柴犬でもわかる FLOCSS

FLOCSSの作者による唯一の解説書

谷 拓樹 著

柴犬でもわかる
FLOCSS

2018/10/2 版 マメヒコファンクラブ 発行

はじめに

本書について

本書は私が考案したCSSの構成案である**FLOCSS**[*1]について解説した書籍です。

FLOCSSは2014年4月にGitHub上に公開し、2018年9月1日現在で574starをいただいています。ただのREADME.mdのファイルひとつのリポジトリですが、多くの方に参考にしていただいているようで、素直に嬉しい気持ちです。現在は有志の方によって翻訳された英語・韓国語版も公開しています。

前述のとおり、FLOCSSについてはひとつのREADME.mdのファイルを公開しているだけで、公開以降に大きな加筆修正はおこなっていません。そのため、多くの方がFLOCSSの解釈に頭を悩ませ、自らのアイデアを加えながら独自の設計を実践している方も多いようです。これは結果として、FLOCSSを強固なルールを持った設計手法というよりは、ひとつの構成のアイデアとして公開したつもりであった自分にとってはよい流れであったと感じています。

FLOCSSは数多あるプロジェクトやチームにそのままフィットするとは限らないため、自分たちでアレンジするのが言ってしまえばFLOCSSのベストプラクティスであるともいえます。とはいえ、ブログ記事やTwitterなどを見回してみると、FLOCSSとしてどうコードを書けばよいのかを悩む人たちが多い現状をみて、技術書典というイベントを機に考案者である私自身でFLOCSSにおける考え方などを公開してみることにしました。

ここで1つ読者の皆さんに伝えたいことがあります。

この書籍では私がFLOCSSを再解釈し、アップデートした内容を公開します。その内容を見ると次のように考える方もいるかもしれません。

> 公開されているドキュメントと説明が違う！

> 自分が理解していたFLOCSSではない！

> 自分のFLOCSSは間違えているから、ドキュメントを修正しないといけない...

どう捉えて実践するのかは皆さん次第ではありますが、知っておいてほしいのは、考案者が語っているからといって、その手法が正しい=すべての問題を解決する、というわけではないということです。繰り返しますが、プロジェクトやチームにあったマイナーチェンジはあって然るべきですので、FLOCSSの考え方をベースにCSSの設計を考える上でのヒントにさえなれば幸いです。

本書で扱わないこと

本書はCSS設計に関する書籍にはなりますが、基本的にはFLOCSSにフォーカスした内容となります。そのためCSSやSassについての基礎、CSS設計についての基本的な説明等は省略しています。

***1** https://github.com/hiloki/flocss

本書での用語の扱い

FLOCSSを構成する要素としてProject（プロジェクト）やComponent（コンポーネント）があるため、文中における「プロジェクト」や「コンポーネント」の表現が混同することがあります。本書では、FLOCSSにおけるProjectを指す場合はアルファベット表記、Webサイトや企画という意味での「プロジェクト」はカタカナで表記しています。

「コンポーネント」についても同様です。本書では「UIコンポーネント」「パーツ」としての単位は基本的に「モジュール」という呼称にしています。世のデザイン手法や、CSS設計手法では「コンポーネント」「コンポーネント設計」と呼ばれることが多いのですが、FLOCSSのComponentと混同しないように「モジュール」に置き換えています。

著者について

筆者の名前は**谷拓樹（たにひろき）**といいます。Web上では**hiloki**というIDでTwitterやGitHubなどのサービスに登録しているので、Web関連、CSS、UIデザインに関心がある方、あるいは私の趣味であるコーヒーの話や愛犬の柴犬についてのことにも興味がある方はぜひフォローしてください。

本書の執筆時は大きな事業会社に属していますが、Web業界に入ってからはベンチャー、スタートアップ、フリーランスなどを経験し、受託・自社事業開発いずれも経験してきました。いくつかの著者があり、本書と関係が強いものですと「Web制作者のためのCSS設計の教科書」[*2]という印象的な緑色の装丁の書籍を執筆しました。この書籍の中でもFLOCSSを少し紹介しています。

本書に関しての感想、フィードバック、あるいは内容についての質問等がありましたら、前述のTwitterにでもメッセージなどをいただければとおもいます。※すべてのご質問等に回答できない場合がありますので、ご了承ください。

謝辞

今回の技術書典への初出展にあたり、色々とアドバイスをいただいた皆さまありがとうございました。

この書籍は、Re:VIEW+CSS組版の環境で作成しています。この環境を公開されている@vvakameさんと、そのfork版を公開されている@at_grandpaさんに感謝します。

表紙や挿絵などのアートワークは妻の@tomo_eにおねがいしました。彼女も「ともえ会」というサークルでJavaScript関連の絵本を出しているので、興味がある方は探してみてください。

また本書「柴犬でもわかるFLOCSS」の柴犬レビュワーとして参加いただいた、愛犬のマメヒコにも大きな感謝をしています。

***2** 「Web制作者のためのCSS設計の教科書」出版社：インプレス https://book.impress.co.jp/books/1113101128

目次

第1章

FLOCSSの概要

FLOCSSの基本原則については、GitHubですべて公開していますが、本書でもFLOCSSの特長などを解説します。

1.1　影響を受けたCSS設計手法

FLOCSSは、公開当時私がCSS設計を学んでいたときに参考にしていたCSS設計手法などを参考に構築したアイデアです。

ちなみに**FLOCSS**は「**フロックス**」と呼びます。「フロシーエスエス」と呼ばれることもありますが、公式呼称としては「フロックス」です。

CSS設計としての基本理念は、OOCSS[*1]。やSMACSS[*2]の影響を強く受けています。

これらの設計手法を学ぶことで、デザインカンプを「面」で解釈して組み上げていくのではなく、UIの構造や装飾などを抽象化して「点」や「線」で考えることができるようになりました。再利用性や拡張性を意識しながら、モジュールの単位でのマークアップやCSSを書く、というのがFLOCSSにおける基本的な思想になります。

セレクタの命名規則は、BEM[*3]のBlock、Element、Modifierという構造を命名に反映したルールに基づいています。

FLOCSSにおける「レイヤー」という概念や、その依存関係の考え方については、MCSS[*4]の影響を強く受けています。

***1** http://oocss.org/

1.2　基本原則

FLOCSSは**Foundation**、**Layout**、**Object**と、Objectレイヤーのサブレイヤーである**Project**、**Component**、**Utility**で構成されます。（表1.1）

▼表1.1: FLOCSSのレイヤー構成

レイヤー	スタイル、モジュール例
Foundation	reset, normalize, base...
Layout	header, main, sidebar, footer...
Object/Component	button, form, avatar...
Object/Project	articles, ranking, promo...
Object/Utility	clearfix, display, margin...

