"SMACSS is becoming one of the most useful contributions to front-end discussion in years"

Scalable and Modular Architecture for CSS

Français

A flexible guide to developing sites small and large.

by Jonathan Snook

Scalable and Modular Architecture for CSS

Architecture Évolutive et Modulaire des CSS

By Jonathan Snook

SMACSS: Scalable and Modular Architecture for CSS
Architecture Évolutive et Modulaire des CSS
http://smacss.com

ISBN 978-0-9856321-0-6

Snook.ca Web Development, Inc.
Ottawa, Ontario, Canada
http://snook.ca

Traduction : Estelle Bonhomme
Relecture : Jonathan Path, Laurent Sutterlity & Geoffrey Crofte

Quelques mots de l'auteur

Salut, je m'appelle Jonathan Snook. Je suis développeur et web designer. J'ai commencé à créer des sites web en tant qu'amateur en 1994, puis j'en ai fait mon métier en 1999.

J'entretiens un blog, Snook.ca, sur lequel je partage des astuces, des conseils et des liens au sujet du développement web. Je suis aussi invité régulièrement comme orateur à des conférences et des ateliers, et je suis reconnaissant de pouvoir voyager dans le monde entier pour partager mes connaissances et ma passion.

J'ai co-écrit deux livres à ce jour : The Art and Science of CSS (disponible chez Sitepoint) et Accelerated DOM Scripting (disponible chez Apress). J'ai également écrit plusieurs articles pour les magazines en ligne A List Apart, Sitepoint.com et de nombreuses autres ressources en ligne et hors-ligne.

Après avoir travaillé sur des centaines de projets web parmi lesquels la récente refonte de Yahoo! Mail, j'ai écrit ce livre pour faire part de mon expérience dans la création de sites web de toutes tailles.

J'aimerai exprimer mon immense reconnaissance envers tous ceux de la communauté. Chacun d'entre vous me permet de continuer à apprécier et à vivre pleinement ma carrière. Un grand merci particulièrement à Kitt Hodsden pour m'avoir encouragé à écrire ce livre et à le partager. Pour finir, merci à mes fils, Hayden et Lucas qui continuent à me rendre meilleur chaque jour.

INTRODUCTION

J'ai longtemps cherché à savoir combien de sites internet j'avais créés, en vain. Vous pourriez penser qu'après en avoir construit quelques centaines, j'aurai découvert LA bonne méthode. En réalité, je ne pense pas qu'il n'y ait qu'une seule bonne méthode. Par contre, j'ai découvert des techniques qui permettent de structurer et de mieux organiser ses feuilles de style CSS, ce qui permet d'avoir un code plus facile à mettre en place et à maintenir.

J'ai analysé mes procédés et ceux des autres, pour trouver comment structurer au mieux le code pour un large panel de projets. Les concepts étaient déjà plus ou moins existants pour les petits sites sur lesquels j'avais travaillé, mais sont devenus plus concrets lorsque j'ai commencé à participer à davantage de projets complexes. Les petits sites ne présentent souvent pas les mêmes problématiques que les plus gros ou que le travail en grande équipe ; les petits sites ne sont pas aussi complexes ni ne changent aussi souvent. Cependant, ce que je décris dans ces pages est une approche valable autant pour les gros sites que pour les petits.

SMACSS (prononcé « smacks ») est plutôt un guide pour la construction des feuilles de style qu'un modèle rigide. Vous n'y trouverez aucune librairie à télécharger ou à installer. SMACSS est un moyen d'analyser vos procédés en matière de design et de transformer ces structures rigides en un processus de pensée souple et adaptable. L'objectif de ce livre est de fournir une approche pertinente au développement web avec l'utilisation des CSS. Honnêtement, qui construit son site sans CSS de nos jours ? Sentez-vous libre de considérer ce livre dans son ensemble ou simplement partiellement, en retenant uniquement ce avec quoi vous serez le plus à l'aise. Ou encore, ne l'utilisez pas du tout. Je suis bien conscient

qu'il ne sera pas au goût de tout le monde. Lorsqu'il s'agit de développement web, la réponse à la plupart des questions est : « Ça dépend ».

Qu'y trouverez-vous ?

J'ai organisé mes idées autour de différents sujets concernant l'architecture CSS. Chacune d'elles est détaillée dans son propre chapitre. Lisez les chapitres les uns après les autres, ou bien dans le désordre, en choisissant ceux qui vous intéressent le plus. Ce n'est pas un livre de 1000 pages, les chapitres sont relativement courts et digestes.

Maintenant commençons !

LA CATÉGORISATION DES RÈGLES CSS

Tout projet a besoin d'une organisation. Insérer chaque nouveau style que vous créez dans un fichier unique rendra la recherche d'éléments plus difficile et compliquera la tâche des autres personnes travaillant sur le projet. Il leur sera difficile de s'y retrouver. Bien entendu, vous avez probablement déjà une organisation en place. J'espère que ce que vous lirez dans ces pages vous permettra de trouver ce qui peut fonctionner avec votre procédé actuel, et si j'ai de la chance, vous pourrez même découvrir de meilleurs moyens pour améliorer votre manière de faire.

Comment choisir entre l'utilisation des sélecteurs ID ou celle des sélecteurs de classe, ou encore, comment choisir le nombre de sélecteurs que vous avez à disposition ? De quelle manière décidez-vous à quels éléments appliquer les styles en question ? Comment faites-vous pour que l'organisation de votre site et de vos feuilles de style soit facilement compréhensible ?

Le classement par catégorie se trouve au cœur de SMACSS. En classant les règles CSS, nous à voir des modèles apparaître et pouvons ainsi définir de meilleurs procédés pour chacun de ces modèles.

Il existe cinq catégories de règles :

1. La base
2. L'agencement
3. Le module
4. L'état
5. Le thème

Nous nous retrouvons souvent à mélanger les styles de ces cinq catégories. Si nous devenons davantage conscients des éléments que nous voulons styler, nous pourrons alors éviter la complexité provoquée par l'entremêlement de ces règles.

À chaque catégorie s'appliquent certaines directives. Cette séparation succincte nous permet de nous poser les bonnes questions pendant le processus de développement. De quelle manière allons-nous coder les choses et surtout pourquoi allons-nous faire ainsi ?

L'objectif principal du classement par catégorie est de codifier des modèles – les choses qui se répètent dans votre site web. Cela permet de diminuer la quantité de code, d'avoir une maintenance plus facile et d'atteindre une plus grande uniformité dans l'expérience de l'utilisateur. Il n'y a donc que des avantages ! Quelques exceptions à la règle peuvent être intéressantes dans certains cas mais doivent toujours être justifiées.

Les règles de base sont les règles par défaut. Elles concernent presque exclusivement les sélecteurs d'éléments uniques mais peuvent également inclure des sélecteurs d'attributs, des sélecteurs de pseudo-classes, des sélecteurs d'enfant ou des sélecteurs adjacents. Un style de base dit essentiellement que, quel que soit l'endroit où se trouve l'élément sur la page, il doit porter *tel style.*

Exemples de styles de base

```
html, body, form { margin: 0; padding:0; }
input[type=text] { border: 1px solid;#999; }
a { color: #039; }
a:hover { color: #03C; }
```

Les règles d'agencement divisent la page en différentes sections. Les agencements maintiennent un ou plusieurs modules ensemble.

Les règles de modules sont réutilisables, ce sont les parties modulaires de votre design. Ce sont les zones de texte, les barres latérales, la liste de produits etc.

Les règles d'état sont un moyen de décrire à quoi doit ressembler tel module ou agencement dans un état particulier. Sera-t-il caché ou déroulé ? Actif ou inactif ? Elles décrivent également de quelle manière un module ou un agencement doit apparaître sur des écrans plus grands ou plus petits et à quoi ressemble un module dans un gabarit différent, comme la page d'accueil ou une des pages internes du site.

Pour finir, **les règles de thème** sont similaires aux règles d'état dans la mesure où elles décrivent à quoi doit ressembler un module ou un agencement. La plupart des sites n'ont pas besoin d'avoir différents thèmes mais il est bon de savoir que ces règles existent.

Nommage des règles

Lorsque l'on sépare les règles en cinq catégories, des conventions de nommages sont nécessaires pour comprendre immédiatement à quelle catégorie tel style spécifique appartient et son rôle dans l'ensemble général de la page.

J'aime utiliser un préfixe pour différencier les règles d'agencement, d'état et de module. Pour l'agencement, j'utilise l-. L'utilisation de préfixes tels que grid- permet également de clarifier la séparation entre différents styles d'agencement. Pour les règles d'état, j'aime bien utiliser is- comme dans is-hidden ou is-collapsed. Cela me permet de décrire les choses de manière très lisible.

Les modules représentent la majeure partie d'un projet. C'est la raison pour laquelle, le fait que chaque module commence par un préfixe comme `.module-` serait trop long et inutile. Les modules utilisent simplement le nom du module lui-même.

Exemples de classes

```
/* Exemple de Module */
.example { }
/* Module de texte */
.callout { }
/* Module de texte avec un état */
.callout.is-collapsed { }

/*  Module de formulaire */
.field { }

/* Agencement en ligne  */
.l-inline { }
```

Les éléments relatifs à un module ont pour préfixe le nom de la base. Sur ce site, l'exemple de code utilise `.exm` et les sous-titres s'écrivent `.exm-caption`. Il me suffit donc de voir la classe du sous-titre pour comprendre instantanément qu'il est relatif aux exemples de code et pour retrouver l'endroit où son style est décrit.

Les modules qui sont une variation d'un autre module devraient également avoir pour préfixe le nom du module initial. Le thème du sous-classement est abordé plus en détail dans le chapitre Les Règles de Module.

Cette convention de nommage sera utilisée tout au long du livre. Comme la plupart des choses que je vais partager, vous n'avez pas à vous conformer à ces conseils de manière rigide. Ayez une convention, documentez-la et tenez-vous en à ce que vous avez décidé.

LES RÈGLES DE BASE

Une règle de base s'applique à un élément qui utilise un sélecteur d'élément, un sélecteur descendant, ou un sélecteur d'enfant, ainsi que toute pseudo-classe. Elle ne comporte aucune classe ou sélecteur ID. Elle définit le style par défaut d'un élément pour toutes ses apparitions sur la page.

Exemples de styles de base

```
body, form {
   margin: 0;
  padding: 0;
}
a { color: #039;
}
a:hover {
   color: #03F;
}
```

Les styles de base définissent la taille des titres, le style par défaut des liens, le style par défaut des typographies et les fonds du site. L'utilisation de `!important` ne devrait pas être nécessaire dans un style de base.

Je recommande fortement d'attribuer un fond au corps de page. Certains utilisateurs peuvent définir leur propre fond et choisir autre chose que du blanc. Si vous travaillez en vous attendant à ce que le fond par défaut soit blanc, vous risquez de vous retrouver avec un design qui ne ressemblera pas à ce que vous aviez en tête. Et pire, votre choix de couleur pourrait rentrer en conflit avec les paramètres utilisateurs et rendre votre site inutilisable.

Réinitialisation CSS

Une réinitialisation CSS est un ensemble de styles de base prévus pour ôter – ou réinitialiser la marge externe, la marge interne et les autres propriétés par défaut. L'objectif est de bâtir un site sur des fondations solides, qui proposera un rendu harmonieux sur les différents navigateurs.

De nombreux modèles de réinitialisation sont trop radicaux et peuvent introduire plus de problèmes qu'ils n'en résolvent. Enlever la marge externe et la marge interne des éléments pour les réintroduire par la suite duplique les efforts et augmente la quantité de code nécessaire à envoyer au client.

Beaucoup de gens trouvent la réinitialisation des styles utile dans le développement web. Assurez-vous tout de même de bien comprendre les enjeux et les inconvénients du modèle que vous souhaitez utiliser, et adaptez votre programmation en fonction.

Le fait de développer votre propre ensemble de styles par défaut, que vous réutiliserez de manière régulière d'un site à un autre, peut aussi être avantageux.

LES RÈGLES D'AGENCEMENT

L'objectif du CSS, par nature, est de disposer les éléments sur une page. Cependant, il existe une différence entre les agencements qui définissent les composants principaux et ceux qui définissent les composants secondaires. Les composants mineurs – comme les zones de texte, le formulaire d'inscription, ou le menu de navigation – sont intégrés dans les composants principaux comme l'entête ou le pied de page. Je vais donc utiliser le terme « Module » pour qualifier les composants secondaires, thème que j'aborderai dans le prochain chapitre. Pour parler des composants principaux, j'emploierai l'expression « Styles d'agencement ».

Les styles d'agencement peuvent eux-aussi être divisés en styles majeurs et styles mineurs, en fonction de votre fréquence de réutilisation de ces styles. Les styles majeurs d'agencement comme l'entête et le pied de page sont en général mis en page à l'aide des sélecteurs ID, mais prenez le temps de réfléchir aux éléments communs à tous les composants de la page et n'utilisez des sélecteurs de classe que lorsque cela est nécessaire.

Définition d'un agencement

```
#header, #article, #footer {
   width: 960px;
  margin: auto;
}
#article {
   border: solid #CCC;
  border-width: 1px 0 0;
}
```

Certains sites ont besoin d'un modèle d'agencement plus général comme par exemple, le site 960.gs[1]. Ces styles d'agencement mineurs utilisent des noms de classe plutôt que des sélecteurs ID afin de pouvoir être utilisés plusieurs fois sur la page.

En général, un style d'agencement n'a qu'un sélecteur : un seul ID ou un seul nom de classe. Dans certains cas cependant, un agencement peut avoir besoin de s'adapter à différents paramètres. Par exemple, vous pouvez établir des agencements différents en fonction de la préférence de l'utilisateur. Cette préférence sera toujours définie comme un style d'agencement et sera utilisée en combinaison avec les autres styles d'agencement.

Utiliser un style d'agencement supérieur affectant les autres styles d'agencement

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

Dans cet exemple, la classe `.l-flipped` s'applique à un niveau d'élément supérieur comme l'élément body et permet d'inverser le contenu de l'article et de la barre latérale, déplaçant la barre latérale de la droite vers la gauche et vice-versa pour l'article.

