 構造と見た目を分離する

（Separate Structure and Skin）

OOCSSはマルチクラス（複数のクラスを指定すること）が前提になる。
そして、クラスの１つ１つに役割をもたせて分離する。
この「役割がクラスを分ける粒度の単位」になる。

役割の1つに「構造と見た目」がある。

見た目が違うので気づきにくいが、３つのボタンには共通するものがある。それは「構造」。

TOPへ

詳しく見る

送信

```css
.button-top {
  display: inline-block;
  padding: 0.75em;
  line-height: 1;
  text-align: center;
  text-decoration: none;
  color: #fff;
  background-color: #f6aa10;
}
.button-more {
  display: inline-block;
  padding: 0.75em;
  line-height: 1;
  text-align: center;
  text-decoration: none;
  color: #fff;
  background-color: #89c245;
}
.button-send {
  display: inline-block;
  padding: 0.75em;
  line-height: 1;
  text-align: center;
  text-decoration: none;
  color: #fff;
  background-color: #f35353;
}
```

```css
.button-top {
  display: inline-block;
  padding: 0.75em;
  line-height: 1;
  text-align: center;
  text-decoration: none;
  color: #fff;
  background-color: #f6aa10;
}
.button-more {
  display: inline-block;
  padding: 0.75em;
  line-height: 1;
  text-align: center;
  text-decoration: none;
  color: #fff;
  background-color: #89c245;
}
.button-send {
  display: inline-block;
  padding: 0.75em;
  line-height: 1;
  text-align: center;
  text-decoration: none;
  color: #fff;
  background-color: #f35353;
}
```

```css
.button {
  display: inline-block;
  padding: 0.75em;
  line-height: 1;
  text-align: center;
  text-decoration: none;
}
```

それぞれのボタンを作るための
骨や筋肉になっている要素を
「構造」と呼ぶ。

OOCSSでは
構造という役割で分離する。
今回は .button というクラスを使用。

```css
.button-top {
  color: #fff;
  background-color: blue;
}
.button-more {
  color: #fff;
  background-color: orange;
}
.button-send {
  color: #fff;
  background-color: red;
}
```

特徴的だった色は、「見た目」という
役割で、それぞれ別のクラスへ。

構造を担当するクラスの名前を
引き継ぐと関係性を示すことができる。

コード量はどうなったのか？

27行だったコードが19行（約30％ダウン）になりました！

```
.button-success {
  display: inline-block;
  padding: 0.75em;
  line-height: 1;
  text-align: center;
  text-decoration: none;
  color: #fff;
  background-color: blue;
}
.button-warning {
  display: inline-block;
  padding: 0.75em;
  line-height: 1;
  text-align: center;
  text-decoration: none;
  color: #fff;
  background-color: orange;
}
.button-alert {
  display: inline-block;
  padding: 0.75em;
  line-height: 1;
  text-align: center;
  text-decoration: none;
```

```
.button {
  display: inline-block;
  padding: 0.75em;
  line-height: 1;
  text-align: center;
  text-decoration: none;
}
.button-success {
  color: #fff;
  background-color: blue;
}
.button-warning {
  color: #fff;
  background-color: orange;
}
.button-alert {
  color: #fff;
  background-color: red;
}
```

メリット

・バリエーションが増えても、色などの**最小限の指定だけで再現**ができるのでコード量は緩やかにしか増えない。

・共通する構造に変更があっても**１箇所変更するだけ**で対応可能。

・１つ１つのクラスは小さくなるので**全体像を把握**しやすい。

デメリット

- CSS はシンプルに保てる反面、HTMLに複数のクラスを指定するので**複雑**になる。
- 小さなパーツが多くなるので、**スタイルガイド**をつくって組み合わせるパターンを共有する必要がある。
- 組み合わせても干渉し合わないオブジェクトを作るのに**一定のスキル**が必要。

「構造と見た目の分離」のまとめ

- 構造と見た目という役割で**クラスを分離**する。
- クラスを組み合わせることで**メンテナンス性**が上がり
  パフォーマンスを向上させることもできる。
- CSSの代わりに**HTMLが複雑**になる。
- **組み合わせるのを補助できる環境**が必要。