これらは「カテゴリ」と呼んでも構わないのですが、これらが階層化されていることにFLOCSSとしての理由があるので、MCSS同様に「レイヤー」と呼んでいます。なおFLOCSSの由来は、**F**oundation、**L**ayout、**O**bjectの頭文字をとってFLOCSSとしました。

1.2.1　Foundation

Reset.cssやNormalize.cssなどを用いたブラウザのデフォルトスタイルの初期化や、プロジェクトにおける基本的なスタイルを定義します。 ページの下地としての全体の背景や、基本的なタイポグラフィなどが該当します。Sassなどのツールを使っている場合には、変数 、MixinやFunctionのコードもFoundtionに定義してもよいでしょう。

***2** https://smacss.com/ja

***3** http://bem.info/

***4** http://operatino.github.io/MCSS/ja/

1.2.2 Layout

ページを構成するヘッダーやメインのコンテンツエリア、サイドバーやフッターといったプロジェクト共通のコンテナーブロックのスタイルを定義します。

公式ドキュメントでは、ヘッダーやフッターといった要素が基本的にページ単位でユニークな要素であることを踏まえて、IDと属性セレクタの活用を提案していますが、今の私の考え方としては、`l-*`という接頭辞をつけて、classとして扱うほうがよいと考えています。

理由としては、他のレイヤーがclassであることも踏まえ、一貫してしまうほうが意識すべきことが減るためです。このLayoutレイヤーについては、本書の第3章「Layoutの扱い」で解説していますので、そちらも参考にしてください。

1.2.3 Object

OOCSS（Object Oriented CSS ＝ オブジェクト指向CSS）のコンセプトを元に、プロジェクトにおける繰り返されるビジュアルパターンをすべてObjectと定義します。

Component

パターンとして再利用できる汎用的なモジュールが定義されるレイヤーです。

Webサイトを構築するにあたってほぼ必ず使われるような汎用的な要素をイメージしてください。たとえば、HTMLのプリミティブな要素であるフォームのテキストフィールド（`<input type="text">`など）や、ボタンなどが該当します。他にもCSSフレームワークであるBootstrapのComponent[*5]にある

モジュールの粒度が、そのままFLOCSSにおけるComponentとほぼ同等だと考えてよいでしょう。

Project

プロジェクト固有のパターンが定義されるレイヤーです。

たとえば、記事一覧や、ユーザープロフィール、画像ギャラリーなどコンテンツを構成する要素などが該当します。Projectのモジュールは、定義されたBlockとElementで完結することも多いのですが、一部の要素をComponentが担うこともあります。（コード1.1）

▼コード1.1: Projectの例

```
/* p-promoのみで完結 */
<section class="p-promo">
  <h2 class="p-promo__title">スペシャルセール情報！</h2>
  <div class="p-promo__banner">
    <a href="/sale">
      <img src="banner.png" alt="歳末セール特設ページ">
    </a>
  </div>
</section>

/* p-articleとc-buttonの組み合わせ */
<article class="p-article">
  <header class="p-article__header">
    <h1 class="p-article__title">記事タイトル</h1>
  </header>
  <div class="p-article__body">...</div>
  <div class="p-article__footer">
    <button class="c-button p-article__commentButton">
      コメントする
    </button>
  </div>
</article>
```

***5** http://getbootstrap.com/docs/4.1/components/alerts/

Utility

Utilityは、ComponentとProjectで解決することが難しい、例外的にわずかなスタイル調整が必要な場合などに使えるヘルパークラスを定義します。具体的な例を挙げると、`.u-block { display: block; }`のようなコードです。あまり多用されるべきレイヤーではないのですが、プロジェクトによっては必要となることもあるため、Objectのひとつとして用意しています。

1.3　命名規則と記述順

1.3.1　接頭辞

各レイヤーに分類されたモジュールは、それぞれ属するレイヤーを表す接頭辞を名前の先頭につけることを推奨しています。

▼コード1.2: 接頭辞ルール

```
Layout: .l-*
Component: .c-*
Project: .p-*
Utility: .u-*
```

FLOCSSでは、そのモジュールがどのレイヤーに属しているかを理解することが重要であるため、こうした命名を推奨しています。しかし、プロジェクトやチームによって命名規則をアレンジしたい場合は、そのルールに一貫性があれば問題ないと考えています。命名規則については第4章「FLOCSSの命名規則」も参考にしてください。

1.3.2　BEM記法

　すべてのモジュールはBEMの構造にならい、Block、Element、Modifierをあらわす名前をベースにしています。（図1.1、コード1.3）

▲図1.1: BEMの記法を元にした名前

▼コード1.3: BEMの記法を元にした名前:コード例

```
/* Layout */
.l-header { ... }

/* Component */
.c-button { ... }
.c-button__icon { ... }
.c-button--primary { ... }

/* Project */
.p-article { ... }
.p-article__header { ... }
.p-article--ad { ... }

/* Utility */
.u-block { ... }
.u-mbs { ... }
```

1.3.3　モジュールの記述順

基本原則のレイヤー構成に従い、次のような順番でCSSを記述します。（コード1.4）

▼コード1.4: FLOCSSの コード構成:Foundation、Layout

```
/**
 *  Foundation
 * ========================== */

/* Reset */

html, body, div, ... { ... }

/* Base */

body { ... }

/**
 *  Layout
 * ========================== */

/* Header */

.l-header { ... }

/* Main */

.l-main { ... }
```

Objectレイヤーの場合は次のようなコードです。

▼コード1.5: FLOCSSの コード構成:Object

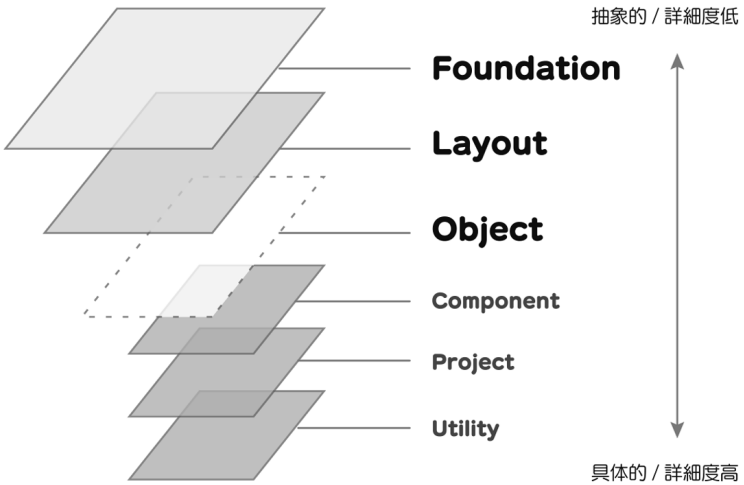
```
/* Object
 * ========================== */

/* Component ------------ */

/* Button */

.c-button { ... }
.c-button--primary {
  ...
}

/* Project ------------ */

/* Articles */

.p-articles { ... }
.p-article {
  ...
}
.p-article__title {
  ...
}

/* Utility ------------ */

/* Display */

.u-block {
  ...
}
```