1.http://960.gs/

Deux styles d'agencement pour faire passer un élément de fluide à fixe

```
#article {
   width: 80%;
  float: left;
}
#sidebar {
   width: 20%;
  float: right;
}
.l-fixed #article {
   width: 600px;
}

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

Dans ce dernier exemple, la classe `.l-fixed` modifie le design en rendant l'agencement fixe (largeur définie en pixels) alors qu'il était fluide avant (largeur définie en pourcentage).

Une autre chose à remarquer dans cet exemple est la convention de nommage que j'ai utilisée. Les déclarations qui utilisent des sélecteurs ID prennent le nom de l'élément et ne comportent pas d'espace de nommage particulier. Les sélecteurs découlant d'une classe, en revanche, comportent un préfixe `l-` . Cela permet d'identifier facilement la cible de ces styles et de les distinguer des modules ou des états. Les styles d'agencement sont la seule catégorie primaire à utiliser des sélecteurs ID. Si vous souhaitez faire un espace de nommage pour votre sélecteur ID, en utilisant un préfixe, vous pouvez, mais il n'est pas forcément nécessaire de faire ainsi.

L'utilisation des sélecteurs ID

Soyons clairs, utiliser des attributs ID dans votre code HTML peut être une bonne chose, voire dans certains cas, s'avérer absolument

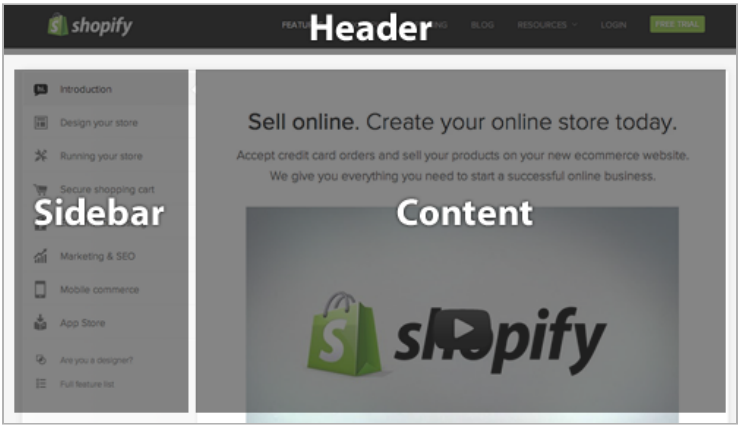
nécessaire. Par exemple, ils constituent des points d'ancrage très efficaces pour JavaScript. Pour le langage CSS, en revanche, les sélecteurs ID ne sont pas nécessaires puisque la différence de performance entre les sélecteurs ID et les sélecteurs de classe est quasi-inexistante et peut rendre la structuration du style plus complexe en raison d'une spécificité accrue.

Exemples d'agencement

Passons maintenant à la mise en pratique pour bien comprendre le concept. Regardons un site web actuel et analysons ce qui fait partie de l'agencement et ce qui relève du module.

En regardant le site Shopify, nous voyons qu'il comporte des modèles qui se retrouvent dans la grande majorité des sites. Par exem-

ple, il y a un en-tête, une zone de contenu principal, une barre latérale et un pied de page.



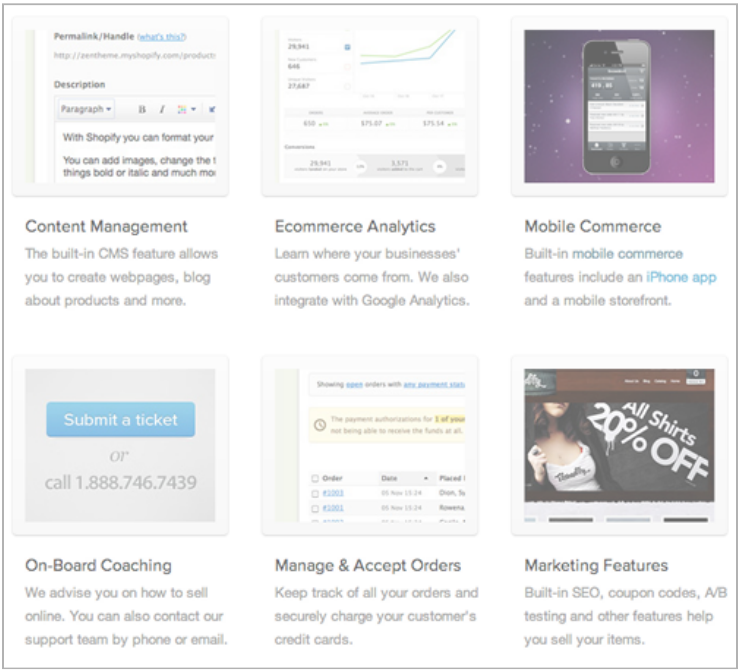
Dans votre tête, imaginez ce à quoi doit ressembler la structure HTML. Probablement un ensemble de `div`. Peut-être que vous utilisez HTML5 et que vous commencez à utiliser les éléments `header` et `footer`. Dans les deux cas, vous auriez probablement attribué un ID à chacun des récipients.

La structure CSS ressemblerait à cela :

```
#header { … }
#primarynav { … }
#maincontent { … }

<div id="header"></div>
<div id="primarynav"></div>
<div id="maincontent"></div>
```

C'est évident et je suis sûr que vous vous dites : « Sérieusement ? Vous me montrez comment faire ça ?! » Regardons une autre partie de cette page.



En observant la section Features, nous voyons une grille d'items, contenue dans un élément parent `div`, contenant lui-même une série d'enfants `div`. Utiliser une liste non-ordonnée peut aussi être un moyen pratique de marquer ces items, c'est la méthode que je vais utiliser dans cet exemple.

Exemple d'un code HTML pour l'agencement de la section « Featured »

```
<div>
<h2>Featured</h2>
<ul>
    <li><a href="…">…</a></li>
    <li><a href="…">…</a></li>
    …
</ul>
</div>
```

Si nous ne prenions pas en compte l'approche SMACSS, nous pourrions être tenté d'ajouter un ID `featured` aux divs concernées puis, de mettre en page le contenu sur cette base.

Première approche possible pour mettre en page la liste des éléments de la section « Featured »

```
div#featured ul {
   margin: 0;
  padding: 0;
   list-style-type: none;
}
div#featured li {
   float: left;
   height: 100px;
   margin-left: 10px;
}
```

Avec cette approche, nous présupposons trois choses :

1. Qu'il n'y aura jamais qu'une seule section « Featured » sur la page.
2. Que les éléments de la liste sont en `float left`.
3. Que les éléments de la liste ont une hauteur fixe de 100 pixels.

Ces suppositions sont raisonnables. Voici un premier exemple de jusqu'où un petit site peut aller avec cette structure : elle ne changera probablement pas, et ne deviendra donc probablement pas plus complexe qu'elle ne l'est déjà. *Peut-être.* En revanche, les plus gros sites, ayant un taux plus élevé de changements, présentent une probabilité accrue qu'un ou plusieurs composants doivent être redéfinis dans la page, engendrant un réajustement des styles qui leur sont attribués.

Si on retourne à notre exemple, nous pouvons donc très certainement apporter des améliorations. Le sélecteur ID ne nécessitait pas l'emploi d'un sélecteur de balise et puisque la liste est un descendant direct de `div`, le sélecteur d'enfant (>) aurait pu être utilisé.

Voyons comment ce bout de code pourrait être réajusté dans le but de nous offrir une plus grande souplesse.

Du point de vue de l'agencement, tout ce qui compte est la manière dont chaque élément est relié aux autres. Nous ne nous soucions pas nécessairement du design des modules en eux-mêmes, ni du contexte dans lequel cet agencement s'inscrit.

Grille du module qui s'applique à une OL ou une UL.

```
.l-grid {
   margin: 0;
  padding: 0;
   list-style-type: none;
}
.l-grid > li {
  display: inline-block;
  margin: 0 0 10px 10px;

/* hack IE7 pour imiter la déclaration inline-block
sur les éléments blocs */
 *display: inline;
 *zoom: 1;

}
```

Dans cette approche, quels problèmes ont été résolus et lesquels ont été introduits ? (Il est très rare qu'une solution résolve 100% des problèmes.)

1. L'agencement de la grille peut maintenant être appliqué à n'importe quel conteneur pour créer un agencement avec un style flottant.
2. Nous avons diminué la *profondeur de l'applicabilité* d'un cran (Pour en savoir plus, cf. le chapitre sur la Profondeur de l'Applicabilité).
3. Nous avons réduit la spécificité des sélecteurs.
4. La hauteur fixe a été supprimée. Chaque ligne s'ajustera automatiquement à la hauteur de l'élément le plus haut de cette ligne.

Maintenant, quels sont les désavantages de cette écriture ?

1. En utilisant un sélecteur d'enfant, nous devenons incompatibles avec IE6. (Nous pourrions contourner cela en évitant le sélecteur d'enfant.)
2. Le CSS a augmenté en taille et en complexité.

L'augmentation de la taille peut être reprochée mais elle n'est que temporaire. Maintenant que nous avons ce module réutilisable, nous pouvons l'appliquer dans l'ensemble du site sans duplication de code. Il en est de même pour la complexité accrue. Nous avons dû inclure des particularités pour les navigateurs obsolètes et intégrer des hacks qui peuvent hérisser les poils de certains. Cependant, les sélecteurs sont moins complexes, ce qui nous permet d'étendre cet agencement tout en continuant à minimiser l'impact de la spécificité.

LES RÈGLES DE MODULES

Comme je l'ai déjà mentionné brièvement dans la section précédente, un module est un composant de la page moins imposant. C'est votre barre de navigation, vos carrousels, vos dialogues, vos widgets etc. Ce sont les agréments sur votre page. Les modules se trouvent à l'intérieur des composants de l'agencement. Les modules peuvent aussi parfois être intégrés dans d'autres modules. Chaque module devrait être conçu comme un composant seul et unique. En faisant ainsi, la page sera plus souple. Si votre code est bien fait, les modules pourront facilement être déplacés dans d'autres endroits de l'agencement sans rien casser.

Lorsque vous définissez la règle d'un module, évitez l'emploi de sélecteurs ID ou de sélecteurs d'éléments, tenez-vous en aux classes. Un module contient sûrement un certain nombre d'éléments, il serait donc pratique d'utiliser des sélecteurs d'enfant ou descendants pour les cibler.

Exemple de Module

```
.module > h2 {
   padding: 5px;
}
.module span {
   padding: 5px;
}
```

Éviter les sélecteurs d'éléments

N'utilisez des sélecteurs d'enfant ou descendants avec des sélecteurs d'éléments que si les sélecteurs d'éléments sont et res-

teront prévisibles. Utiliser `.module span` est correct si vous pouvez prévoir que span sera utilisé et mis en page de la même manière à chacune de ses occurrences dans ce module.

Exemple de Module

```
<div class="fld">
    <span>Nom du dossier</span>
</div>

/* Module du dossier */
.fld > span {
   padding-left: 20px;
   background: url(icon.png);
}
```

Un problème se pose ici : plus un projet devient complexe, plus vous allez devoir étendre la fonctionnalité d'un composant et plus vous allez, par conséquent, être limité par le fait d'avoir utilisé cet élément générique dans votre règle.

Utiliser un élément générique pour la mise en page

```
<div class="fld">
    <span>Nom du dossier</span>
    <span>(32 items)</span>
</div>
```

Maintenant nous sommes dans le pétrin. Nous ne voulons pas que l'icône apparaisse sur les deux éléments du dossier de notre module. Cela me mène au point suivant :

N'incluez que des sélecteurs comportant de la sémantique. Un `span` ou un `div` n'en comportent pas. Un en-tête en a un peu. En revanche, une classe définie sur un élément en a beaucoup.

Utiliser un élément générique pour la mise en page

```
<div class="fld">
    <span class="fld-name">Nom du dossier</span>
    <span class="fld-items">(32 items)</span>
</div>
```

En ajoutant les classes aux éléments, nous augmentons leur sémantique et nous enlevons toute ambiguïté concernant le style qui leur est appliqué.

Si vous préférez utiliser un sélecteur d'élément, vous devez le placer au premier niveau de profondeur du sélecteur de classe. En d'autres termes, vous devez être dans la situation d'utiliser un sélecteur d'enfant, sur le modèle `.class element`. Autrement, vous devez être complètement sûr que l'élément en question ne pourra pas être confondu avec un autre élément.

Plus l'élément HTML est général au niveau sémantique (comme un `span` ou un `div`) plus il y a de chances que cela crée un conflit dans le déroulement du projet. Les éléments ayant une sémantique plus développée, comme les titres (h1,h2...), ont plus tendance à apparaître naturellement dans un conteneur spécifique, vous permettant alors plus facilement d'utiliser un sélecteur d'élément à bon escient.

Nouveaux contextes

Utiliser l'approche du module nous permet également de comprendre plus facilement où les changements de contexte vont se produire. Par exemple, la nécessité d'un nouveau contexte de positionnement va probablement se produire au niveau de l'agencement ou à la racine d'un module.

Sous-classement de modules

Lorsque nous avons le même module dans différentes sections, notre premier réflexe est d'utiliser l'élément parent pour designer le module différemment.

Sous-classement

```
.pod {
   width: 100%;
}
.pod input[type=text] {
  width: 50%;
}
#sidebar .pod input[type=text] {
 width: 100%;
}
```

L'ennui dans cette approche est que vous pouvez rencontrer des problèmes spécifiques qui nécessiteront encore plus de sélecteurs ou l'utilisation de `!important pour` les résoudre.

Si nous allons plus loin dans cet exemple, nous voyons que nous avons un champ de saisie avec deux largeurs différentes. Dans tout le site, l'`input` est accompagné d'un label qui permet à la largeur de ce champ d'être réduite de moitié. Par contre, dans la barre latérale, le champ serait trop petit donc nous lui redonnons une taille 100% tout en conservant le label. Tout semble bien fonctionner. Maintenant, nous devons ajouter un nouveau composant à notre page, qui a besoin du même style que le conteneur .pod. Nous réutilisons donc cette classe. Cependant, ce conteneur `.pod` est particulier et a une largeur limitée quelle que soit sa position sur le site. Il doit mesurer 180 pixels de largeur.

Combat contre la spécificité

```
.pod {
   width: 100%;
}
.pod input[type=text] {
  width: 50%;
}

#sidebar .pod input[type=text] {
  width: 100%;
}

.pod-callout {
  width: 200px;
}

#sidebar .pod-callout input[type=text],.pod-callout
input[type=text] {
  width: 180px;
}
```

Nous avons dédoublé nos sélecteurs pour pouvoir outrepasser la spécificité du `#sidebar`.

Au lieu de cela, nous devrions plutôt reconnaître que l'agencement fixe de la barre latérale est une sous-classe du conteneur `.pod` et donc, le mettre en page en fonction.

Combat contre la spécificité

```
.pod {
   width: 100%;
}
.pod input[type=text] {
  width: 50%;
}

.pod-constrained input[type=text] {
  width: 100%;
}

.pod-callout {
  width: 200px;
}

.pod-callout input[type=text] {
  width: 180px;
}
```

En sous-classant le module, les noms des classes du module de base et du sous-module sont appliqués à l'élément html.

Nom de classe d'un sous-module en HTML

```
 <div class="pod pod-constrained">...</div>
<div class="pod pod-callout">...</div>
```

Essayez d'éviter le style conditionnel basé sur la localisation. Si vous voulez changer le design d'un module pour l'utiliser autre part sur la page ou le site, sous-classez plutôt le module.

Dans ce combat contre la spécificité (et si IE6 n'est pas une préoccupation), alors vous pouvez dédoubler votre nom de classe comme dans l'exemple précédant.

Sous-classer

```
.pod.pod-callout { }

<!-- Dans le HTML -->
<div class="pod pod-callout"> ... </div>
```

Vous pouvez être concerné par cela, en fonction de l'ordre de chargement. Par exemple, sur Yahoo! Mail, le code provient de différents endroits. Nous avions notre style de base pour les boutons, puis nous avions un ensemble de boutons spécifiques pour l'écran Composer. Cependant, lorsque vous cliquiez pour ajouter un contact à votre carnet d'adresse, cela chargeait un composant d'un module différent : le carnet d'adresse. (Oui, Le carnet d'adresse est un produit différent chez Yahoo!) Le carnet d'adresse chargeait ses propres styles de boutons de base, en outrepassant les styles des boutons sous-classés que nous avions.

Si l'ordre de chargement entre en jeu dans votre projet, alors faites attention à ces problèmes spécifiques.

Des composants, ayant un agencement spécifique défini grâce aux ID, pourraient très bien être utilisés pour permettre les styles spécifiques de modules. Mais le fait de sous-classer le module lui permettra d'être plus facilement déplacé dans d'autres sections du site et vous évitera d'accroître inutilement la spécificité de votre code.

LES RÈGLES D'ÉTAT

Un état est quelque chose qui grandit et qui outrepasse les autres styles. Par exemple, une section accordéon peut être soit dans l'état replié, soit dans l'état déroulé. Un message apparaît soit dans un état de succès, soit dans un état d'erreur.

Les états s'appliquent en général au même élément qu'une règle d'agencement ou au même élément qu'une classe d'un module de base.

Un état appliqué à un élément