OOCSSの2つの原則

❷ コンテナとコンテンツを分離する

（Separate Container and Content）

・コンテナとは内包する要素のこと

・コンテンツとは内包される要素のこと

例えるなら、
コンテナは鞄、コンテンツは鞄の中の書類。

container

contents

コンテンツとは

ページ内の内容（コンテンツ）のこと

例）
- 見出し
- 段落
- リスト
- 画像
- ボタン
- ラベルなど

コンテナとは

コンテナとはコンテンツを内包する要素のことで、
例えばヘッダーやフッターなどの共通エリアや本文を
内包する .content や .aricle のようなエリアのこと。

なぜ「コンテナとコンテンツの分離」をするのか？

構造と見た目の分離だけでは
再利用性を担保できないから。

例えるなら、
機能的にはデスクトップパソコンがいいけど、
持ち歩きたいなら、ノートパソコンのほうがいい。

desktop pc

note pc

#articleという場所に依存しているので使い回すことができない。

```css
/* Bad */
#article .button {}
#article .button-primary {}

/* Good */
.button {}
.button-primary {}
```

要素セレクタを指定することも要素という場所に
依存することになる。

例えば、以下の指定では`h2`にしか適応することができない。

```css
/* Bad */
h2.heading {}

/* Good */
.heading {}
```

つまり、コンテナとコンテンツの分離とは、

「場所に依存しないセレクタを書く」ということ。

05

オブジェクトの役割と分離する粒度

OOCSSには4つの役割の分け方がある。

- 構造と見た目
- コンテナとコンテンツ

役割について、もう少し考えてみよう。

このようなコードを見たことはありませんか？

HTML

```
<div class="container">
  <div class="row">
    <div class="col-md-8">.col-md-8</div>
    <div class="col-md-4">.col-md-4</div>
  </div>
</div>
```

Bootstrapのグリッドシステムのコード（以降グリッドオブジェクトと呼ぶ）。これもコンテナにあたる。

```HTML
<div class="container">
  <div class="row">
    <div class="col-md-8">.col-md-8</div>
    <div class="col-md-4">.col-md-4</div>
  </div>
</div>
```