FLOCSSにおいては、このような順番で記述することに理由があります。

1.4　カスケーディングと詳細度

カスケーディングは、CSSのもっとも大きな特長のひとつです。運用の中でCSSが破綻していくのは、安易なカスケーディングとセレクタの管理に原因があることが多いでしょう。

FLOCSSでは、後ろのレイヤーになるほど具体的になり、カスケーディングにおいて強いルールである必要があります。そのため、Componentの前にProjectのモジュールのスタイルが宣言されているなど、レイヤーの順序を前後させてはいけません。(図1.2)



▲ 図1.2: FLOCSSのレイヤーイメージ

前述に記載したコード例（コード1.5）のように記述の順番を守っていれば「ボタン」というComponentレイヤーの `.c-button` モジュールを、コード上で後述された「特定の幅や文字サイズの大きな登録ボタン」というProject

レイヤーの`.p-register__button`によって自然と上書きできるようになります。（コード1.6、コード1.7）

▼コード1.6: ComponentとProject:HTML

```
<button class="c-button p-register__button">登録</button>
```

▼コード1.7: ComponentとProject:CSS

```
.c-button {
  display: inline-block;
  padding: 0.5em 1em;
  border: 0;
  border-radius: 4px;
  border: 1px solid #666666;
  background-color: #FFFFFF;
  color: #666666;
  vertical-align: middle;
  text-decoration: none;
  white-space: nowrap;
  font-size: inherit;
  cursor: pointer;
}

.p-register__button {
  width: 360px;
  font-size: 24px;
  background-color: #339922;
  color: #FFFFFF;
}
```

これらのComponent、Projectの定義や扱いについては第2章「ComponentかProjectか」の章で詳解しているので、そちらも参考にしてください。

1.4.1　同じレイヤーのモジュールの扱い

Componentのモジュール同士、Projectのモジュール同士でセレクタを混在することは避けるようにしてください。（コード1.8）

▼コード1.8: モジュール間の依存関係

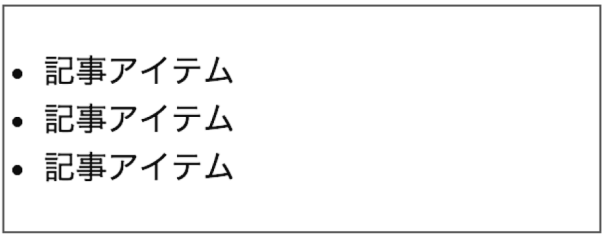
```
/* これらはすべて
 * FLOCSSとしてはアンチパターンとなる
 */

.c-button .c-icon { ... }

.p-entryList .p-features__list { ... }
```

複数のモジュールが結びつくセレクタにしてしまうと、その間に依存関係が生まれます。そうなると一方のモジュールの名前やスタイルが変わってしまったときに、その影響を受けることになります。そのため原則的にはこのようなセレクタは避けましょう。

また1つの要素に対し、同じレイヤーのモジュールを混ぜることも推奨しません。（コード1.9）混ぜてしまうことによって、次のコード例でいえば、`.p-entryList`が先にスタイルを宣言するか、`.p-features__list`が先にスタイルを宣言するかで最終的なスタイルが異なります。（コード1.10、コード1.11）

▼コード1.9: 同じレイヤーの複数のモジュールを混ぜた例

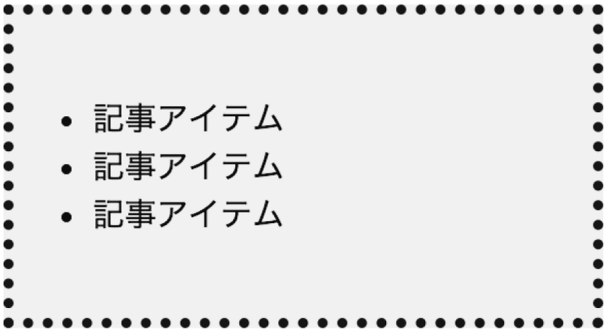
```
/*
 * 非推奨なマークアップ例
 */
<ul class="p-entryList p-features__list">
  <li>...</li>
</ul>
```



▲図1.3: .p-features__listが優先:図

▼コード1.10: .p-features__listが優先:CSS

```
.p-entryList {
  border: 1px solid #FF0000;
  padding: 20px;
}

.p-features__list {
  border: 5px dotted #000880;
  padding: 40px;
}
```



▲ 図1.4: .p-entryListが優先:図

▼コード1.11: .p-entryListが優先:CSS

```
.p-features__list {
  border: 5px dotted #000880;
  padding: 40px;
}

.p-entryList {
  border: 1px solid #FF0000;
  padding: 20px;
}
```

同じマークアップ（コード1.9）でも、スタイルの記述順で結果が変わってしまいます。FLOCSSとしては同じレイヤー内でのモジュールの読み込み順に依存しないようにしたいため、このような実装は避けるほうが懸命です。なおComponentとProjectのように異なるレイヤーの場合にはまた違う考え方もあるので第2章「ComponentかProjectか」の「意図的なカスケーディング」もあわせて参考にしてください。

1.5　Blockごとの分割とインポート

もしSassようなCSSプリプロセッサ、その他ビルドツールを使って、CSSファイルを分割してインポートと結合できる環境にあれば、次のようにディレクトリとファイルを分割して管理することを推奨します。次の例は、Sassを採用した場合の例です。（コード1.12）

▼コード1.12: Blockごとに分割する

```
├── foundation
│   ├── _reset.scss
│   └── _base.scss
├── layout
│   ├── _footer.scss
│   ├── _header.scss
│   ├── _main.scss
│   └── _sidebar.scss
└── object
    ├── component
    │   ├── _avatar.scss
    │   ├── _button.scss
    │   ├── _dialog.scss
    ├── project
    │   ├── _articles.scss
    │   ├── _comments.scss
    │   ├── _gallery.scss
    │   └── _profile.scss
    └── utility
        ├── _display.scss
        ├── _align.scss
        └── _margin.scss
```

モジュール（Block）単位でディレクトリ、ファイルを分割することによって、モジュールの追加・削除の管理がしやすくなります。

これらの分割したファイルを管理、インポートするためのapp.scssのようなファイルからは次のように参照します。（コード1.13）

▼コード1.13: Sassでのimport例

```
/**
 *  Foundation
 * ========================== */

@import "foundation/_reset";
@import "foundation/_base";

/**
 *  Layout
 * ========================== */

@import "layout/_footer";
@import "layout/_header";

/* Object
 * ========================== */

/* Component ------------ */

@import "object/component/_avatar";
@import "object/component/_button";

/* Project ------------ */

@import "object/project/_articles";
@import "object/project/_comments";

/* Utility ------------ */

@import "object/utility/_align";
@import "object/utility/_margin";
```