```
<div id="header" class="is-collapsed">
    <form>
        <div class="msg is-error">
            Il y a une erreur !
        </div>
        <label for="searchbox"
class="is-hidden">Recherche</label>
        <input type="search" id="searchbox">
    </form>
</div>
```

L'élément en-tête n'a qu'un ID. C'est la raison pour laquelle nous pouvons considérer que tout style sur cet élément, s'il y en a, servira les objectifs de l'agencement et que le `is-collapsed` représente un état replié. Nous pouvons présumer que sans cette règle d'état, l'état par défaut est l'état déroulé.

Le module `msg` est assez simple et comporte un état d'erreur. Nous pourrions imaginer qu'un état de succès pourrait être attaché au message, dans le cas contraire.

Pour finir, au label du champ est appliqué un état caché, pour qu'il ne soit pas visible à l'oeil nu mais tout de même présent pour les lecteurs d'écran. Dans ce cas, nous appliquons l'état à l'élément de base et nous n'outrepassons pas un agencement ou un module.

N'est-ce pas simplement un module ?

Il y a de nombreuses similitudes entre le style d'un sous-module et le style d'un état. Ils modifient tous deux l'apparence existante d'un élément. Cependant, ils diffèrent en deux points :

1. Les styles d'état peuvent s'appliquer à l'agencement et/ou au style du module.
2. Les styles d'état sont dépendants d'un JavaScript.

C'est ce second point qui constitue la distinction la plus importante. Les styles de sous-module s'appliquent à un élément en temps voulu puis ne sont plus jamais modifiés. Les styles d'état, eux, s'appliquent aux éléments pour indiquer un changement d'état lors de son utilisation sur la machine client.

Par exemple, cliquer sur un onglet permet d'activer cet onglet. C'est la raison pour laquelle l'utilisation d'une classe `is-active` ou `is-tab-active` est appropriée. Cliquer sur le bouton de fermeture d'un dialogue cache le dialogue. C'est pourquoi, l'utilisation d'une classe `is-hidden` est appropriée.

L'utilisation de !important

Les états devraient être appliqués de sorte à pouvoir se suffire à eux-mêmes et sont en général composés d'un sélecteur de classe unique.

Puisque qu'un état va probablement outrepasser le style d'une règle plus complexe, l'utilisation du `!important` est autorisée et, si j'ose dire, recommandée. (J'avais l'habitude de dire que `!important` n'était jamais utile mais il s'avère que dans certains systèmes

complexes, il est souvent indispensable.) En général, vous n'aurez pas deux états appliqués au même module ou deux états affectant le même type de style, les conflits de règles provenant du fait d'utiliser `!important` devraient donc être rares.

Cela dit, faites attention. Laissez le `!important` inactif tant que vous n'en avez pas réellement besoin (et vous comprendrez pourquoi dans ce prochain exemple). Souvenez-vous que l'utilisation du `!important` devrait être évitée pour tous les autres types de règles. Seuls les états peuvent en avoir.

Combinaison des règles d'états et des modules

Inévitablement, une règle d'état ne va pas pouvoir compter sur l'héritage pour appliquer son style au bon endroit. Parfois, un état est très spécifique à un module particulier ayant un style tout à fait unique.

Dans le cas où une règle d'état est créée pour un module spécifique, le nom de la classe d'état devrait comporter le nom du module. La règle d'état devrait également être placée avec les autres règles du module et non avec le reste des règles d'état générales.

Règles d'état pour le module

```
.tab {
    background-color: purple;
    color: white;
}
.is-tab-active {
    background-color: white;
    color: black;
}
```

Si vous faites un chargement en temps réel de votre CSS, les états génériques devraient être considérés comme faisant partie des styles de base généraux, et être chargés dès la première page. Les

styles d'un module particulier n'auront pas besoin d'être chargés tant que le module en question ne l'est pas.

LES RÈGLES DE THÈME

Les règles de thème ne sont pas si fréquentes dans un projet, c'est la raison pour laquelle j'ai hésité, au départ, à les considérer comme une catégorie à part entière. Cependant, certains projets doivent y faire appel comme ce fut notre cas pour le projet Yahoo! Mail.

C'est probablement évident me direz-vous, mais un thème définit les couleurs et les images qui donnent à votre application ou à votre site son aspect et son effet ! Séparer le thème en différents styles permet de redéfinir facilement ces styles pour en faire des thèmes alternatifs. La thématisation dans un projet est nécessaire lorsque vous voulez donner à l'utilisateur la possibilité de choisir entre différentes formes du site.

Par exemple, votre site peut proposer différentes couleurs pour les différentes partie du site. Ou bien vous pouvez laisser la possibilité à l'utilisateur de choisir sa couleur en fonction de ses préférences. Ou bien vous pouvez encore avoir besoin de différents thèmes s'adaptant à différents pays ou langues.

Les thèmes

Un thème peut affecter tous les types primaires. Il peut outrepasser les styles de base et les couleurs des liens par défaut. Il peut changer les éléments des modules comme les couleurs et les bordures. Il peut affecter les agencements avec différentes variantes ainsi qu'altérer l'apparence des états.

Admettons que vous ayez un module de dialogue qui doit avoir une bordure bleue. La bordure en elle-même sera initialement définie dans le module, et le thème définira ensuite la couleur :

Intégration d'un thème dans un module

```
/* dans nom-du-module.css */
.mod {
  border: 1px solid;
}
/* dans theme.css */
.mod {
  border-color: blue;
}
```

En fonction de l'ampleur de la thématisation, il peut être plus facile de définir des classes spécifiques au thème. Dans le cas de Yahoo! Mail nous avons gardé la thématisation des régions spécifiques de la page. Cela a rendu la conception de nouveaux thèmes plus simple, sans pour autant toucher la conception globale, et tout en donnant à l'utilisateur une certaine personnalisation possible.

Pour une thématisation plus vaste, il faut utiliser un préfixe `theme-` sur les éléments spécifiques du thème. Il sera alors plus facile de les appliquer à d'autres éléments de la page.

Classes de thèmes

```
/* dans le fichier theme.css */
.theme-border {
  border-color: purple;
}
.theme-background {
  background: linear-gradient( ... );
}
```

Chez Yahoo! Mail, nous utilisons un Template Mustache pour nos CSS, qui permet de spécifier plusieurs valeurs de couleur, une image de fond et de créer un fichier CSS final pour la production, cela en vue de maintenir une homogénéité entre tous nos fichiers de thème – il y en a plus d'une cinquantaine.

La typographie

Dans la thématisation, il y a des moments où vous devez redéfinir les polices utilisées de manière générale sur le site, comme c'est le cas lors d'une localisation de site. Certains pays, comme la chine ou la Corée, ont des idéogrammes difficiles à lire et des tailles de police plus petites. En conséquence, nous créons des fichiers de police séparés pour chaque localisation, qui redéfinissent la taille de la police des composants.

Les règles de police affectent normalement les styles de base, des modules et de l'état. Les styles de police ne sont en général pas spécifiés au niveau de l'agencement car ceux-ci sont utilisés pour le positionnement et le placement des éléments, non pour les changements stylistiques comme la police et les couleurs.

Tout comme pour les fichiers de thème, la définition de classes pour la police, comme par exemple, `font-large`, ne devrait pas être nécessaire. Votre site ne devrait avoir que 3 à 6 tailles de police différentes. Si vous avez plus de 6 tailles de police dans votre projet, vos utilisateurs risquent de ne même pas le remarquer et vous rendez votre site plus complexe à maintenir.

Ce que renferme un nom

Nommer les classes de thème et de typographie est en général complexe car nous sommes dans une industrie qui les considère comme étant non-sémantiques. Les composants d'un thème, quant à eux, sont visuels et non-sémantiques de manière inhérente. Mais pour la typographie, ce n'est pas vraiment le cas. Après tout, c'est de hiérarchie visuelle dont il s'agit dans le design, votre typographie devrait donc refléter cela. C'est la raison pour laquelle la convention de nommage que vous choisissez au final devrait indiquer les différents niveaux d'importance, tout comme vous le feriez avec les niveaux de titres dans HTML.

LES CHANGEMENTS D'ÉTAT

Vous avez un fichier Photoshop devant vous et vous avez la mission de l'intégrer par la magie du html et du css (avec peut-être, et dans une bonne mesure, un peu de JavaScript).

Il semblerait évident de commencer directement la conversion du document en code. Cependant, il y a de fortes chances pour que plusieurs des éléments de votre page aient besoin d'être représentés dans différents états. Il y a l'état par défaut, décrivant l'apparence par défaut d'un composant, puis son apparence lors du changement d'état.

Qu'est-ce qu'un changement d'état ?

Les changements d'état se font par l'un des moyens ci-dessous :

1. Un nom de classe
2. Une pseudo-classe
3. Une Media Query

Un changement par **nom de classe** se fait grâce à JavaScript, par l'intermédiaire d'une interaction, comme le mouvement de la souris, une commande du clavier, ou un autre événement. L'élément en question se voit alors appliquer une nouvelle classe entraînant une modification de son apparence.

Un changement par **pseudo-classe** se fait par l'intermédiaire d'un certain nombre de pseudo-classes, et il en existe beaucoup. Dans ce cas, nous n'avons plus besoin de JavaScript pour décrire le changement d'état. Mais les pseudo-classes sont tout de même limitées, dans le sens où nous ne pouvons appliquer des changements de

style qu'aux éléments descendants ou frères de l'élément auquel la pseudo-classe s'applique. Sinon, nous sommes obligés de revenir à JavaScript.

Pour finir, les **Media Queries** décrivent la manière dont les choses devraient changer de style en fonction de certains critères, comme par exemple, la variation de taille des viewports.

Dans un système basé sur les modules, il est important de considérer un design fondé sur l'état, à appliquer à chaque module. Lorsque vous vous demandez quel est l'état par défaut, alors vous êtes en train de penser de manière proactive à l'amélioration progressive de votre projet. C'est également une manière légèrement différente d'aborder les problèmes.

Changement par l'intermédiaire d'un nom de classe

La plupart des changements par nom de classe sont directs. Ils s'appliquent à des éléments qui changent d'état. Par exemple, un utilisateur clique sur une icône de fermeture pour faire apparaître ou cacher un élément sur la page.

Changement d'état avec JavaScript par l'intermédiaire d'un nom de classe

```
// avec jQuery
$('.btn-close').click(function(){
  $(this).parents('.dialog').addClass('is-hidden');
})
```

Cet exemple jQuery ajoute un gestionnaire d'événement « click » à chaque élément comportant le nom de classe `btn-close`. Lorsque l'utilisateur clique sur le bouton, la source de l'évènement est prise en compte et la structure du DOM est remontée jusqu'à ce que l'élément portant la classe `dialog` soit retrouvé. Ensuite, il lui est

appliqué l'état de classe `is-hidden`. Dans d'autres cas, un changement d'état peut avoir un plus grand impact.

Un modèle de design d'interface fréquent est celui d'un bouton donnant accès à un menu. Dans ce cas, le menu évolue d'un état enroulé à un état déroulé. Quelles options s'offrent à nous pour provoquer ce changement ? Cela dépend fortement de votre structure HTML. Par exemple, sur le site Yahoo!, les menus sont chargés au moment de la requête et sont donc insérés tout en haut du DOM. Nous avons utilisé une convention de nommage pour relier les deux ensemble.

Le bouton et le menu séparés dans le même document

```
<div id="content">
   <div class="toolbar">
      <button id="btn-new" class="btn"
data-action="menu">New</button>
   </div>
</div>
<div id="menu-new" class="menu">
   <ul> ... </ul>
</div>
```

L'action liée à un clic JavaScript transmet l'info « Hey, tu dois charger le menu ». Le bouton ID est nécessaire pour trouver le menu correspondant. Voici comment cela fonctionnerait avec jQuery :

Charger le menu avec jQuery

```
// Ajoute un gestionnaire de clic au bouton
$('#btn-new').click(function(){
    // Insert jQuery au bouton cliqué
    var el = $(this);
    // Change l'état du bouton
    el.addClass('is-pressed');
    // Trouve le menu en déroulant btn- et
    // l'ajoute au sélecteur de menu
    $('#menu-'
+el.id.substr(4)).removeClass('is-hidden');
});
```

Comme illustré dans l'exemple ci-dessus, l'état change grâce à un seul élément qui est modifié sur deux éléments différents, à deux endroits différents, à l'aide de JavaScript. Mais quand est-il si le menu est disposé juste à côté du bouton ?

Bouton et menu dans la même partie du document.