つまり、コンテナとは

・任意のクラスを指定するだけでどこでも使える。

・レイアウトを担当するオブジェクト。

・内包する要素が何であっても機能する（干渉されない）。

と言い換えることもできます。

グリッドが便利なのは

コンテナという役割を持っているから。

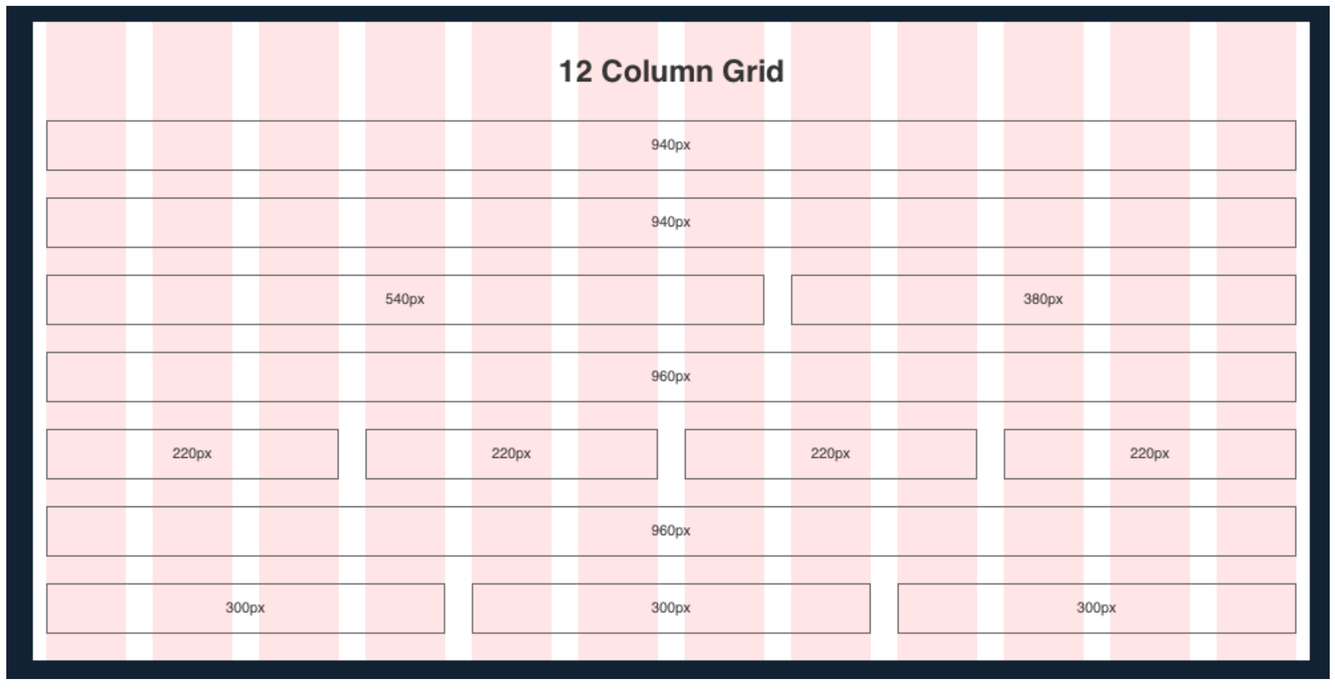
※OOCSS ではグリッドの使用を推奨している。

でも…

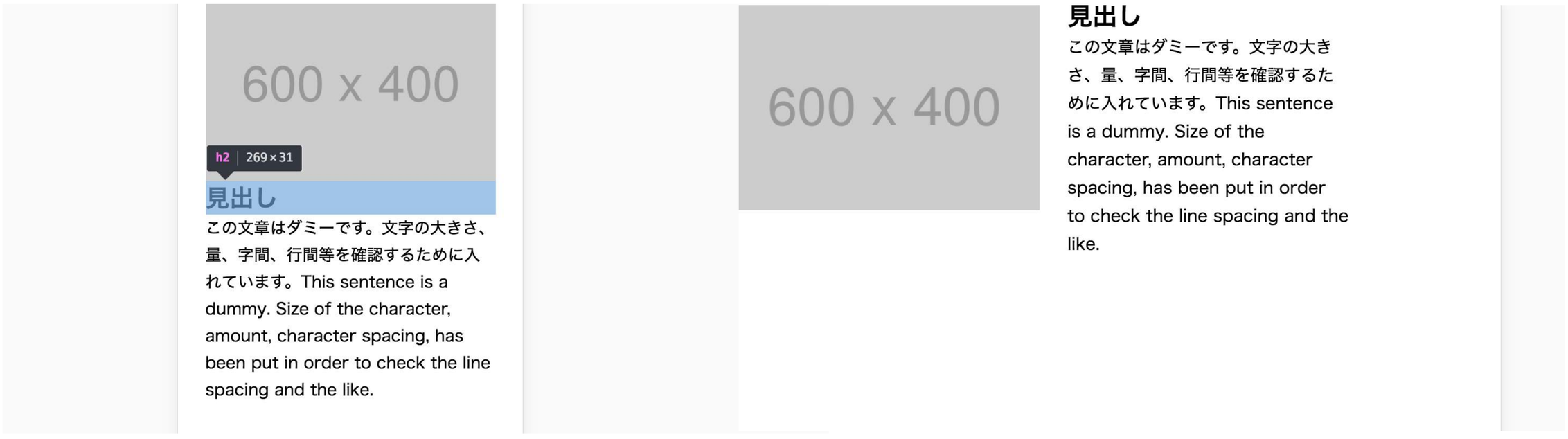
でも…

グリッドオブジェクトが便利だからといって、
どんな場面でも使うのは避けたほうがいい。

グリッドオブジェクトの役割は（主に）12で割った横幅を
各行（カラム）に割り当てること。
一般的に内包する要素には干渉しない。



例えば、スマホでは画像とテキストがブロック（縦に並ぶ）で、タブレット以降は50%ずつの横並びになるパターン。



ここでは.gridと.grid-cellというクラス名にする。

```HTML
<div class="grid">
  <div class="grid-cell grid-cell--md-6">
    <img src="#" alt="">
  </div>
  <div class="grid-cell grid-cell--md-6">
    <h2>見出し</h2>
    <p>テキスト</p>
  </div>
</div>
```

横並びのときに見出しと
画像の上のツラを合わせてるな。

h2 | 440×33

見出し

この文章はダミーです。文字

確認するために入れています

Size of the character, an

been put in order to che

like.

```html
<div class="grid">
  <div class="grid-cell grid-cell-md-6">
    <img src="#" alt="">
  </div>
  <div class="grid-cell grid-cell-md-6">
      <h2> 見出し </h2>
      <p> テキスト </p>
  </div>
</div>
```

HTML

```css
@media screen and (min-width: 768px) {
  .u-offset-top-md  {
    /**
     * line-height が 1.4 の場合 :
     * (1.4 - 1) / 2 = 0.2em
     */
    margin-top: -0.2em;
  }
}
```

CSS

縦に並んでるときに
画像とテキストの間に余白が入ってるな。

見出し

この文章はダミーです。文字の大きさ、量、字間、行間等を確認するために入れています。This sentence is a dummy. Size of the character, amount, character spacing, has been put in order to check the line spacing and the like.