1.5.1　globによるインポート

分割されたファイルは**glob**を使ったインポートを使うと便利です。

globとは、ワイルドカード（`*`）を用いたファイル名を特定するパターンです。具体的な例を用いて補足すると、`@import "object/project/**"`のように参照すれば、`object/project/`ディレクトリ配下のファイルをすべて一括してインポートすることができます。

モジュールごとにファイルを分割し、個別にインポートをするとなると、ひとつひとつ追記していくことが億劫になります。そのときにこのglobをつかえば、ファイルが増えたとしても、インポートを管理するファイルに追記する必要がなくなるということです。

globパターンについて知っておかないといけないのが、globパターンを使うと、ファイルシステムでのファイル名の降順でインポートされるということです。これはつまり、OSのフォルダでのファイル名でソートされたときの順番だと考えてください。

たとえば`avatar.scss`と`button.scss`というファイルがあれば、`avatar.scss`が先にインポートされます。

そのため、同じディレクトリに並列でファイルを展開してしまうと、ファイル名すなわちモジュール名によってインポートされる順番が代わり、結合後のファイル上での記述順にも影響があります。

そこで、前述の分割時の構成（コード1.12）のように、まずはディレクトリで分割し、ディレクトリごとにglobでインポートします。（コード1.14）

▼コード1.14: globパターンを使ったimport例

```
/**
 *  Foundation
 * ========================== */

@import "foundation/_reset";
@import "foundation/_variables";
@import "foundation/_function";
@import "foundation/_mixin";
@import "foundation/_base";

/**
 *  Layout
 * ========================== */

@import "layout/*";

/* Object
 * ========================== */

/* Component ------------ */

@import "object/component/*";

/* Project ------------ */

@import "object/project/*";

/* Utility ------------ */

@import "object/utility/*";
```

インポートの宣言がかなり減りました。ポイントは、FLOCSSのレイヤーの順番を守るために、Foundationのディレクトリ、Layout、ObjectのProject、Utilityのディレクトリ単位でのglobにしています。

このコードで気づいた人もいるかもしれませんが、Foundationレイヤーに関してはglobは推奨しません。Foundationでは変数やSassのmixinなどを管理するファイルも分割することがあるのですが、それらのファイルは読み込み順への依存関係がとても強いファイルです。そのためglobに委ねるのではなく、個別でインポートするようにしています。FoundationレイヤーはObjectレイヤーのようにファイルが増え続けることは無いので、手間がかかるということも無いでしょう。

1.6　あくまでも基本原則

FLOCSSについて、公開されているドキュメントを元に、補足説明などを加えて解説しました。本章以後は、FLOCSSを採用するにあたって悩むポイントの解説や、FLOCSSを元にした設計手法の考え方について紹介します。

第2章

ComponentかProjectか

FLOCSSを扱う上でよく疑問に出るのが、対象の要素がComponentなのかProjectなのかということです。あらためてComponentとProjectそれぞれの定義を公式ドキュメントから引用しつつ、補足を加えていきましょう。

2.1　Component、Projectの定義

公式ドキュメントでのComponent、Projectそれぞれの定義を引用します。

Componentの定義

> 再利用できるパターンとして、小さな単位のモジュールを定義します。一般的によく使われるパターンであり、例えばBootstrapのComponentカテゴリなどに見られるButtonなどが該当します。出来る限り、最低限の機能を持ったものとして定義されるべきであり、それ自体が固有の幅や色などの特色を持つことは避けるのが望ましいです。

Projectの定義

> プロジェクト固有のパターンであり、いくつかのComponentと、それに該当しない要素によって構成されるものを定義します。例えば、記事一覧や、ユーザープロフィール、画像ギャラリーなどコンテンツを構成する要素などが該当します。

この定義だけでは対象のモジュールがどのレイヤーであるのかを判断するのが難しいと感じるかもしれないので、この考え方を基準に補足と具体例を加えていきます。

基本的に多くのWebプロジェクトにおいては、ほとんどのモジュールがProjectレイヤーに属すると考えて構いません。その中で、パターンとして抽象化できるものがあれば、それはFLOCSSのComponentレイヤーに属するものであると考えます。

たとえば、アバター（ユーザーのアイコン）画像のような要素があるUIが数点存在するとします。プロフィールエリア、記事のエリア、コメントのエリアそれぞれにアバター画像表示エリアがあるのを想像してください。（図2.1、コード2.1）

▲ 図2.1: アバター画像の例

▼コード2.1: アバター画像のCSS

```
.p-profile__avatar {
  display: inline-block;
  border-radius: 50%;
  border: 1px solid black;
  width: 128px;
  height: 128px;
}

.p-article__avatar {
  display: inline-block;
  border-radius: 50%;
  border: 1px solid black;
  width: 64px;
  height: 64px;
}

.p-comment__avatar {
  display: inline-block;
  border-radius: 50%;
  border: 1px solid black;
  width: 48px;
  height: 48px;
}
```

　コードを見ると、同じようなプロパティが宣言されているのがわかります。このWebプロジェクトにおいて、アバターは基本的に円形にトリミングされたような表現となっているようです。

そうすると次のようにリファクタリングすることができそうです。（コード2.2、コード2.3）

▼コード2.2: リファクタリング後のCSS

```
/* Component */
.c-avatar {
  display: inline-block;
  border-radius: 50%;
}

/* Project */
.p-profile__avatar {
  border: 1px solid black;
  width: 128px;
  height: 128px;
}

.p-article__avatar {
  border: 1px solid black;
  width: 64px;
  height: 64px;
}

.p-comment__avatar {
  border: 1px solid black;
  width: 48px;
  height: 48px;
}
```

▼コード2.3: HTML

```
<img src="article-avatar.png" class="c-avatar p-profile__avatar">
<img src="article-avatar.png" class="c-avatar p-article__avatar">
<img src="article-avatar.png" class="c-avatar p-comment__avatar">
```

リファクタリング後のコードをみてみると、`border: 1px solid black;`も共通で使われているスタイルとして存在しています。ならばこのスタイルも`.c-avatar`に持たせよう、と考えたくなりますが、ここで公式のドキュメントのComponentの定義をみると、次のような記述があります。

> できる限り、最低限の機能を持ったものとして定義されるべきであり、それ自体が固有の幅や色などの特色を持つことは避けるのが望ましいです。

このアバターの例では、`border: 1px solid black;`というのが装飾のスタイルなので、Componentレイヤーのモジュールに持たせるスタイルとしては不適切にみえます。素直に従うならば、前述のコード2.2のように、このボーダーのスタイルはComponentの`.c-avatar`ではなく、`.p-profile__avatar`などのProjectレイヤーに持たせるほうが適切であるということになります。