```
<div id="content">
   <div class="toolbar">
      <button id="btn-new" class="btn"
data-action="menu">New</button>
      <div id="menu-new" class="menu">
          <ul> ... </ul>
      </div>
   </div>
</div>
```

Le code précédant fonctionnerait exactement de la même manière et n'aurait absolument pas besoin d'être modifié. Cependant, il existe des alternatives. Votre premier réflexe pourrait être d'ajouter une classe à un élément parent et de définir le style du bouton et du menu à ce niveau là.

Ajout d'une classe à un élément parent pour appliquer le style les éléments enfants

```
<div id="content">
   <div class="toolbar is-active">
      <button id="btn-new" class="btn"
data-action="menu">Nouveau</button>
      <div id="menu-new" class="menu">
          <ul> ... </ul>
      </div>
   </div>
</div>

/* Le CSS pour appliquer le style */

.is-active .btn { color: #000; }
.is-active .menu { display: block; }
```

Le problème de cette approche est que la structure HTML ne peut plus être modifiée. Il doit y avoir un élément contenant. Le menu et le bouton doivent exister dans cet élément contenant. Espérons qu'il n'y ait aucun bouton supplémentaire à rajouter dans cette barre d'outils !

Une autre approche consiste à appliquer la classe `active` au bouton, comme nous l'avons fait précédemment, et à utiliser un sélecteur adjacent pour activer le menu.

Activer le menu avec un sélecteur adjacent

```
<div id="content">
   <div class="toolbar">
      <button id="btn-new" class="btn is-active"
data-action="menu">Nouveau</button>
      <div id="menu-new" class="menu">
          <ul> ... </ul>
      </div>
   </div>
</div>

/* Le CSS pour pour appliquer le style */

.btn.is-active { color: #000; }
.btn.is-active + .menu { display: block; }
```

Je préfère cette approche à celle appliquant une classe d'état à un élément parent, car l'état est mieux combiné au module auquel il s'applique. Cependant, le menu HTML doit toujours être lié au bouton HTML : l'un doit venir immédiatement après l'autre. Si vous êtes en mesure de pouvoir établir cette homogénéité et cette cohérence dans votre projet, alors c'est une approche qui peut vous correspondre.

Pourquoi les états parents et adjacents posent-ils problème ?

La raison pour laquelle cette approche peut être plus problématique que celle consistant simplement à appliquer un état à chaque module, est que l'endroit où cette règle s'applique n'est plus vraiment clair. Le menu n'est plus un simple menu. C'est un menu de boutons. Si vous aviez besoin de modifier l'état actif de ce module, devriez-vous modifier le CSS du bouton ou celui du menu ?

Tout cela pour dire qu'appliquer un état à chaque bouton est l'approche la plus recommandée. Vous créez une meilleure sépara-

tion entre les modules, rendant ainsi votre site plus facile à tester, à développer et à faire évoluer.

Gérer le changement d'état avec les sélecteurs d'attribut

En fonction de votre support de navigateur, vous pouvez également tirer parti des sélecteurs d'attribut pour gérer les changements d'état. Cela peut être utile pour :

- Isoler les états des classes de l'agencement et du module
- Permettre des transitions plus faciles entre les états multiples

Regardons l'exemple d'un bouton pouvant avoir différents états : par défaut, pressé ou désactivé.

Vous pouvez utiliser une convention de nommage de sous-module.

Convention de nommage de sous-module

```
.btn { color: #333; }
.btn-pressed { color: #000; }
.btn-disabled { opacity: .5; pointer-events: none; }
```

Si un bouton doit pouvoir alterner entre différents états, alors il semblerait plus logique d'utiliser une convention de nommage d'état.

Convention de nommage d'état

```
.btn { color: #333; }
.is-pressed { color: #000; }
.is-disabled { opacity: .5; pointer-events: none; }
```

J'aime la comparaison entre ces deux exemples parce qu'elle montre que dans ce livre, il s'agit beaucoup de clarification et de conventions de nommage. Je serais ravi de voir l'un ou l'autre de ces exem-

ples dans un projet. Maintenant, abordons une autre approche : *les sélecteurs d'attribut*.

Convention de sélecteur d'attribut

```
.btn[data-state=default] { color: #333;}
.btn[data-state=pressed] { color:
#000;}.btn[data-state=disabled] { opacity:.5;
pointer-events: none; }

<!-- HTML -->
<button class="btn"
data-state="disabled">Désactivé</button>
```

Le préfixe `data-` sur l'attribut fait partie des spécifications du HTML5 qui vous permettent de créer vos propres noms d'attribut et de les placer dans l'espace de nommage des données afin qu'il n'y ait aucun conflit avec les futures spécifications des attributs HTML. Changer l'état d'un bouton ne nécessite pas d'enlever ou d'ajouter des classes mais simplement de changer la valeur d'un seul attribut.

Changement d'état avec jQuery

```
// Ajoute un gestionnaire de clic à chaque bouton
$(".btn").bind("click", function(){
    // change l'état à celui de pressé
    $(this).attr('data-state', 'pressed');
});
```

Il faut reconnaître qu'avec les bibliothèques JavaScript comme jQuery, il n'est pas compliqué de manipuler les classes pour gérer les états. jQuery a des méthodes comme `hasClass`, `addClass`, et `toggleClass` qui rendent le travail avec les noms de classes très facile.

Cela est suffisant pour dire que vous avez différentes possibilités pour représenter un état.

Le changement d'état basé sur les classes avec les animations CSS

Les animations sont un procédé intéressant dont certains disent qu'il définit le comportement à un niveau où il ne le devrait pas. Les CSS sont pour le style après tout. Et JavaScript est pour le comportement.

La distinction à faire ici est de comprendre que les CSS définissent un état visuel. Nous pouvons utiliser JavaScript pour changer l'état d'un élément sur notre page. JavaScript ne devrait cependant pas être utilisé pour décrire un état. C'est-à-dire qu'il ne devrait pas être utilisé pour ajouter des styles `inline`.

Historiquement, nous avons utilisé JavaScript pour créer des animations parce que c'était le seul moyen que nous avions à disposition (malgré HTML+TIME2[2]).

Réfléchir en ces termes peut nous aider à formuler notre approche dans différentes situations. Par exemple, il n'est pas rare qu'un message apparaisse sur une page pendant quelques instants puis disparait ensuite.

Changement d'état avec JavaScript

```
function showMessage (s) {
    var el = document.getElementById('message');
    el.innerHTML = s;
    /* établit l'état */
    el.className = 'is-visible';
    setTimeout(function(){
        /* établit l'état d'origine */
        el.className = 'is-hidden';
    }, 3000);
}
```

L'état du message passe de caché à visible pour revenir à l'état caché. Le code JavaScript établit les changements entre ces états et

2.http://www.w3.org/TR/NOTE-HTMLplusTIME

c'est ensuite le CSS qui peut être utilisé pour animer ces états – en utilisant soit des transitions soit des animations.

Le CSS gérant les transitions

```
@keyframes fade {
    0% { opacity:0; display:block; }  100% {
opacity:1; display:block; }
}

.is-visible {
    display: block;
    animation: fade 2s;
}

.is-hidden {
    display: none;
    animation: fade 2s reverse;
}
```

Je reconnais que ce dernier exemple ne pourrait pas réellement fonctionner. Malheureusement, cette spécification d'animation et les implémentations du navigateur ne nous permettent pas de spécifier, dans notre animation, des propriétés qui ne concernent pas l'animation. Heureusement, la spécification évolue et pourrait autoriser cela au final – si les implémentations des navigateurs suivent l'évolution. En même temps, nous devons faire quelque chose comme ce qui suit.

Les animations dans les navigateurs actuels

```
@-webkit-keyframes fade {
    0% { opacity:0;  }
    100% { opacity:1; display:block; }
}
.is-visible {
    opacity: 1;
    animation: fade 2s;
}

.is-hidden {
    opacity: 0;
    animation: fade 2s reverse;
}

.is-removed {
    display: none;
}

/* puis notre javascript */
function showMessage (s) {
    var el = document.getElementById('message');
    el.innerHTML = s;
    /* définit l'état */
    el.className = 'is-visible';
    setTimeout(function(){
        /* définit le retour à l'état */
        el.className = 'is-hidden';
        setTimeout(function(){
            el.className = 'is-removed';
        }, 2000);
    }, 3000);
}
```

Dans ce cas, j'ai modifié le code pour qu'il fasse l'animation, mais cette-fois-ci, en utilisant JavaScript pour enlever l'élément du flux une fois que l'animation est finie. De cette manière, nous maintenons la séparation entre le style (ou l'état) et le comportement.

Changement par l'intermédiaire de pseudo-Classes

Comme nous venons de le voir, nous pouvons utiliser des classes et des attributs pour définir les changements d'état sur un module. Cependant, les CSS nous proposent de nombreuses pseudo-classes pouvant également nous aider à gérer les états et les changements d'état. Dans CSS2.1, les trois pseudo-classes les plus utiles sont les « dynamiques » qui réagissent à l'interaction de l'utilisateur : `:hover`, `:focus`, et `:active`. CSS3 ajoute plusieurs nouvelles pseudo-classes dont la plupart sont basées sur la structure HTML (comme `:nth-child` ou `:last-child`). Il y a plusieurs pseudo-classe UI (interface homme-machine) en CSS3 qui peuvent répondre à des interactions de forme, ce qui peut également être bien utile. L'état par défaut d'un module est normalement défini sans pseudo-classe. Celle-ci sert plutôt à définir l'état secondaire d'un module.

Définition d'états par les pseudo-classes

```
.btn {
    background-color: #333; /* gris */
}

.btn:hover {
   background-color: #336; /* bleu-vert */
}
.btn:focus {
    /* Ombre portée bleue */
    box-shadow: 0 0 3px rgba(48,48,96,.3);
}
```

Comme les modules sont sous-classés, vous pouvez être amené à définir les états de pseudo-classes pour les sous-modules également, ce qui peut potentiellement devenir compliqué.

Définition d'états des sous-modules à l'aide de pseudo-classes

```
.btn {
    background-color: #333; /* gris */
}
.btn:hover {
    background-color: #336; /* bleu-vert */
}

.btn:focus {
    /* Ombre portée bleue */
    box-shadow: 0 0 3px rgba(48,48,96,.3);
}

/* l'état par défaut d'un bouton est le choix
 * par défaut d'une sélection de boutons
 */

.btn-default {
 background-color: #DEDB12; /* jaunâtre */
}

.btn-default:hover {
    background-color: #B8B50B; /*jaune plus foncé */
}

/* pas besoin de définir un état focus par défaut */
```

Dans ce dernier exemple, nous avons essentiellement créé 5 variations pour un seul module : un module primaire, un sous-module, puis les états de pseudo-classes possibles. Cela peut devenir encore plus compliqué lorsque nous introduisons des états basés également sur la classe de chacune de ces variations.

Modules, sous-modules, états de classe et états de pseudo-classes Oh mince…

```
.btn { ... }
.btn:hover { ... }
.btn:focus { ... }
.btn-default { ... }
.btn-default:hover { ... }
.btn.is-pressed { ... }
.btn.is-pressed:hover { ... }
.btn-default.is-pressed { ... }
.btn-default.is-pressed:hover { ... }
```

Heureusement, seuls peu de modules dans votre interface vont nécessiter ce type de structure. Cependant, il est clair qu'une organisation appropriée de vos styles assurera une maintenance plus facile de votre projet.

Changement d'état par l'intermédiaire des Media Queries

Alors que les changements d'état par l'intermédiaire des classes et des pseudo-classes sont assez communs, les Media Queries sont en train de devenir une nouvelle approche pour gérer le changement d'état – une approche qui, traditionnellement, n'était possible qu'avec JavaScript. L'*adaptative* et le *responsive design*[3] (conception *adaptative* d'un site web) utilisent les Media Queries pour réagir à différents critères. Les CSS pour l'impression (Print Style Sheets) étaient l'une des premières Media Queries nous permettant de définir à quoi les éléments devaient ressembler une fois imprimés sur une feuille.

Une Media Query peut être définie comme une feuille de style séparée qui utilise l'attribut media sur le lien de l'élément ou peut être définie à l'intérieur d'un bloc `@media` intégré à une feuille de style spécifique.

3.http://www.alistapart.com/articles/responsive-web-design/

Liens entre feuilles de style

```
<link href="main.css" rel="stylesheet">
<link href="print.css" rel="stylesheet"
media="print">

/* dans le fichier main.css */
@media screen and (max-width: 400px) {
    #content { float: none; }
}
```

La plupart des exemples de Media Queries définissent un point de rupture et y renvoient tous les styles qui s'y rapportent.

Dans ce livre, nous essayons de garder les styles attachés à un module particulier avec le reste du module. Cela signifie qu'au lieu d'avoir un seul point de rupture, soit dans une feuille CSS principale, soit dans une feuille de style Media Query séparée, nous plaçons la Media Query proche des états du module.

Media Queries modulaires

```
/* default state for nav items */
.nav > li {
   float: left;
}

/* Etat alterné pour les éléments nav sur petits
écrans */
@media screen and (max-width: 400px) {
    .nav > li { float: none; }
}

... autre part pour l'agencement ...

/* agencement par défaut*/
.content {
   float: left;
   width: 75%;
}
.sidebar {
   float: right;
   width: 25%;
}

/* Etat alterné pour les agencements sur petits
écrans */
@media screen and (max-width: 400px) {
    .content, .sidebar {
        float: none;
        width: auto;
  }
}
```

Oui, cela signifie que la Media Query sera probablement déclarée plusieurs fois, mais cela permet à toute l'information concernant un module d'être regroupée au même endroit. Souvenez-vous que rassembler toute l'information d'un module au même endroit (particulièrement dans son propre fichier CSS) permet de tester le module de manière isolée et, en fonction de la manière dont vous construisez votre application, permet aux gabarits modularisés et aux CSS d'être chargés après le chargement de la page initiale.

C'est de l'état dont il s'agit

Ce chapitre a passé en revue les trois types de changement d'état : classe, pseudo-classe et les Media Queries. Le fait de penser à votre interface non seulement de manière modulaire, mais également comme une représentation de ces modules dans leurs différents états, simplifiera la séparation appropriée des styles et permettra la construction de sites plus simples à maintenir.

LA PROFONDEUR DE L'APPLICABILITÉ

Lorsque nous apprenons à travailler avec les CSS, nous utilisons des sélecteurs pour sélectionner les éléments HTML de la page auxquels nous voulons appliquer un style. Le code CSS s'est développé au fil des années pour nous donner encore plus de possibilités, grâce à un nombre croissant de sélecteurs. Néanmoins, chaque ensemble de règles que nous ajoutons à notre feuille de style accroît le nombre de connexions entre la CSS et le code HTML.

Analysons un bloc typique de code CSS que vous pourriez trouver sur un site internet.

La manière de relier le CSS à l'HTML

```
#sidebar div {
    border: 1px solid #333;
}
#sidebar div h3 {
    margin-top: 5px;
}
#sidebar div ul {
    margin-bottom: 5px;
}
```

En regardant cela, vous pouvez voir à quoi doit ressembler la structure HTML. Il y a probablement une ou plusieurs sections dans une barre latérale, qui ont chacune un titre suivi d'une liste non-ordonnée. Si le site n'évolue pas très souvent, ce type de CSS est adapté. Je n'ai pas changé le design de mon blog en deux ans. Le besoin d'adaptation rapide n'est donc pas présent. Mais si j'essaie d'utiliser cette approche sur un site plus grand, qui peut évoluer

plus fréquemment et avoir une plus grande variété de conditions au niveau du code, je vais rencontrer des problèmes. Je vais avoir besoin de rajouter davantage de règles comportants des sélecteurs plus complexes. Je peux me retrouver dans un cauchemar au niveau la maintenance.

Où est-ce que je me suis trompé ? Il y a deux soucis précis dans l'exemple que j'ai donné :

1. Je dépends fortement d'une structure HTML définie.
2. Le degré de profondeur du code HTML auquel s'appliquent les sélecteurs est trop élevé.

Diminuer la profondeur

Le HTML est comme la structure d'un arbre généalogique. La profondeur d'applicabilité est le nombre de générations affectées par une règle donnée. Par exemple, `body.article > #main > #content > #intro > p > b` a une profondeur d'applicabilité de 6 générations. Si ce sélecteur était écrit de la manière suivante : `.article #intro b`, la profondeur serait la même, 6 générations.

Le problème d'une telle profondeur est qu'elle engendre une plus grande dépendance à une structure HTML précise. Les composants de la page ne pourront pas être déplacés facilement. Si nous retournons à notre exemple avec la barre latérale, de quelle manière pouvons-nous recréer ce module dans une autre partie de la page comme le pied de page ? Nous devons dupliquer les règles.

Duplication des règles

```
#sidebar div, #footer div {
    border: 1px solid #333;
}
#sidebar div h3, #footer div h3 {
    margin-top: 5px;
}
#sidebar div ul, #footer div ul {
    margin-bottom: 5px;
}
```

L'intersection d'origine se trouve au niveau du `div` et c'est à partir de là que nous devrions créer nos styles.

Simplification des règles