```html
<div class="grid">
  <div class="grid-cell grid-cell-md-6 u-mb u-mb0-md"><!-- 追加 -->
    <img src="#" alt="">
  </div>
  <div class="grid-cell grid-cell-md-6">
      <h2 class="u-offset-top-md">見出し</h2>
      <p>テキスト</p>
  </div>
</div>
```

HTML

```css
.u-mb {
  margin-bottom: 0.5rem;
}

@media screen and (min-width: 768px) {
  .u-mb0-md  {
    margin-bottom: 0;
  }
}
```

CSS

本当にこれでいいの？

汎用的だけど、それはグリッドという役割に関してだけ。

無理やり使おうとすると、本当に汎用的かわからないよ
うな汎用クラスが増えていくだけ。
マークアップも複雑に…。

オブジェクトは
役割を明確に小さく

OOCSSでは「構造と見た目」や「コンテナとコンテンツ」で分離する。つまり、役割や機能でオブジェクトを分けて、組み合わせて使うということ。

先ほどの例も、.gridという汎用的なオブジェクトではなく、「画像とテキストがスマホでは縦に、タブレット以降では横に並ぶ」という具体的なオブジェクトの .block-gridを作ったほうが変更しやすい。

オブジェクトの制限を定義する

機能に制限をかける理由

- **無限に拡張**していくことはできない。
- **拡張するごとに複雑さが増す**ので実装が難しくなる。
  「**機能の複雑さ × 機能の数 = 工数**」にはならない。
- 制限をかけなければ、機能だけでなく**工数もかかり続ける**。
- バリエーションが少ないほうが**運用面での負担が少ない**。

もちろん制限すればいいというわけではないので、
デザイナーとエンジニアのコミュニケーションは必須。

特にレスポンシブでは、ブレイクポイントの間を
想像することは難しいので、相談しながら作っていく。

ビジュアルデザインの意図・意味を
コードに反映できたら、
デザインもコードも破綻しない。

06

まとめ

まとめ

- OOCSSからCSSのアンチパターンを学ぶ。
- 2つの原則をもとにオブジェクトとして分離、組み合わせる。
- 役割を定義して機能を限定させる。
- 役割は明確に小さく。
- デザイナーとエンジニアが一緒に作り上げることが
  本当のデザイン（設計）。

ありがとうございました。

slide writing : yasuda manabu
slide design : nakajima eri