ですが、もしデザインとして「必ずアバター画像には枠線がつく」というのがほぼ約束されているのであれば、`.c-avatar`に`border: 1px solid black;`があったとしても構わないと考えています。ドキュメントのComponentの説明にある「特色をもつことは避ける」には反しますが、一方で「できる限り」と書いているように、あまり「特色をもつことは避ける」に引きづられる必要もないと考えています。このあたりは実際にコードを書いていく上での効率などを考えたときに、毎度同じ装飾の宣言をProjectレイヤーのモジュールに書き続けているのであれば、Componentレイヤーにそれらの装飾の宣言も移しても構わないでしょう。

2.2　意図的なカスケーディング

FLOCSSであれば、Componentレイヤーに装飾のスタイルがあったとしても、必要あればProjectレイヤーで上書きすることを許容しています。

仮に`.c-avatar`にデフォルトの枠線のスタイルとして`border: 1px solid black;`が宣言されているとして、もし文脈に応じて枠線の装飾に変更を加えなければならない場合があればどうでしょうか。

例えば、大きなサイズで表示される`p-profile__avatar`に対して、太い`border-width`にしたいというような場合は次のようにコードを記述するだけです。（図2.2、コード2.5、コード2.4）

▼コード2.4: HTML

```
<img src="article-avatar.png" class="c-avatar p-profile__avatar">
```

▼コード2.5: CSS

```
.c-avatar {
  display: inline-block;
  border-radius: 50%;
  border: 1px solid black;
}

.p-profile__avatar {
  border-width: 10px;
  width: 128px;
  height: 128px;
}
```

▲図2.2: 一部のアバター画像の枠線を太くする

CSSのカスケーディングによって、後述された宣言で上書きされているというだけの例ですので、特に問題もなく、FLOCSSとしても適切な例といえるでしょう。

しかし、次のようにComponentとProjectの記述順が変わってしまえば、期待とは異なる見た目になってしまいます。（図2.3、コード2.6）

▼コード2.6: CSS

```
.p-profile__avatar {
  border-width: 10px;
  width: 128px;
  height: 128px;
}

.c-avatar {
  display: inline-block;
  border-radius: 50%;
  border: 1px solid black;
}
```

▲ 図2.3: 枠線に変化がない

これがCSSの容易さであり、一方で破綻しやすさでもある特長です。FLOCSSでは、CSS（Cascading Style-Sheet）らしくカスケーディングをあえて許容しており、FLOCSSで定義しているレイヤーの関係性を厳密にすることで、安全にカスケーディングされるよう設計しています。

> FLOCSSでは、後ろのレイヤーになるほど具体的になり、カスケーディングにおいて強いルールである必要があります。そのため、レイヤーの順序を前後させてはいけません。

つまり前述の例でいえば、コードの記述順は Component の次に Projectとする必要があるため、コード2.6に例として挙げた `.c-avatar` が `.p-profile__avatar` より後ろに記述されるのはFLOCSSとしては不正なコードということになります。

2.3　まずProjectレイヤーから

ComponentかProjectかという本題についてまとめると、

1. 基本的にProjectレイヤーのモジュールとして考える
2. その中で数度繰り返されたパターンをComponentとして切り出す

これが基本的な考え方となります。

またFLOCSSでは、ProjectがComponentのモジュールのスタイルを補助や上書き（カスケーディング）することを許容しているので、必要に応じてそれを前提としたルールやガイドラインを設けてみてください。

はじめからComponentは定義できないか？

完全なデザインカンプデータを受け取り、そのカンプを見ながらコーディングをしていく工程だと難しいかもしれませんが、はじめからUIモジュールの粒度をすり合わせることができるなら、Componentをあらかじめ定義することができます。

しかし残念ながら現場では、カンプから意図や再利用できる「かも」というのを汲み取るしかないこともあるでしょう。できるかぎり、デザイナーとUIの意図やパターンの認識合わせをできるのが理想的です。

第3章

Layoutの扱い

LayoutはFLOCSSという名前を支える重要なレイヤーです。しかし、FLOCSSのユーザーの声を拾い上げてみると「どう使えばいいか悩ましい」「Layoutは必要ないのではないか」という内容を耳にします。私自身もそのように考えたことはあります。改めてLayoutレイヤーの定義や、その活用について思うところはありますので、この章で解説していきます。

3.1　Layoutレイヤーの定義

FLOCSSのドキュメントにおいて、Layoutは次のように定義されています。

> ページを構成するヘッダーやメインのコンテンツエリア、サイドバーやフッターといったプロジェクト共通のコンテナーブロックのスタイルを定義します。

ページの構造として大きな枠組み、ストラクチャなどとも呼ばれる要素がFLOCSSにおけるLayoutの定義です。

Layout（レイアウト）という言葉に引きづられてしまうと、すべてのモジュールにいける配置（レイアウト）をすべてLayoutレイヤーが担うのか、と思われる方もいます。

どういうことかというと、レイアウトに関するスタイルはObjectレイヤーのComponent/Projectのモジュールには持たせず、Layoutレイヤーのモジュールに持たせる、という考え方です。（コード3.1、コード3.2）

▼コード3.1: Layoutが過剰な例:CSS

```
/*
 * Layout
 * 配置や幅などに関するスタイル
 */

.l-article {
  margin-bottom: 64px;
  width: 640px;
}

.l-article__header {
  margin-bottom: 24px;
}

.l-article__body {
  width: 520px;
}

/*
 * Project
 * 装飾や内側余白のスタイル
 */

.p-article {
  border: 1px solid #666;
  padding: 24px;
}

.p-article__title {
  font-size: 24px;
}
```

▼コード3.2: Layoutが過剰な例:HTML

```
<aritcle class="l-article p-article">
  <header class="l-article__header">
    <h1 class="p-article__title">記事タイトル</h1>
  </header>
  <div class="l-article__body">
    ...
  </div>
</aritcle>
```

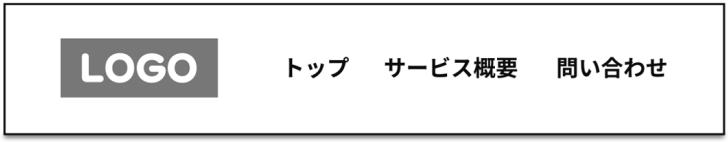
FLOCSSとしては、あくまでもヘッダーやフッターといったページの大きな構造の要素を対象としているので、ひとつひとつのモジュールごとにLayoutレイヤーを用意するわけではありません。つまりコード3.1、コード3.2のコード例は、FLOCSSとしては適切ではないといえます。

しかし、ここで補足したいのは、FLOCSSをベースにCSS設計を考えたときに、Layoutを再定義し、役割・責務を変えてしまってもよいということです。その例については、本章の後半で説明します。

3.2 Layoutのユースケース

FLOCSSにおけるLayoutの定義は理解したとしても、実際にどう使うのかという疑問もあります。特に悩ましく感じるのは「ヘッダー」という要素があるときではないでしょうか。