```
.pod {
    border: 1px solid #333;
}

.pod > h3 {
   margin-top: 5px;
}

.pod > ul {
    margin-bottom: 5px;
}
```

Le conteneur `.pod` dépend encore d'une structure HTML précise mais qui est d'une profondeur beaucoup plus faible que ce que nous avions auparavant. En contre-partie, nous devons répéter la classe pod sur un certain nombre d'éléments sur la page, alors qu'avant, nous n'avions que deux éléments comportant des ID. Bien entendu, nous voulons éviter de retourner à l'aire où nous faisions des choses idiotes comme ajouter des noms de classe à chaque paragraphe.

L'avantage de cette approche qui diminue la profondeur d'applicabilité, est la possibilité de convertir plus facilement ces

modules en gabarits pour avoir du contenu dynamique. Sur Yahoo!, par exemple, nous nous appuyons sur une structure Mustache pour la plupart des besoins de nos gabarits. Voici comment nous organiserions notre gabarit pour ces conteneurs `.pod` :

Un exemple de Template Mustache

```
<div class="pod">
    <h3>{{heading}}</h3>
    <ul>
        {{#items}}
        <li>{{item}}</li>
        {{/items}}
    </ul>
</div>
```

Nous essayons de trouver un équilibre entre la maintenance, la performance et la lisibilité. Aller trop en profondeur signifierait diminuer le nombre de classes dans votre HTML, ce qui accroîtrait le travail de maintenance et de gestion. De même, vous ne voulez pas (ou n'avez pas besoin) de mettre des classes sur tout. Rajouter des classes sur le `h3` ou le `ul` dans cet exemple aurait été un peu inutile à moins que nous ayons eu besoin d'avoir un système encore plus souple.

Ce modèle de design va nous permettre d'aller encore plus loin dans cet exemple. C'est un contenant avec un en-tête et un corps principal. (Parfois, vous aurez également un pied de page.) Nous avons un ul dans cet exemple, mais dans les autres exemple nous auront soit un `ol`, soit un `div` à la place.

Encore une fois, nous pouvons dupliquer nos règles pour chaque variation.

Duplication de règle

```
.pod > ul, .pod > ol, .pod > div {
    margin-bottom: 5px;
}
```

L'alternative est de classifier l'élément `body` du conteneur `.pod`.

Simplification avec une classe

```
.pod-body {
    margin-bottom: 5px;
}
```

Avec l'approche de la règle du module, il n'est même pas nécessaire de spécifier la classe `.pod`. Nous pouvons voir visuellement que `.pod-body` est associé au module `pod` et, du point de vue du code, cela fonctionnera bien.

Un exemple de Template Mustache

```
<div class="pod">
    <h3>{{heading}}</h3>
    <ul class="pod-body">
        {{#items}}
        <li>{{item}}</li>
        {{/items}}
    </ul>
</div>
```

Grâce à ce petit changement, nous avons pu réduire la profondeur de l'applicabilité à la plus petite valeur possible. Le sélecteur simple signifie également que nous évitons ainsi des problèmes de spécificité potentiels. Nous sommes donc gagnants au final.

LA PERFORMANCE DES SÉLECTEURS

Dans mon travail, j'ai du faire pas mal d'analyses sur la performance. Nous utilisons plusieurs outils sur une application ou sur un site pour détecter les endroits où il y a des nœuds. L'un de ces outils est Google Page Speed[4] qui fournit plusieurs recommandations quant au moyen d'améliorer notre code JavaScript et de rendre le site en question plus performant. Avant de détailler ces recommandations, nous devons comprendre un peu plus la manière dont les navigateurs évaluent les CSS.

La manière dont les CSS sont évaluées

Le style d'un élément est évalué à la création de l'élément

Nous voyons souvent nos pages comme des documents remplis et complets, pleins d'éléments et de contenu. Cependant, les navigateurs sont conçus pour gérer les documents en flux continu. Ils commencent à recevoir un document du serveur et peuvent donner son rendu avant même qu'il ne soit complètement chargé. Chaque nœud est évalué et rendu au viewport dès qu'il est réceptionné.

4.http://code.google.com/speed/page-speed/

Un exemple de document HTML

```
<body>
   <div id="content">
      <div class="module intro">
         <p>Lorem Ipsum</p>
      </div>
      <div class="module">
         <p>Lorem Ipsum</p>
         <p>Lorem Ipsum</p>
         <p>Lorem Ipsum <span>Test</span></p>
      </div>
   </div>
</body>
```

Le navigateur commence sa lecture en haut et voit un élément `body`. À ce stade, il pense que le `body` est vide. Il n'a rien évalué d'autre. Puis le navigateur va déterminer les styles et les appliquer à l'élément. Quelles sont la police, la couleur, la taille de caractère ? Après avoir analysé ces informations, elles apparaissent à l'écran.

Ensuite, il voit un élément `div` avec un ID pour le contenu. Encore une fois, à ce point, il pense qu'il est vide. Il n'a rien analysé d'autre. Le navigateur doit analyser le style et c'est alors qu'il peut apparaître à l'écran. Le navigateur va alors déterminer s'il doit redéfinir le `body` – est-ce que l'élément est devenu plus large, plus haut ? (J'imagine qu'il y a d'autres considérations mais les variations de largeur et de hauteur sont les effets des éléments enfants sur leur parents les plus fréquents.)

Ce processus continue jusqu'à ce que le navigateur ait atteint la fin du document.

Pour visualiser ce processus chez Firefox, visitez le lien suivant : http://youtu.be/ZTnIxIA5KGw[5]

5.http://http://youtu.be/ZTnIxIA5KGw

Les CSS sont évaluées de droite à gauche.

Pour déterminer si une règle CSS s'applique à un élément particulier, le navigateur la lit de droite à gauche.

Si vous avez une règle comme `body div#content p { color: #003366; }` alors pour chaque élément, le navigateur va d'abord chercher à savoir si c'est un paragraphe – au moment de son rendu sur la page. Si oui, alors il va remonter dans la structure du DOM pour demander et vérifier si un des parents porte un attribut ID équivalent à `content` puis vérifier si c'est bien un élément `div` qui porte l'attribut. S'il trouve ce qu'il cherche, il va continuer à remonter le DOM jusqu'à vérifier la présence du parent `body`.

En travaillant ainsi, de la droite vers la gauche, le navigateur peut déterminer si une règle s'applique à cet élément particulier qu'il veut faire apparaître sur le viewport le plus rapidement possible. Pour déterminer la performance d'une règle, vous devez analyser combien de nœuds doivent être évalués avant que le navigateur puisse déterminer si tel style s'applique à tel élément.

Quelles règles dominent ?

Alors que chaque élément est généré sur la page, le navigateur doit trouver quels styles s'appliquent à quel élément. Maintenant, regardez les recommandations[6] de Google Speed Page. Il y a quatre règles principales étant considérées comme inefficaces :

6.http://code.google.com/speed/page-speed/docs/rendering.html#UseEfficientCSSSelectors

- Les règles comportant des sélecteurs descendants : Ex : `#content h3`
- Les règles comportant des sélecteurs d'enfant ou adjacents. Ex : `#content > h3`
- Les règles comportant des sélecteurs trop qualifiés. Ex : `div#content > h3`
- Les règles appliquant :hover à des éléments qui ne sont pas des liens. Ex : `div#content:hover`

Au regard de ces recommandations, il est important de noter que l'évaluation d'un seul élément pour déterminer le style est insuffisante. Sinon, cela signifierait que vous ne pourriez utiliser qu'un seul sélecteur dans votre règle : un sélecteur de classe, un sélecteur ID, un sélecteur d'élément ou un sélecteur d'attribut. Si nous prenions ces recommandations au pied de la lettre, c'est comme si on nous conseillaient de revenir au temps des `<p class="bodytext">`. (Mais si vous regardez le CSS généré sur des produits comme Search et Google Mail, vous verrez qu'ils suivent effectivement ces recommandations.)

Faites attention mais ne perdez pas trop de temps

Pour tous les autres, je crois que nous pouvons être un peu plus pratiques et trouver un équilibre entre un extrême (ajouter des classes et des identifiants partout) et l'autre (utiliser des règles de sélecteurs très profondes créant une dépendance forte entre le HTML et le CSS).

J'ai l'habitude de suivre trois directives simples pour limiter le nombre d'éléments ayant besoin d'être évalués :

1. J'utilise des sélecteurs d'enfant.
2. J'évite les sélecteurs de balise pour les éléments fréquents.
3. Je fais en sorte que les noms de classe soient les sélecteurs les plus à droite.

Par exemple, `.module h3` pourrait être acceptable si je peux savoir à l'avance que je ne dépasserai pas la douzaine de `h3` sur ma page. À quelle profondeur sont placés mes titres `h3` dans le DOM ? Sont-ils au niveau 4 (ex : `html > body > #content > h3`) ou bien sont-ils au niveau 10 (ex : `html > body > #content > div > div > ...> h3`) ? Puis-je limiter la complexité du labyrinthe du DOM en utilisant uniquement des sélecteurs d'enfant ? Si je peux utiliser `.module > h3` (désolé IE6), alors je sais que pour les 12 titres `h3` que j'ai sur ma page, seuls 24 éléments devront être évalués. En revanche, si je fais .module div , et que j'ai 900 div sur ma page (je viens de regarder le code de ma boîte de réception Yahoo! Mail et il y en a 903), alors le nombre d'allers-retours sera beaucoup plus élevé. Un simple `<div><div><div></div></div></div>` (3 niveaux de profondeurs) engendre 6 évaluations. C'est factoriel. 4 niveaux de profondeurs engendrent 24 évaluations et 5 en engendrent 120.

Cela dit, même ces optimisations simples peuvent être excessives. Steve Souders, qui travaille sans relâche sur le test de la performance a étudié l'impact des sélecteurs[7] CSS sur la performance et a déterminé en 2009 que la variation de temps entre le pire des cas et le meilleur était de 50ms. En d'autres termes, prenez en considérations la performance des sélecteurs mais ne perdez pas trop de temps la dessus.

7.http://www.stevesouders.com/blog/2009/03/10/performance-impact-of-css-selectors/

HTML5 ET SMACSS

Alors qu'on approche de la fin, sachez que l'approche SMACSS fonctionne tout aussi bien avec HTML5 qu'avec HTML4 ou tout autre HTML que vous utilisez avec votre CSS. Ceci, parce l'approche SMACSS à deux objectifs :

1. Accroître la valeur sémantique d'une section comportant du HTML et du contenu
2. Diminuer les contraintes engendrées par une structure HTML spécifique

L'HTML5 introduit plusieurs éléments nouveaux qui peuvent nous aider à accroître la valeur sémantique d'une section comportant du HTML et du contenu. Les tags comme `section`, `header`, et `aside` sont plus descriptifs qu'une simple div. Nous avons de nouveaux types d'input qui nous permettent de différentier un champ numérique d'un champ de date d'un champ de texte etc. Les tags et les attributs supplémentaires nous permettent d'être plus descriptifs. Ce qui est une bonne chose.

Cependant, tous les tags, même les nouveaux, ne sont pas nécessairement descriptifs d'un module particulier sur la page. Un élément `nav` va-t-il toujours contenir le même type et style de navigation ?

<nav> implementation

```
<nav class="nav-primary">
    <h1>Navigation primaire</h1>
    <ul>…</ul>
</nav>

<nav class="nav-secondary">
    <h1>Liens extérieurs</h1>
    <ul>…</ul>
</nav>
```

La navigation primaire utilise une navigation horizontale en haut de la page, mais la navigation secondaire, pour une barre latérale par exemple, liste les éléments verticalement. Les noms de classe permettent de différentier ces types de navigation.

Ils aident à décrire notre contenu de manière très spécifique – par des moyens encore plus spécifiques que les nouveaux éléments du HTML5. Cela sert donc l'intérêt de notre premier objectif qui est d'accroître la valeur sémantique d'une section HTML.

Votre premier réflexe pourrait être de faire quelque chose qui ressemblerait à cela :

Les éléments CSS<nav>

```
nav.nav-primary li {
    display: inline-block;
}

nav.nav-secondary li {
display: block;
}
```

En faisant ainsi, nous indiquons que ces classes ne peuvent être utilisées que sur les éléments `nav`. Si vous savez que votre code ne sera jamais amené à changer, alors c'est bon. Cependant, l'intention de ce livre est de décrire des projets évolutifs, donc re-

gardons un exemple de la manière dont les choses peuvent évoluer dans ce projet.

Notre navigation primaire ne présente qu'un seul niveau, mais le client revient pour demander à ce que nous ajoutions des menus déroulants. Notre structure HTML va donc devoir être modifiée.

L'implémentation <nav>

```
<nav class="nav-primary">
    <h1>Navigation primaire</h1>
    <ul>
        <li>Qui nous sommes
            <ul>
                <li>L'équipe</li>
                <li>Nous retrouver</li>
            </ul>
        </li>
    </ul>
</nav>
```

Avec cette sous-navigation, de quelle manière pouvons-nous styler les éléments afin qu'ils apparaissent verticalement et non horizontalement ?

Avec la CSS que nous avons déjà, nous devrions ajouter un `<nav class="nav-secondary">` autour de chaque liste ul pour obtenir le style que nous souhaitons.

Nous pouvons augmenter la CSS pour cibler les éléments internes de la liste.

CSS <nav> augmentée

```
nav.nav-primary li {
    display: inline-block;
}
nav.nav-secondary li,
nav.nav-primary li li {
    display: block;
}
```

Une autre alternative consiste à ôter la nécessité d'un élément `nav` auquel appliquer nos classes. Cela va dans le sens de notre deuxième objectif qui est de réduire les contraintes engendrées par un HTML spécifique.

SMACSS-style <nav> CSS

```
.l-inline li {
    display: inline-block;
}
.l-stacked li {
    display: block;
}
```

Dans ce cas, nous avons opté pour des règles d'agencement, puisque nous affectons la manière dont les modules individuels (la liste d'éléments) doivent se positionner. La classe `.l-stacked` peut s'appliquer à la sous-navigation `ul`. Cela va créer le résultat désiré.

Spécifier la liste d'éléments comme étant un élément enfant requis lie tout de même encore les règles d'agencement aux éléments spécifiques du HTML. Comme le dit l'expression, il y a certainement plusieurs moyens d'accommoder le lapin. Par exemple, vous pouvez vouloir dire que tous les éléments enfants vont porter ce style.

SMACSS-style <nav> CSS

```
.l-inline > * {
    display: inline-block;
}

.l-stacked > * {
    display: block;
}
```

La faille de cette approche est que les règles vont devoir être évaluées pour chaque élément de la page et pas seulement pour les éléments de la liste. Cibler uniquement les descendants directs évite beaucoup d'allers-retours pour le navigateur. Cela nous permet d'utiliser les classes `inline` et `stacked` sur presque tous les éléments dont nous souhaitons que les éléments enfants adoptent ces styles.

L'implémentation <nav>

```
<nav class="l-inline">
    <h1>Navigation primaire</h1>
    <ul>
        <li>Qui nous sommes
            <ul class="l-stacked">
                <li>L'équipe</li>
                <li>Nous retrouver</li>
            </ul>
        </li>
    </ul>
</nav>
```

Même avec cet exemple plutôt évident, nous avons réussi à garder une CSS simple et évité de rendre nos sélecteurs plus complexes. Et le HTML est encore compréhensible.

Souvenez-vous des deux objectifs que nous visons dans l'approche SMACSS : augmenter les sémantiques et diminuer la dépendance à un HTML spécifique.

CONCEPTION D'UN PROTOTYPE

Les bons programmeurs aiment les modèles. Les bons designers aussi. Les modèles permettent d'établir une familiarité et encouragent leur réutilisation. SMACSS cherche à identifier les modèles de votre design et à les codifier de manière à les rendre réutilisables.

Un prototype devrait aider à visualiser les composants de manière partielle ou globale et permettre la codification des éléments de design en blocs. L'industrie du webdesign apprécie les composants réutilisables, ce que nous pouvons voir dans de nombreux systèmes tels que Bootstrap[8], qui propose une variété de composants de sites, et 960.Gs[9], qui propose des grilles d'agencement.

Sur Yahoo!, l'équipe de conception de prototype a créé ses blocs de construction et les utilise pour la production. Cela permet une plus grande homogénéité entre les différents produits, dans la mesure où ils sont tous construits sur les mêmes fondements.

Les objectifs d'un prototype

Un prototype peut servir différents objectifs :

- Montrer des états
- Vérifier la localisation
- Isoler les dépendances

8.http://twitter.github.com/bootstrap/
9.http://960.gs/

Les états

De l'état par défaut, à l'état enroulé, à l'état d'erreur, à quel que soit l'état que vous avez défini, il est important de pouvoir les visualiser chacun et de pouvoir s'assurer que le module est construit avec précision.

Si un module fonctionne sur des données, alors des données réelles ou simulées peuvent être utilisées dans votre prototype pour pouvoir tester le rendu.

La localisation

Pour les projets s'étendant sur plusieurs pays, il sera très intéressant de pouvoir tester les modules en utilisant différentes conditions, correspondant aux différentes localisation, pour nous assurer que les agencements ne se cassent pas.

Les dépendances

Pour finir, il est important d'isoler les dépendances. Quelles dépendances CSS et JavaScript sont nécessaires pour le bon rendu d'un module ? Dans les gros projets où le chargement différé est utilisé, être en mesure d'isoler les dépendances et les diminuer au minimum requis signifie que vous avez construit un module efficace et pertinent, qui pourra être intégré dans le site sans impacter négativement les autres modules de la page.

Sur Yahoo!, les modules sont isolés en fichiers CSS individuels et sont groupés à l'aide d'un combo handler (technique permettant de réduire le nombre de requêtes HTTP pour nos fichiers CSS et JavaScript) lorsque cela est nécessaire. Par exemple, lorsque la boîte de réception charge, elle combine les fichiers CSS pour charger les boutons, la liste de message, la barre latérale, les tabulations et l'en-tête. Au moment où l'utilisateur demande la page de recherche, le combo-handler combine les styles de recherche spécifiques puis les délivre. Le module de recherche utilise une variante

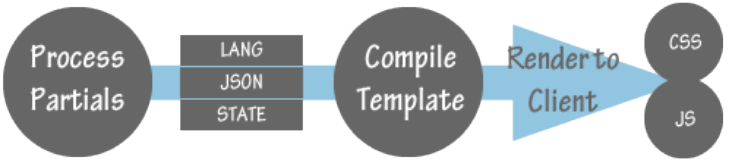
des styles de la liste de messages, par défaut, et de la barre latérale, ce qui signifie qu'il ne lui reste plus qu'à charger les modules sous-classés.

Les pièces du puzzle

Sur Yahoo!, nous avons construit un moteur de conception de prototype pour nous faciliter la tâche. C'est la taille de votre projet qui définit si vous avez besoin d'un tel outil ou non.

Le concepteur de prototype utilise un Template Mustache[10] à sa racine. Des données simulées sont stockées dans un fichier JSON, les conditions de localisation (localization strings) sont stockées par paires clef/valeur dans un fichier texte, et les dépendances CSS et JavaScript y sont intégrées selon les besoins. Cela permet à l'équipe de visualiser un menu, une fenêtre modale ou un formulaire de manière isolée ou dans le contexte du site entier. En faisant ainsi, chacun peut vérifier certaines fonctionnalités et certains designs avant de se lancer dans l'aspect technique. Nous pouvons lancer le développement de certains modules et harmoniser le design par la suite.

Le concepteur de prototype de Yahoo!



Dans le cas de notre concepteur de prototype, une partie de la gestion des états est gérée avant que le module ne soit généré : les éléments conditionnels, la filtration des données et tout ce qui pourrait normalement être traité par le serveur. La gestion des états ne

10.http://mustache.github.com/

se limite pas simplement à un changement de nom de classe sur un élément HTML.

Votre prototype

Avoir un concepteur entièrement développé pour compiler vos modules n'est peut-être pas vraiment nécessaire, particulièrement pour un petit site. Il est tout de même avantageux d'isoler vos composants dans un format facile à visualiser. MailChimp, par exemple, possède une feuille interne de modèles[11] de design pour la simulation. Ils l'utilisent pour construire le site. Cette feuille contient différents modèles utilisés dans tout le site, ainsi que le code utilisé pour chaque module.

Souvenez-vous de cela : avoir un modèle, c'est bien. Codifier ces modèles aussi. Mais mettre en place un procédé qui permet de voir et de tester ces modèles, c'est super !

11.http://www.flickr.com/photos/aarronwalter/5579386649/

LES PRÉPROCESSEURS

Aussi bien que soit le CSS, il manque des fonctionnalités que beaucoup de designers et développeurs aimeraient avoir. Pour pallier ce manque – et permettre d'accélérer le développement – des outils ont été créés pour nous simplifier la vie.

L'un de ces outils est le préprocesseur CSS. Je vais expliquer ce qu'est un préprocesseur, ce qu'il peut faire, et de quelle manière il peut vous aider à construire un code CSS évolutif et modulaire.

Qu'est-ce qu'un préprocesseur ?

Un préprocesseur CSS vous permet d'utiliser une syntaxe spéciale dans votre CSS, qui est ensuite compilée dans votre projet. Certains préprocesseurs essaient d'être aussi proches que possible de la syntaxe CSS réelle, tandis que d'autres essaient de simplifier les choses autant que possible.

Regardez cet exemple utilisant le préprocesseur Stylus[12].

Code utilisant Stylus

```
@import 'vendor'
body
  font 12px Helvetica, Arial, sans-serif

a.button
  border-radius 5px
```

12.http://learnboost.github.com/stylus/

Pour ceux qui connaissent Ruby[13] ou CoffeeScript[14], cela va vous sembler familier. Le préprocesseur Stylus enlève les accolades et les point-virgules qu'il remplace par des espaces. C'est l'indentation qui indique quelle propriété s'applique à quelle règle. Les noms des propriétés ne comportent jamais d'espace ce qui signifie que le premier espace suivant le nom de la propriété doit forcément séparer la propriété de la valeur.

En plus de Stylus, les deux préprocesseurs en tête du marché sont Sass[15] (ou Compass[16]) et Less[17].

Installer un préprocesseur

Les designers en particulier, qui ne sont pas tellement familiers des lignes de commande, se plaignent fréquemment de la complexité d'installation des préprocesseurs. En fonction de votre environnement, ils peuvent effectivement l'être. Mais ils peuvent également être aussi simples à installer qu'une application à placer dans votre dossier d'applications.

Pour installer Sass sur Mac, par exemple, j'ai simplement ouvert le terminal et écrit :

Installer Sass

```
sudo gem install sass
```

Ou installer Compass

```
sudo gem install compass
```

13.http://www.ruby-lang.org/fr/
14.http://coffeescript.org/
15.http://sass-lang.com/
16.http://compass-style.org/
17.http://http://lesscss.org/

Gem[18] est un outil de ligne de commande qui provient du monde de Ruby et qui agit comme un gestionnaire de package pour installer des applications. Il est pré-installé sur les versions récentes de Mac OS X.

Une fois que Sass est installé sur ma machine, je lui demande d'analyser un dossier sur lequel je travaille.

Faire fonctionner Sass

```
sass --watch before:after
```

`Before` est le dossier où tous mes fichiers `.scss` (fichiers CSS pré-compilés) sont contenus et `after` est le dossier où sont placés tous les fichiers CSS après compilation. Tous les changements que j'effectue dans mes fichiers `.scss`, dans le dossier before, sont automatiquement compilés et placés dans dans le dossier `after` au format `.css`. C'est très pratique pour construire et tester les choses rapidement pendant le développement. (Un fichier `.scss` est comme un fichier `.css` normal sauf qu'il utilise la syntaxe Sass que je vais expliquer plus tard dans ce chapitre.)

Il existe également l'application Compass Mac[19] qui évite l'utilisation de la ligne de commande.

Less utilise npm, le node package manager[20], qui n'est pas pré-installé. C'est pourquoi vous allez d'abord devoir l'installer avant d'installer Less. Less a également une version JavaScript qui peut fonctionner côté-client pendant le processus de développement.

L'implémentation de Less côté client

```
<link rel="stylesheet/less" type="text/css"
href="styles.less">
<script src="less.js"></script>
```

18.http://rubygems.org/
19.http://compass.handlino.com/
20.https://github.com/isaacs.npm

Souvenez-vous de ne pas utiliser cela sur un site réel. Compilez toujours le CSS avant la mise en production.

Ligne de commande Less pour lancer la Compilation

```
lessc styles.less
```

Le développement de site nécessite de plus en plus des outils en ligne de commande. Ils peuvent être un moyen pratique d'optimiser vos procédés et même, de vous sortir de situations complexes que les outils avec une interface graphique ne peuvent parfois pas résoudre.

Les fonctionnalités pratiques d'un préprocesseur

Les préprocesseurs proposent de nombreuses fonctionnalités intéressantes pouvant faciliter le codage du CSS. En voici quelques-unes :

- Les variables
- Les opérations
- Les mixins
- Les imbrications
- Les fonctions
- Les interpolations
- L'importation de fichiers
- Les extensions

Examinons certaines de ces fonctionnalités. (Je vais utiliser Sass pour les exemples qui suivent, mais sachez que Less et Stylus ont également des approches similaires pour les mêmes concepts.)

Les variables

Tous ceux qui ont déjà travaillé avec CSS pendant plus d'une heure ont eu envie de créer une fonctionnalité permettant d'établir la valeur d'une couleur une seule fois dans le fichier CSS, puis de pouvoir la réutiliser n'importe où dans la feuille de style. Avec Sass, vous pouvez définir une variable, en plaçant le signe
$

devant le mot et en lui assignant une valeur.

Utilisation des variables

```
$color: #369;

body {
    color: $color;
}

.callout {
    border-color: $color;
}
```

Le compilateur va ensuite convertir ce bout de code en un fichier final pour la création du site.

Les variables compilées

```
body {
    color: #369;
}

.callout {
    border-color: #369;
}
```

C'est très pratique pour centraliser les grands changements d'un site. (Pour information, le W3C travaille sur une ébauche de spécification des variables CSS[21].)

21.http://dev.w3.org/csswg/css-variables/

Les imbrications

Lorsque vous codez du CSS, il est assez commun d'avoir une chaîne de sélecteurs.

Une chaîne de sélecteurs

```
.nav > li {
    float: left;
}

.nav > li > a {
    display: block;
}
```

L'imbrication permet à ces styles d'être regroupés de manière plus claire dans des fichiers CSS pré-compilés.

L'imbrication avec Sass

```
.nav {
    > li {
        float: left;
        > a {
            display: block;
        }
    }
}
```

Chaque ensemble de style est imbriqué dans l'ensemble supérieur. À quoi cela ressemble-t-il une fois généré ?

Génération du CSS à partir du Sass

```
.nav > li {
  float: left; }
  .nav > li > a {
    display: block; }
```

L'imbrication aide à clarifier quels styles s'appliquent à quels éléments mais pas beaucoup plus que si vous faisiez les indentations vous-même. Nous nous épargnons simplement de l'écriture en évitant de répéter `.nav` à chaque fois.

Les mixins

Un mixin est une fonctionnalité très pratique qui permet de regrouper des styles pouvant être réutilisés dans vos CSS. Ils peuvent comporter des paramètres qui vont personnaliser le résultat de ce mixin. L'une des utilisations les plus fréquentes des mixins se trouve dans la gestion des préfixes constructeurs (spécifiant l'éditeur du navigateur), même s'ils peuvent être utilisés pour toute règle CSS répétitive.

Un exemple de mixin pour `border-radius`

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

Une fois que vous avez déclaré un mixin, vous pouvez ensuite l'appeler de n'importe où dans votre CSS en utilisant la commande `include`.

Un exemple de Mixin pour `border-radius`

```
.callout {
    @include border-radius(5px);
}
```

Le préprocesseur va ensuite compiler ceci en cela :

CSS généré à partir du `mixin` `border-radius`

```
.callout {
  -webkit-border-radius: 5px;
     -moz-border-radius: 5px;
          border-radius: 5px;
}
```

Les fonctions

L'exemple ci-dessus ressemble déjà à une fonction. Il existe cependant la possibilité de faire des choses très puissantes en calculant des valeurs. Par exemple, la fonction `lighten` reprend la valeur d'une couleur et lui attribue un pourcentage pour ajuster la luminosité de cette valeur.

Ajustement de la valeur d'une couleur à l'aide d'une fonction

```
$btnColor: #036;
.btn {
    background-color: $btnColor;
}
.btn:hover {
    background-color: lighten($btnColor, 20%);
}
```

Le préprocesseur va ensuite compiler ceci en cela :

CSS compilé avec des fonctions de couleur

```
.btn {
    background-color: #003366;
}
.btn:hover {
    background-color: #0066cc;
}
```

Sass apporte de nombreuses fonctionnalités pratiques comme celles détaillées ci-dessus, et Compass en ajoute encore d'autres. *(Si*

vous êtes amené à utiliser Sass, je vous recommande fortement d'utiliser également les avantages de Compass.)

Les extensions

Les extensions permettent d'étendre un module avec les propriétés d'un autre. Dans Sass, cela se fait avec la commande `extend`.

Extensions Sass

```
.btn {
    display: block;
    padding: 5px 10px;
    background-color: #003366;
}
.btn-default {
    @extend .btn;
    background-color: #0066cc;
}
```

L'extension prend le style du `btn` et l'applique au `btn-default`. Sass est tout de même intelligent, il ne se contente pas de dupliquer simplement les règles dans la seconde déclaration. Il crée une combinaison de sélecteurs pour le premier ensemble de règles :

CSS compilé à partir des extensions de Sass

```
.btn, .btn-default {
  display: block;
  padding: 5px 10px;
  background-color: #003366; }

.btn-default {
  background-color: #0066cc; }
```

Les extensions sont limitées aux sélecteurs simples. Vous ne pourriez étendre `#main .btn` par exemple. Nous reviendrons sur les extensions et la manière dont elles impactent l'approche SMACSS plus loin dans ce chapitre.

Et même plus

Ceci n'est que la partie visible de l'iceberg en matière de préprocesseurs. Il existe encore de nombreuses autres fonctionnalités et exemples que nous pourrions trouver sur différents sites. Les préprocesseurs peuvent faire un peu peur au départ, mais ne vous sentez pas obligé d'utiliser immédiatement toutes les fonctionnalités.

Se mettre dans le pétrin et s'en sortir

Vous connaissez le dicton qui dit : « Un grand pouvoir implique de grandes responsabilités ». Ces préprocesseurs proposent de nombreuses et bonnes fonctionnalités permettant de maintenir vos fichiers CSS pré-compilés dans un état propre et lisible. Cependant, une fois compilé, la magie peut résulter en un CSS bloqué, difficile à débugger. En d'autres termes, vous êtes de retour à la case départ. Cela peut se produire quelle que soit votre manière de coder – que ce soit à la main, à l'aide d'un outil WYSIWYG comme Dreamweaver, ou par l'intermédiaire d'un préprocesseur en ligne de commande. Il est également possible de créer un bon code en utilisant l'un de ces outils.

Voyons ce qui engendre ces blocages.

L'imbrication profonde

Lorsque vous commencez à imbriquer, vous pouvez facilement aller trop en profondeur. Imaginez, par exemple, ce type de code :

Imbrication profonde dans Sass