ヘッダーの例として、よくあるWebサイトヘッダーの構成をあげます。ロゴとナビゲーションを抱えたヘッダーです。（図3.1）



▲ 図3.1: よくあるサイトヘッダー

ヘッダーなのでLayoutと定義するなら、どのようにマークアップするのでしょうか。

たとえば次のようなマークアップが考えられます。（コード3.3）

▼コード3.3: よくあるサイトヘッダー:HTML

```
<header class="l-header">
  <h1 class="l-header__logo">
    <img src="logo.png" alt="サイト名" class="p-logo">
  </h1>
  <div class="l-header__nav">
    <nav class="p-nav">...</nav>
  </div>
</header>
```

ヘッダーの配置から、ヘッダー内のロゴやナビゲーションの配置までを担うモジュールとして、実にLayoutらしい例かもしれません。ではこのマークアップを前提に、ヘッダーとしての装飾は`.l-header`が持つのでしょうか？（コード3.4）

▼コード3.4: Layoutが装飾を持つ:CSS

```
.l-header {
  max-width: 1024px;
  width: 100%;
  padding: 24px;
  border: 1px solid #666666;
  background-color: #FFFFFF;
  box-shadow: 0 0 2px #000;
}
```

このCSSでも見た目をつくる上では成立していますが、FLOCSSとしては基本的にLayoutを装飾でかためることは基本的に避けたいところです。

今度はLayoutレイヤーをやめて、Projectレイヤーのモジュール `.p-header` でマークアップをした場合はどうでしょうか。（コード3.5、コード3.6）

▼コード3.5: Project としてのヘッダー:HTML

```
<header class="p-header">
  <h1 class="p-header__logo">
    <img src="logo.png" alt="サイト名" class="p-logo">
  </h1>
  <div class="p-header__nav">
    <nav class="p-nav">...</nav>
  </div>
</header>
```

▼コード3.6: Projectとしてのヘッダー:CSS

```
.p-header {
  max-width: 1024px;
  width: 100%;
  padding: 24px;
  border: 1px solid #666666;
  background-color: #FFFFFF;
  box-shadow: 0 0 2px #000;
}
```

class名の接頭辞が変わっただけなので、これでも見た目は成立します。Layoutに装飾のスタイルを持たせることが問題となるようであれば、Projectとしてしまえば良さそうに見えます。

ではFLOCSSにおいてLayoutが必要でないかといえば、そういうわけではありません。モジュール、コンポーネント設計の考え方としてレイアウトを分離する方法を考えてみましょう。

LayoutのないFLOCSS

本章においてLayoutの必要性を説いていますが、考案者である私自身でも、Projectでまかなってしまえばは不要だと考えないわけではありません。そのためFLOCSSのLはLayoutのLであるので、**FOCSS**として新しく定義し直すことを考えたこともあります。

FOCSSは未公開のままではありますが、本章を参考にしていただいた上で、やはりLayoutは不要だと感じたなら、FLOCSS、改めFOCSSのような設計にするのも良いでしょう。

3.2.1　LayoutとProjectの使い分け

具体的なコード例をまず挙げてみます。（コード3.7、コード3.8）

▼コード3.7: LayoutとProjectの組み合わせ:HTML

```
<header class="l-header">
  <div class="p-header">
    <h1 class="p-header__logo">
      <img src="logo.png" alt="サイト名" class="p-logo">
    </h1>
    <div class="p-header__nav">
      <nav class="p-nav">...</nav>
    </div>
  </div>
</header>
```

▼コード3.8: LayoutとProjectの組み合わせ:HTML

```
.l-header {
  max-width: 1024px;
  width: 100%;
}

.p-header {
  padding: 24px;
  border: 1px solid #666666;
  background-color: #FFFFFF;
  box-shadow: 0 0 2px #000;
}
```

レイアウトのスタイルは`l-header`に持たせ、サイトヘッダーとして装飾のスタイルは`p-header`に分離させています。

デザインによっては`l-header`が一切のスタイルを持つ必要がない場合があるかもしれません。しかし、私の場合は、そのページに構造があれば、それ

ぞれにLayoutのclassを振ります。ヘッダーがあれば、ページの構成の中でヘッダーのレイアウトを役割を持つ要素として、クラスを与えます。（コード3.9）

その理由としては、マークアップをみたときに構造が把握しやすいということです。またclass属性には「コンテンツの性質をあらわす」という仕様があるので、スタイルを持たなくとも、意味を持たせるために付与してもよいだろうと考えているからです。

▼コード3.9: Layoutのクラスを持つ

```
<div class="l-header">...</div>
```

3.3　再利用性のあるモジュールとしてのLayout

FLOCSSにおけるLayoutレイヤーはページの構造に関わる要素に対するもの、グリッドレイアウトのように、再利用性のあるレイアウト機能をもったものはどのように解釈するのがよいでしょうか？

公式ドキュメントではComponentとして`c-grid`のようなモジュールにするのがよいとしていますが、Layoutとしても悪くはないでしょう。

注意したい点としては、前節のようなページの構造を担う役割と混同すると、Layoutの定義が曖昧になってしまう点です。この場合、いっそ再利用性のあるレイアウト機能を持ったモジュールを抱えるレイヤーとして、Layoutを定義してしまうほうがよいかもしれません。（コード3.10）

▼コード3.10: LayoutとしてのGridモジュール

```
.l-grid {
  display: flex;
  flex-wrap: wrap;
}

.l-grid__item {
  box-sizing: border-box;
  flex-shrink: 0;
}

.l-grid__item--1of12 {
  width: calc(100% * 1 / 12);
}
```

FLOCSSとしての定義とは異なりますが、それがプロジェクトやチームにとって適切なルールになりそうなのであれば、再定義することは正しい選択です。

3.4 余白調整のLayout

実験的なアプローチとして、モジュール間の余白を制御する、`l-spacer`のようなモジュールを扱うというものです。これはCSSによるレイアウトがまだ定着していない時代における、`spacer.gif`を使ったアプローチ、と言いかえれば理解できる人もいるかもしれません。`l-spacer`はただの空のdivで、その要素自身に`margin`のルールを持つものです。モジュール間に余白が必要であれば、`l-spacer`を持つ要素を挿入するということです。（コード3.11、コード3.12）

▼コード3.11: LayoutとProjectの組み合わせ:HTML

```
<!-- 記事 -->
<article class="p-article">...</article>

<!-- 余白 -->
<div class="l-spacer-md"></div>

<!-- 関連記事リスト -->
<div class="p-relatedArticles">...</div>
```

▼コード3.12: LayoutとProjectの組み合わせ:HTML

```
.l-spacer-md {
  margin-bottom: 24px;
}

.l-spacer-lg {
  margin-bottom: 48px;
}
```