```
#sidebar {
  width: 300px;
  .box {
    border-radius: 10px;
    background-color: #EEE;
    h2 {
      font-size: 14px;
      font-weight: bold;
    }
    ul {
      margin:0;
      padding:0;
      a {
        display:block;
      }
    }
  }
}
```

Ce cas de figure n'est pas rare. Je l'ai déjà vu plusieurs fois en travaillant sur des fichiers Sass existants. Et voici le CSS qui est généré :

CSS compilé après avoir utilisé une imbrication profonde dans Sass

```
#sidebar {
  width: 300px; }
  #sidebar .box {
    border-radius: 10px;
    background-color: #EEE; }
    #sidebar .box h2 {
      font-size: 14px;
      font-weight: bold; }
    #sidebar .box ul {
      margin: 0;
      padding: 0; }
      #sidebar .box ul a {
        display: block; }
```

L'imbrication dans l'approche SMACSS

L'approche SMACSS, par sa nature même, diminue l'imbrication profonde due à la profondeur de l'applicabilité. La séparation des agencements et des modules permet également d'éviter ces problèmes. Avec SMACSS, l'exemple précédant ressemblerait plus à cela :

Imbrication profonde avec SMACSS

```
#sidebar {
  width: 300px;
}

.box {
  border-radius: 10px;
  background-color: #EEE;
}

.box-header {
  font-size: 14px;
  font-weight: bold;
}

.box-body {
  margin: 0;
  padding: 0;
  a {
    display: block;
  }
}
```

Il n'y a presque aucune imbrication ! Tout simplement parce que nous n'avons pas besoin de longs sélecteurs pour obtenir les styles désirés. C'est uniquement lorsque nous avons besoin de cibler des sélecteurs d'éléments dans une partie du module que l'imbrication devient nécessaire.

Créer de longues chaînes de sélecteurs rend le travail du navigateur plus difficile pour déterminer si un style s'applique à l'élément en cours de rendu.

L'extension inutile

Retournons à notre exemple d'extension d'un bouton par défaut qui reprend les styles de celui-ci. Regardons à nouveau ce code.

L'extension d'un bouton dans Sass

```
.btn {
    display: block;
    padding: 5px 10px;
    background-color: #003366;
}
.btn-default {
    @extend .btn;
    background-color: #0066cc;
}
```

Ce code CSS nous permet de produire le code HTML suivant, éliminant la nécessité de spécifier les classes `btn` et `bnt-default` sur un élément. Seul l'une des deux doit être spécifiée. Nous déplaçons les déclarations multiples du HTML au CSS.

Application d'une classe sur un lien

```
<a class="btn-default">My button</a>
```

Étendre les modules pour créer des sous-modules est un moyen d'éviter de définir plusieurs classes dans le code HTML. La convention de nommage devient alors plus importante dans cette situation. Avoir un nom de module `btn` et un nom de sous-module `small` compliquerait les choses dans ce cas où seule la classe `small` devrait être appliquée. Mais à quoi s'applique l'attribut `small` ? Avec `btn-small`, nous pouvons utiliser le nom du sous-module lui-même, tout en connaissant l'effet qu'il est sensé produire.

Lorsque nous regardons le fichier source SASS, nous pouvons également voir que le `btn-default` est un sous-module grâce à

l'utilisation de `@extend`. Dans le code généré, nous pouvons toujours voir que le `btn-default` est un sous-module parce qu'il sera groupé avec la classe `btn`.

Par contre, cette méthode n'est plus pertinente lorsque nous faisons des extensions à travers différents modules.

Extension Sass à travers différents modules

```
.box {
    border-radius: 5px;
    box-shadow: 0 0 3px 0 #000000;
    background-color: #003366;

}
.btn {
    @extend .box;
    background-color: #0066cc;
}
```

L'extension à travers différents modules nous pose problème. La source des styles d'un module dépend d'un autre module qui n'a pas forcément de rapport avec lui.
En regroupant tout dans le CSS, vous perdez la possibilité de faire un chargement en temps réel ou de faire des compilations conditionnelles en fonction des composants que vous devez charger dans votre application. Les styles du bouton et de la boîte doivent être chargés au même moment.

L'extension dans un seul et même module peut elle-même complexifier un projet qui fonctionne avec le chargement des styles en temps réel. Par exemple, chez Yahoo!, nous chargeons les styles des boutons par défaut au moment du chargement de la page mais nous ne chargeons les styles des boutons secondaires – comme ceux des composants – que lorsque la fenêtre en question est demandée. Cela permet d'obtenir et de maintenir un temps de chargement initial de la page très court.

Les extensions SMACSS

Avec l'approche SMACSS, les extensions ne sont gérées qu'au niveau du HTML et non pas au niveau du CSS, grâce à la définition de plusieurs classes dans le code HTML.

Application des classes de module SMACSS sur un bouton

```
<a class="btn btn-default">My button</a>
```

Cela donne l'avantage supplémentaire de pouvoir déterminer où se trouve l'élément racine d'un module. Lorsque nous analysons du code HTML compilé dans un navigateur, il peut être difficile de déterminer où commence un module et où en finit un autre. Mais puisque les noms des modules racines ne comportent aucun trait d'union, ils se distinguent et sortent de la masse.

On peut se demander si c'est pertinent dans la mesure où plusieurs classes (en apparence non nécessaires) sont ajoutées au code HTML. Cependant, ces classes ne sont pas inutiles. Elles clarifient l'intention et augmentent la sémantique de l'élément en question.

Abus de mixins

L'utilisation des mixins est un bon moyen d'éviter la répétition. Cependant, l'utilisation des classes l'est aussi. Si le même CSS s'applique à différents endroits, il pourrait être intéressant de déplacer ce style dans sa propre classe.

Admettons que tout un groupe de modules partage un style visuel similaire : un fond gris et une bordure bleue arrondie. Vous pouvez décider de créer un mixin pour celui-ci.

Un mixin Sass pour un modèle commun

```
@mixin theme-border {
  border: 2px solid #039;
  border-radius: 10px;
  background-color: #EEE;
}

.callout {
  @include theme-border;
}

.summary {
  @include theme-border;
}
```

Voici ce que cela donnera, une fois compilé :

CSS compilé à parti d'un mixin

```
.callout {
  border: 2px solid #039;
  border-radius: 10px;
  background-color: #EEE; }

.summary {
  border: 2px solid #039;
  border-radius: 10px;
  background-color: #EEE; }
```

L'approche SMACSS pour les modèles répétitifs

Dans ce cas, étant donné que c'est un traitement visuel appliqué à différents conteneurs, il vaut la peine de placer ce modèle dans sa propre classe.

Définition d'une classe pour un modèle commun

```
.theme-border {
  border: 2px solid #039;
  border-radius: 10px;
  background-color: #EEE;
}

.callout {
}

.summary {
}
```

Puis, nous appliquons cette classe aux éléments qui requièrent cet habillage.

Application de la classe

```
<div class="callout theme-border"></div>
<div class="summary theme-border"></div>
```

Les mixins paramétrés

Pour être clair, les mixins paramétrés proposent de nombreuses fonctionnalités qu'il n'est pas possible d'avoir avec le CSS seul, et il n'existe aucune approche équivalente pour résoudre cela, à part celle de créer de nombreuses variations. L'exemple du mixin `border-radius` ci-dessus est un bon exemple d'un mixin paramétré.

À consommer avec modération

Nous avons vu quelques difficultés fréquentes liées à l'utilisation des préprocesseurs tout en découvrant comment l'approche SMACSS pouvait y répondre. En conclusion, nous pouvons dire qu'il est bon de les utiliser, mais avec modération. Analysez vos fichiers

générés et voyez si le résultat est bien celui auquel vous vous attendiez. S'il y a de nombreuses répétitions alors pensez à revoir votre approche.

Découvrons encore quelques moyens par lesquels un préprocesseur permet une meilleure modularisation de votre code.

Les Media Queries basées sur l'état et l'imbrication

Comme nous l'avons vu dans le chapitre « Changement d'état », les Media Queries sont l'un des moyens de détecter et de gérer les changements d'état. La plupart des tutoriels utilisent une feuille de styles séparée et regroupent tous les styles relatifs à cet état dans un fichier. Cela implique la séparation du module en différents fichiers, ce qui rend la gestion plus complexe.

Sass permet aux Media Queries d'être imbriquées, permettant à ces changements d'état d'apparaître au bon endroit : au niveau du module.

Voici un exemple montrant une Media Query imbriquée :

Media Query imbriquée dans Sass

```
.nav > li {
  width: 100%;

  @media screen and (min-width: 320px) {
      width: 100px;
      float: left;

  }

 @media screen and (min-width: 1200px) {
          width: 250px;
      }

}
```

L'état par défaut est défini, auquel suivent les états alternatifs de ce module, définis directement à l'intérieur de ce dernier. Vous pouvez même imbriquer des Media Queries dans d'autres Media Queries et Sass fera le lien entre les différentes conditions des Media Queries.

Voici le code de l'exemple ci-dessus, une fois généré :

CSS compilé à partir d'une Media Query imbriquée dans Sass

```
.nav > li {
   width: 100%; }
@media screen and (min-width: 320px) {
  .nav > li {
    width: 100px;
    float: left; } }
  @media screen and (min-width: 1200px) {
    .nav > li {
      width: 250px; } }
```

Sass crée la Media Query séparée et y inclut les sélecteurs. Dans cet exemple précis, j'ai défini l'état par défaut comme étant la vue small screen, qui correspondrait à tout écran inférieur à 320px. Puis, je passe à une navigation fluide une fois que la largeur spécifique est atteinte. Finalement, la largeur est redéfinie à 1200px sans pour autant que le float doive être à nouveau déclaré. J'aime le fait que cet héritage se transmette à partir de l'état par défaut dans les différentes Media Queries.
Mais le mieux, c'est que tous les états alternatifs de votre module soient déclarés dans le module.

L'organisation de vos fichiers

Les préprocesseurs encouragent la même séparation des éléments clefs que SMACSS recommande.

Voici quelques conseils sur la manière de séparer les fichiers de votre projet :

- Placez toutes les règles de base dans leur propre fichier.
- En fonction du type d'agencement que vous avez, placez-les soit tous dans un seul fichier ou bien placez les agencements principaux dans des fichiers séparés.
- Placez chaque module dans son propre fichier.
- En fonction de la taille de votre projet, placez les sous-modules dans leur propre fichier.
- Placez les états généraux dans leur propre fichier.
- Placez les états des modules et des agencements, y compris les Media Queries qui les affectent, dans les fichiers du module.

Séparer les fichiers de cette manière rend votre projet plus facile à prototyper. Des gabarits HTML peuvent être créés pour des composants individuels, et le CSS et les gabarits de chaque composant individuel (ou même d'une sous-catégorie de composants) peuvent être testés de manière isolée.

Les composants propres au préprocesseur comme les mixins et les variables devraient également être spécifiés dans leur propre fichier.

Un exemple de structure

```
+-agencement/
| +-grid.scss
| +-alternate.scss
+-module/
| +-callout.scss
| +-bookmarks.scss
| +-btn.scss
| +-btn-compose.scss
+-base.scss
+-states.scss
+-site-settings.scss
+-mixins.scss
```

Pour finir, créez le fichier CSS primaire qui inclura les autres fichiers. Pour de nombreux sites, cela signifie simplement inclure

tous les fichiers dans la feuille de style principale. Pour les projets comportant des conditions de chargement des données, vous pouvez avoir des fichiers conteneurs qui importent seulement les fichiers nécessaires pour les différentes pages.

Dans le fichier principal

```
@import
   "site-settings",
   "mixins",
   "base",
   "states",
   "layout/grid",
   "module/btn",
   "module/bookmarks",
   "module/callout";
```

Le pré-compilateur va compiler cela en un fichier unique pour le développement ou le déploiement du site.

Lorsque vous êtes prêt pour le lancement, créez une version compressée de votre fichier CSS pour le déploiement. (Votre environnement peut avoir créé des scripts pour déployer le reste du site. Assurez-vous d'intégrer la compilation du préprocesseur dans cette dernière étape de la construction.)

Ligne de commande pour fichier CSS compressé dans Sass

```
sass -t compressed master.scss master.css
```

Conclusion sur les préprocesseurs

Nous avons vu ce qu'est un préprocesseur, comment l'installer et l'utiliser. Nous avons étudié quelques-unes de ses fonctionnalités les plus appréciées, ainsi que quelques-unes des difficultés que nous pouvons rencontrer dans son utilisation. Pour finir, nous avons vu de quelle manière les préprocesseurs pouvaient vous fa-

ciliter l'organisation de votre projet. Les préprocesseurs peuvent clairement être un atout dans le développement de votre projet.

LAISSEZ TOMBER LA BASE

Il y a quelques éléments – peu mais tout de même – qui sont utilisés très souvent. C'est la raison pour laquelle vous pouvez penser (comme moi) qu'il est donc sûr de les styler en tant que règles de base, pensant que leur rôle est unique et ne changera jamais. Comme vous avez dû le comprendre dans les pages précédentes, les choses changent. Nous pouvons planifier en prenant en compte ce changement afin qu'il ne complique pas le travail que nous avons déjà fait.

Quels éléments sont concernés par ce problème potentiel ? Les éléments `button`, `table` et `input` sont les plus fréquents d'après mon expérience. Regardons un exemple de ce qui se passe souvent dans un projet.

L'élément Table

Il est loin le temps où nous utilisions les tableaux de manière standardisée pour construire nos agencements. C'est la raison pour laquelle l'utilisation de l'élément table dans un projet n'est souvent pas nécessaire.

Jusqu'au jour où, elle l'est.

Le premier et unique tableau nécessaire est celui qui nous sert à faire apparaître un certain ensemble de données, comme les tableaux de comparaison par exemple. Le tableau de comparaison

comporte une marge interne, un alignement de colonnes, et des bordures spécifiques. Tout semble normal jusque là.

Styles du tableau

```
table {
    width: 100%;
    border: 1px solid #000;
    border-width: 1px 0;
    border-collapse: collapse;
}

td {
    border: 1px solid #666;
    border-width: 1px 0;
}

td:nth-child(2n) {
    background-color: #EEE;
}
```

Quelques jours, quelques semaines ou quelques mois plus tard, un nouveau tableau est nécessaire. Cette fois-ci en revanche, son objectif est différent et il faut que les en-têtes soient à gauche et les données à droite. Les bordures sont enlevées et les arrières-plans modifiés. Normalement, nous devrions utiliser ici une redéfinition des styles par défaut.

Redéfinition des styles précédents

```
table.info {
    border-width: 0;
}

td.info {
    border-width: 0;
}

td.info:nth-child(2n) {
    background-color: transparent;
}

.info > tr > th {
    text-align: left;
}

.info > tr > td {
    text-align: right;
}
```

Le problème est que nous avons dû redéfinir les styles parce que les règles de base n'étaient prévues que pour un seul objectif. Les règles de base devraient se cantonner au style par défaut et être ensuite redéfinies pour des modules spécifiques. Le tableau de comparaison était un module. Il avait un objectif précis et unique auquel correspondait un design particulier, même s'il était la seule occurrence des éléments présents dans ce module.
La solution est donc maintenant évidente : créez un module.

Mieux vaut créer un module