`div`がセマンティクス的に意味を持たないとはいえ、少し違和感を感じるようなマークアップかもしれません。しかしモジュールの配置が入れ替わりやすいようなデザインなのであれば、モジュールそのものに`margin`をもたせたり、あるいはclassの取り外しをするよりも利便性が高いかもしれません。私としては、この`l-spacer`というモジュールを実際のサービス開発で使いこなした経験はまだないので、強くおすすめはしませんが、有用だと感じられたなら検討してみてください。

第4章

FLOCSSの命名規則

FLOCSSのルールにあわせたフォルダ構成や、セレクタの命名規則が存在します。ルールを機能させるための命名ではありますが、そのルールさえ守られていれば、何と命名するかについては、利用者が決めても構いません。

4.1　レイヤーごとの命名規則

どのレイヤーに属する要素かを判別しやすいようにと、`l-` や `p-`、`c-` といった接頭辞をつけることを公式ドキュメントでは例としてあげていますが、必ずしもこれらの文字を使った接頭辞である必要はありません。重要なことはレイヤーを判別できるかどうかどうかです。

たとえば、ComponentとProjectの区別においては「Componentの場合は頭文字を大文字にする」「Projectの場合は頭文字を大文字にしない」というルールがあってもよいでしょう。（コード4.1）

▼コード4.1: 接頭辞を使わない例

```
// # Component
// Componentの場合は頭文字を大文字にする
.Button { ... }
.Card { ... }

// # Project
// Projectの場合は頭文字を大文字にしない
.userList { ... }
.entryForm { ... }
```

接頭辞を付与する煩わしさが減り、レイヤーの判別もできるので、接頭辞をつけることが気になる人にはよいかもしれません。この場合、ComponentとProjectについては接頭辞なしでもよいのですが、UtilityやLayoutにはそれぞれ`u-`や`l-`をつけるようにするのがよいでしょう。あまり命名のパターンが増えると複雑化し、学習が難しくなるので、シンプルに保つことを意識してください。

4.2 Modifierのパターン

BEMの記法にならい、Block との関係を明確にするために、ElementやModifierにはBlock名を継承する形での記述をFLOCSSのドキュメントでは推奨しています。（コード4.2、コード4.3）

▼コード4.2: 基本的なFLOCSSの記法:CSS

```
.p-article {...} /* Block */
.p-article__header {...} /* Block + Element */
.p-article--sponsored {...} /* Block + Modifier */
```

▼コード4.3: 基本的なFLOCSSの記法:HTML

```
<article class="p-article p-article--sponsored">
  <header class="p-article__header">...</header>
</article>
```

見てのとおり、マークアップから構造・依存関係がわかりやすい点は素晴らしいのですが、コードを書く上では冗長さがあることも否めません。そこで私の場合はModifierに関してはBlock、Elementを省略し、`-modifier`というように、ハイフンひとつを接頭辞としてつける運用をすることが増えています。（コード4.4、コード4.5）

▼コード4.4: Modifierの命名規則を変更:CSS

```
.p-article {...} /* Block */
.p-article__header {...} /* Block + Element */
.p-article.-sponsored {...} /* Block + Modifier */
```

▼コード4.5: Modifierの命名規則を変更:HTML

```
<article class="p-article -sponsored">
  <header class="p-article__header">...</header>
</article>
```

この記法で考えられる問題としては、１つの要素に複数のBlockやElementがあった場合に、そのModifierがどちらのModifierなのかが分からないということでしょう。（コード4.6）

▼コード4.6: ModifierがどちらのBlockに依存するのかがわからない

```
<div class="p-entryList p-relatedEntries -wide">...</div>
```

仮にこの状態が発生したならば、Modifierと依存するBlock、Elementを修正するときに少し悩むかもしれませんが、修正コストとしてあまり大きな影響はないのではと考えています。経験上、ひとつの要素に複数のBlockやElementが入り混じる設計になることがほとんど無いということもありますが、この記法でデメリットを感じたことはありません。

ただ設計によって、同じ要素に複数のBlock、Elementが交じるようなパターンが多い、あるいは問題が発生するのであれば、オリジナルのFLOCSS、BEMのように、Block、Element名をModifierに付与する命名のほうが安全でしょう。

あるいは、記法を用いた回避方法としては、class属性の記述を工夫する手段もあります。採用するかどうかは別として、ひとつのアイデアとして知っておくとよいでしょう。（図4.1）

▲ 図4.1: 順番や半角スペースお空け方 で工夫する

複数のBlockとElementの問題

1つの要素に複数のBlockとElement、特に同じレイヤーのモジュールがコード4.5のように重なる場合は、別の問題に注意してください。

例のように同じProjectレイヤーのモジュールが混ざっていると、そのモジュールのCSSの記述順（カスケーディングの影響）によって適用されるスタイルが異なってしまう問題があります。

この点については第1章「FLOCSSの概要」の「同じレイヤーのモジュールの扱い」も参照してください。

4.2.1　ハイフンからはじまる場合は1つにする

もう一点補足すべき点としては、ハイフンが2つから始めた名前ではなく、ハイフン1つであることです。これはHTMLではなくCSSのセレクタの仕様として、ハイフン2つから始まるセレクタは許容されていないためです。[*1]Block、

Element名を省略するとして、どういう命名であるかはある程度自由ですが、セレクタの仕様としての制限はあるので注意してください。

> In CSS, identifiers (including element names, classes, and IDs in selectors) can contain only the characters [a-zA-Z0-9] and ISO 10646 characters U+00A0 and higher, plus the hyphen (-) and the underscore (_); they cannot start with a digit, two hyphens, or a hyphen followed by a digit. Identifiers can also contain escaped characters and any ISO 10646 character as a numeric code (see next item). For instance, the identifier "B&W?" may be written as "B\&W\?" or "B\26 W\3F".

4.2.2 省略したModifierのスタイル

依存する親の名前を省略したModifierへのスタイルについては注意すべき点があります。BEMの記法と比べてみましょう。（コード4.7）

▼コード4.7: オリジナルのBEMとの比較

```
/* BEM */
.p-ad {...}
.p-ad--large {...}

/* Block、Element名の省略 */
.p-ad {...}
.p-ad.-large {...}
```

ポイントは `.-large` に直接スタイルを記述しないという点です。classセレクタを重ねることによって詳細度は高くなってしまうのですが、`.-large` というModifierは、この場合は `.p-ad` に与えたい修飾なので、理にかなったセ

*1 https://www.w3.org/TR/CSS21/syndata.html#characters

レクタです。また、`.-large`に対して直接スタイルを与えるということは、グローバルにすべての`.-large`にそのスタイルが適用されることになるので、これは大きな問題を引き起こします。（コード4.8、コード4.9）

▼コード4.8: グローバルに影響を与えるModifier:CSS

```
.p-ad {...}
.-large { /* 本来は`.p-ad`に必要なModifier */
  width: 468px;
  height: 60px ;
}
```

▼コード4.9: グローバルに影響を与えるModifier:HTML

```
<img src="ad.png" class="p-ad -large">

<!-- .p-headingにも`.-large`のスタイルが適用されてしまう -->
<h1 class="p-heading -large">Heading</h1>
```

Modifierの名前からBlock、Element名を省略した場合にはスタイルの記述方法に注意しましょう。

第5章

FLOCSSのアレンジ

FLOCSSを考案し、ひとつのREADME.mdファイルとして公開して数年経過しましたが、そこに記載されているルールなどはそれまでほとんど変わっていません。しかし、考案した私自身がFLOCSSを使うときは、基本的な原則には従いつつも、プロジェクトに応じて柔軟にアレンジをしています。ここでは他のCSS設計手法のアイデアを取り入れた例を挙げます。

5.1 EnduringなFLOCSS

ECSS（Enduring CSS）というCSS設計の方法論のことを知っているでしょうか？

ECSSはBen Frain氏によって公開されている方法論で、書籍として販売もされていますが、オンラインでも各章の内容が無償で公開されています。[*1]

端的にいえば「抽象化を避け、モジュールを徹底して疎結合とする」という方法論です。これはFLOCSSを含め、さまざまなCSS設計手法と相反するようなアイデアです。

同じ振る舞い、スタイルのコードは何度も書かずに抽象化して再利用可能にするという考え方が、多くのCSS設計手法の基礎にありますが、それにはモジュール間の依存関係を強くしてしまうというデメリットも存在しています。

過度に抽象化してしまったために、あるモジュールの修正が、意図せず違うモジュールにも変更を与えてしまうというような問題です。

***1** http://ecss.io/

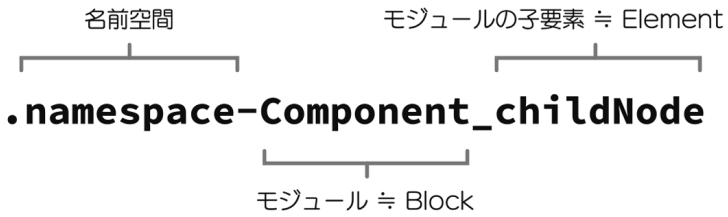
規模が大きくなるほど、抽象化したモジュールのメリットがあるはずなのですが、正しいルールで設計されていない、リファクタリングをおこなう時間を設けられないプロジェクトであれば、逆にデメリットとなってしまうことがあります。

その結果、削除すると壊れるかもしれないと思い、使われているかどうかわからないモジュールは削除されずに残り、新しいモジュールを増やしていく、というような悪夢は私も経験があります。

5.1.1 ECSSの実装例

ECSSでは、同じ見た目や振る舞いのモジュールであっても、基本的にはすべて分離して考えます。

たとえば、ヘッダーにあるログインページに遷移するボタン、登録フォームの送信ボタン、カートに入れる操作のボタンなど、FLOCSSでいえば`c-button`や`c-button.-primary`のようなものを、すべて個々のNamespaceで命名して分離します。（コード5.1）



▲ 図5.1: ECSSの命名ルール

▼コード5.1: ECSSの命名の実装例

```
 /* すべて同じ見た目のボタンスタイル */
.st-Header_login {
  display: inline-block;
  border-radius: 4px;
  padding: 8px 12px;
  background-color: green;
  box-shadow: 0 2px 4px #CCC
  text-decoration: none;
  color: white;
}

.register-Form_Submit {
  display: inline-block;
  border-radius: 4px;
  padding: 8px 12px;
  background-color: green;
  box-shadow: 0 2px 4px #CCC
  text-decoration: none;
  color: white;
}

.product-Action_AddToCart {
  display: inline-block;
  border-radius: 4px;
  padding: 8px 12px;
  background-color: green;
  box-shadow: 0 2px 4px #CCC
  text-decoration: none;
  color: white;
}
```

この一片だけをみると「何度も同じコードを書くことが煩わしい」「コード量=ファイルサイズが増えるのでは」などのネガティブな印象のほうが多くありそうです。しかし、ここまで徹底することによって、モジュールが疎結合となるので互いに影響することがなく、不要になれば容易に捨てることができます。

命名規則としてNamespaceをつけるので、探しやすさという点でも優れているといえます。同じコードを何度も書くことによるCSSのファイルサイズ増加については、gzip[*2]での配信であればさほど問題ではありません。

5.1.2 ECSSのようなFLOCSS

ここで再びFLOCSSの話に戻ります。FLOCSSの設計思想として、OOCSSのように抽象化することを基礎に置いてはいるのですが、一方でECSSのようにモジュールの依存関係を切り離したアプローチも可能です。結論からいってしまえば、Componentのレイヤーを扱わず、すべてProjectレイヤーで成立させてしまうということです。FLOCSSのProjectレイヤーでは、BEMのルールを元にしているので、ECSSでいうところのNamespaceにあたる命名は通常必要としません。

もしECSSのように徹底的に分離するならば、Block名の前にNamespaceをつけるようにすればよいでしょう。その場合、ECSSとほぼ同じ命名規則を採用し、`p-`の接頭辞を外してもよいでしょう。（コード5.2）

▼コード5.2: ECSS風のFLOCSS

```
 /* すべて同じ見た目のボタンスタイル */
.st-header__login { ... }

.register-form__submit { ... }

.product-action__addToCart { ... }
```

このアイデアをまとめると、

- モジュール設計としての主軸はECSSとする

*2 https://developers.google.com/speed/docs/insights/EnableCompression?hl=ja

- FLOCSSのFoundationレイヤーや、Webサイト共通の構造としてのLayoutレイヤーは活かす
- ObjectレイヤーはProjectのみとする

こうなると、これをFLOCSSと呼ぶべきかという意見はあるかもしれませんが、私としてはこれをFLOCSSと呼ばずとも、プロジェクトやチームにマッチした設計手法となっていればそれでよいと考えています。

おわりに

FLOCSSに関するトピックについて、考案者である私としての考え方を展開させてもらいました。FLOCSSに関するいくつかの疑問などは解消できたでしょうか？

本書を読んだとしてもわからないことや、あるいは逆に悩んでしまった人もいるかもしれません。

FLOCSSはおかげさまで多くの方に参考にしていただいていますが、それ自身が設計手法として使ってもらうことよりも、CSS設計、モジュール、コンポーネント設計のヒントやインスピレーションとなれば幸いだと考えています。

FLOCSSのように、みなさんが考えたCSS設計手法やガイドラインなどを是非公開してみてください。

柴犬でもわかるFLOCSS

2018-10-02 技術書典5版 v1.0.0
著者 谷 拓樹
デザイン hiloki
編集 hiloki
発行所 マメヒコファンクラブ

マメヒコファンクラブ 発行