```
table {
    width: 100%;
    border-collapse: collapse;
}

.comparison {
    border: 1px solid #000;
    border-width: 1px 0;
}

.comparison > tr > td {
    border: 1px solid #666;
    border-width: 1px 0;
}

.comparison > tr > td:nth-child(2n) {
    background-color: #EEE;
}

.info > tr > th {
    text-align: left;
}

.info > tr > td {
    text-align: right;
}
```

L'élément tableau conserve certains styles de base. J'ai rarement eu besoin d'étendre les caractéristiques d'un tableau afin de pouvoir remplir ses conteneurs. Je ne me souviens pas non plus ne pas avoir utilisé `border-collapse: collapse.` Et je me dis que cette règle devrait faire partie des règles par défaut des navigateurs.

Notre module de comparaison est maintenant isolé comme il se doit. J'ai spécifié les sélecteurs d'enfant pour garder l'impact aussi faible que possible. Si j'avais besoin d'imbriquer un tableau dans le tableau (ce qu'il faudrait en principe éviter) alors je serais assuré que le module de comparaison n'impacterait pas le tableau intégré à l'intérieur. Notre module d'information est maintenant simplifié à seulement deux règles.

En plus, nous avons utilisé moins de code CSS pour atteindre le résultat désiré et nous avons aussi gagné en clarté au niveau de notre code. Gagnant – Gagnant !

Comme je l'ai dit auparavant, les éléments `button` et `input` peuvent souvent avoir le même destin que les tableaux. Si vos styles ont un objectif très précis, créez un module. Cela permettra d'éviter de redéfinir les styles et de réécrire un ancien code.

LE MODULE ICÔNE

Les Sprites CSS sont devenus un pilier du développement web moderne – et il y a une bonne raison à cela. Ils permettent de rassembler de multiples données en une seule source, diminuant ainsi le nombre de requêtes HTTP. Cela permet aussi que les images, pour les états de survol par exemple, soient déjà chargées au moment où l'on en a besoin.

Avant que les sprites n'existent, les images étaient utilisées dans deux contextes différents : les images de fond, sur lesquelles étaient disposés les différents éléments, et les images de premier plan, au même niveau que le texte. Avec les sprites, toutes les images deviennent des images de fond positionnées dans le masque de l'élément.

Ce chapitre traitera spécifiquement de ce dernier cas de figure.

Prenons l'exemple d'un menu contenant des icônes pour illustrer ce propos :

Menu objet avec des icônes

Menu HTML

```
<ul class="menu">
    <li class="menu-inbox">Inbox</li>
    <li class="menu-drafts">Drafts</li>
</ul>
```

Le code HTML est tout simple. Nous avons une liste comportant les éléments du menu. Une classe est ajoutée à chaque élément du menu afin que nous puissions appliquer à chacun un style différent.

Menu CSS

```
.menu li {
    background: url(/img/sprite.png) no-repeat 0 0;
    padding-left: 20px;
}

.menu .menu-inbox {
    background-position: 0 -20px;
}

.menu .menu-drafts {
    background-position: 0 -40px;
}
```

Tous ces éléments sont attachés au sprite unique, et pour chacun d'eux, le style repositionne le fond pour laisser apparaître la bonne icône.

En apparence, cela s'affiche correctement et a plutôt bien fonctionné pour nous, en grande partie. Mais comme toujours, nous sommes entrés dans un cas plus complexe où les choses se sont compliquées.

- Nous sommes devenus dépendants d'une structure HTML spécifique : la liste d'éléments.
- Les sprites ont du être redéfinis pour pouvoir être utilisés dans d'autres modules.
- Le positionnement parmi les éléments était très délicat : augmenter la taille de la police pouvait révéler d'autres parties du sprite.
- Gérer les interfaces se lisant de droite à gauche était plus difficile puisque nous ne pouvions qu'utiliser les sprites horizontaux et fixer la position x uniquement à 0.

Pour résoudre ces problèmes, nous avons cherché à ce que l'icône elle-même devienne un module : le module icône.

Restructuration du code HTML pour créer le module icône

```
<li><i class="ico ico-16 ico-inbox"></i> Inbox</li>
```

Beaucoup de gens grinceraient des dents en voyant l'utilisation du tag `i`. J'ai choisi de l'utiliser parce qu'il est petit, souvent dépourvu de sémantique et que c'est un élément vide, sans contenu. Pourquoi n'avons-nous pas mis de contenu ? Dans cet exemple, l'icône illustre le texte visible à sa droite dans le code. Si l'icône était seule, nous ajouterions un petit attribut afin qu'elle puisse être lue par un lecteur d'écran ou utilisée comme une info-bulle. Si vous n'êtes pas d'accord avec cette manière de faire et que vous préférez utiliser un `span`, je comprendrais. En utilisant un tag unique auquel nous avons appliqué les différentes classes d'icône, nous avons enlevé toutes les dépendances avec le HTML, ce qui est une bonne chose. Mais pourquoi trois classes différentes ? Chacune joue un rôle différent et, une fois rassemblées, nous avons une imitation de l'élément `img`.

CSS Module d'icône

```
.ico {
    display: inline-block;
    background: url(/img/sprite.png) no-repeat;
    line-height: 0;
    vertical-align: bottom;
}

.ico-16 {
    height: 16px;
    width: 16px;
}

.ico-inbox {
    background-position: 20px 20px;
}

.ico-drafts {
    background-position: 20px 40px;
}
```

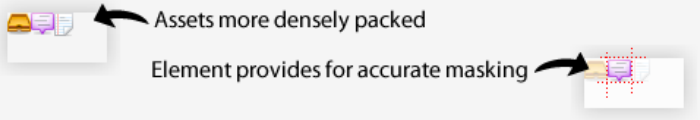
La classe `ico` définit les règles de base pour transformer un élément en `inline-block`, ce qui revient à reproduire une image. Vous pouvez ajuster le `vertical-align` afin que l'icône et le texte soient alignés. Internet Explorer a des difficultés à appliquer `inline-block` aux éléments blocs. Puisque nous l'appliquons à un élément `inline`, le problème n'est plus. Alternativement, vous pouvez utiliser `{zoom:1 ; display:inline;}` pour que les éléments blocs se comportent comme des éléments `inline-block` dans IE.

La classe `ico-16` définit la hauteur et la largeur. Si votre projet ne comporte qu'une seule taille d'icône, vous pouvez la définir dans la classe ico. Si chaque icône a une taille différente, vous pouvez la définir dans la classe de l'icône en question. Pour ce projet nous avons des ensembles d'icônes de quatre tailles différentes.

La dernière classe, `ico-inbox`, positionne le sprite sur les bonnes coordonnées. En ayant une taille d'icône fixe, nous n'avons plus le

souci de l'élément parent qui deviendrait trop grand, et nous pouvons utiliser le même code pour les interfaces en lecture de droite à gauche, sans avoir à modifier la position du fond.

Exemple d'une image sprite dense



Une image plus compacte permet également une meilleure compression. Les fichiers de petite taille améliorent la performance de votre site. (Et si ce n'est pas déjà le cas, je vous recommande les services de Smush.it[22] de Yahoo! ou de ImageOptim[23] pour Mac, pour vous assurer que vos images sont aussi petites que possible.)

Nous venons de voir un exemple de la manière dont nous pouvons reconsidérer une partie spécifique d'un projet pour rendre les choses plus souples. Il existe de nombreuses manières d'aborder un problème. Et parfois, lorsque nous avons l'impression que les choses fonctionnent en apparence, elles peuvent s'avérer problématiques avec l'avancée du projet.

Les projets évoluent au fur et à mesure que les complexités apparaissent. Choisir le meilleur moyen de résoudre ces problèmes fait partie intégrante des joies du métier de développeur web.

22.http://www.smushit.com/ysmush.it/
23.http://imageoptim.pornel.net/

HÉRITAGE COMPLEXE

Ce chapitre aborde le thème de l'héritage et la manière dont il peut parfois ruiner votre code le mieux élaboré.

Dans cet exemple, nous allons nous intéresser à un calendrier : il utilise les règles d'état génériques mais celles-ci sont en conflit avec l'héritage des cellules du tableau. Nous allons voir comment résoudre ce problème.

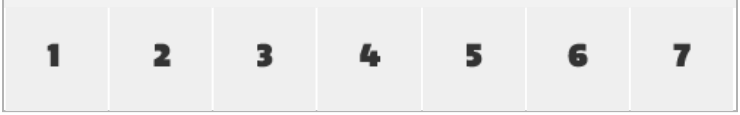
Calendrier en tableau

```
<table class="cal">
    <tr>
        <td>1</td> <td>2</td>
        <td>3</td> <td>4</td>
        <td>5</td> <td>6</td>
        <td>7</td>
    </tr>
    <!-- repeated 3-4 times -->
</table>
```

Le calendrier est composé d'un tableau avec des lignes et des colonnes. Chaque cellule représente un jour. Le style par défaut correspond à l'apparence d'un jour, dans les circonstances normales.

La cellule d'un jour

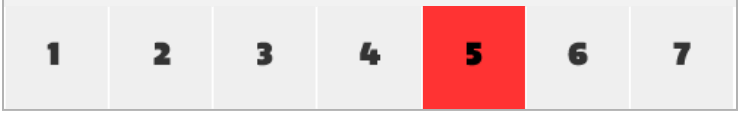
```
.cal td {
    background-color: #EFEFEF;
    color: #333;
}
```



Chaque cellule de mon tableau comporte un fond gris clair avec un texte gris foncé. Maintenant, nous voulons mettre en évidence la date d'aujourd'hui.

Mettre en évidence la date d'aujourd'hui

```
.cal td.cal-today {
    background-color: #F33;
    color: #000;
}
```



La classe `cal-today` montre que la classe `today` fait partie du module cal. La déclaration CSS pour la date d'aujourd'hui est plus spécifique et va ainsi écraser le style par défaut. Un autre sélecteur possible aurait pu être simplement `td.cal-today` qui aurait fonctionné tant qu'il était déclaré après la règle de l'état par défaut. Si nous avions juste utilisé `.cal-today` en tant que sélecteur, nous aurions dû utiliser `!important` pour que ce style fonctionne.

Il est important de réaliser qu'il y a de petites décisions comme celles-ci qui doivent être prises durant le développement du projet. Dans l'orientation que j'ai choisie, je considère que la classe `.cal-today` ne peut s'appliquer qu'à une cellule du tableau (`<td>`) et

qu'elle ne sera qu'à l'intérieur d'un élément portant la classe `cal` (ce qui, selon l'approche SMACSS, est considéré comme évident).

Revenons à notre exemple. Tout semble parfait. Maintenant notre calendrier est en vue réduite, lié à une vue plus grande présentant une semaine complète détaillée. Notre calendrier miniature doit montrer quelle est la semaine sélectionnée.

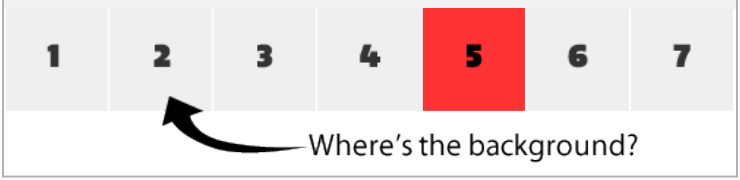
Ligne sélectionnée

```
<tr class="is-selected">
    <td>1</td>
    <td class="cal-today">2</td>
    <td>3</td>
    ...
</tr>
```

L'état sélectionné est utilisé dans toute l'application donc il était logique de l'insérer à cet endroit. A quoi ressemble le style sélectionné ?

Règle pour la ligne sélectionnée

```
.is-selected {
    background-color: #FFD700; /* Jaune */
    color: #000;
}
```



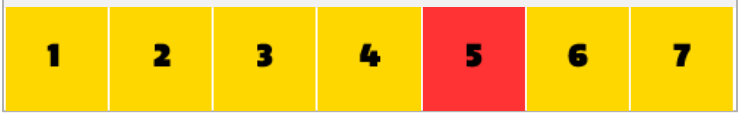
Voyez-vous le problème ? Le voici : alors que la couleur du fond s'applique pourtant à la ligne du tableau, la couleur qui s'affiche est celle définie dans la déclaration CSS ayant la spécificité la plus élevée, c'est-à-dire celle du style de base du jour et du style de la date d'aujourd'hui.

Je pourrais ajouter `!important` à mon état, ce que j'ai mentionné plus haut comme étant acceptable, mais alors qu'il accroît la spécificité lorsque nous appliquons le style au même élément, il n'est pas d'une grande utilité ici parce qu'il ne sera pas hérité par le reste de la cellule. `!important` ne prend pas le pas sur l'héritage, seulement sur la spécificité.

Cela signifie que je dois créer de nouvelles règles pour permettre à l'état sélectionné d'être transmis aux éléments enfants.

Règle de la ligne sélectionnée appliquée aux cellules du tableau

```
.is-selected td {
    background-color: #FFD700; /* Yellow */
    color: #000;
}
```



Si ce sélecteur est défini après le sélecteur de jour du calendrier, alors le rendu devrait être celui attendu.

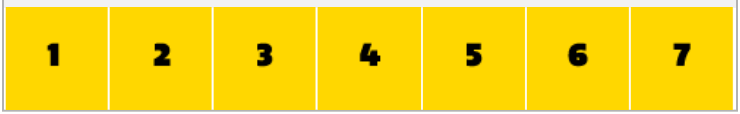
De quelle couleur vont maintenant être nos cellules ? Tout dépend. Ce style est-il défini avant ou après les classes cal ? Si c'est après, alors toutes les cellules de la ligne seront mises en page correctement. Notre cellule aujourd'hui sera toujours coloriée en rouge, ce qui est exactement ce que nous voulions sur ce projet.

Là où !important peut être inapproprié

Pour continuer le débat, que se passerait-il si nous ajoutions la règle `!important` à notre cellule CSS sélectionnée ? Tout à coup, notre cellule aujourd'hui ne montrerait plus aujourd'hui, mais ressemblerait au reste des jours de la semaine.

Ce que la règle `!important` ferait

```
.is-selected td {
    background-color: #FFD700 !important; /* Jaune */
    color: #000 !important;
}
```



Pour que la cellule aujourd'hui fonctionne correctement, nous devrions alors créer une nouvelle règle combinant la règle d'état et la règle du module.

Ajout de règles supplémentaires pour contourner la spécificité.

```
.is-selected td {
    background-color: #FFD700 !important; /* Jaune */
    color: #000 !important;
}

.is-selected td.cal-today {
    background-color: #F33 !important;
    color: #000 !important;
}
```

À partir de cet exemple, vous pouvez voir que nous allons devoir ajouter plus de sélecteurs et plus de `!important` pour que les choses fonctionnent correctement. Ce n'est vraiment pas l'idéal.

Un monde imparfait

L'objectif de cet exemple était de démontrer que l'héritage peut mettre du désordre dans l'organisation de notre code et qu'il n'existe pas de solution parfaite. L'approche SMACSS essaie d'atténuer ces complexités mais au final, vous devrez toujours créer des solutions imparfaites de temps en temps.

Minimiser le nombre de situations problématiques comme celle-ci, participera toujours à rendre votre projet plus gérable.

FORMATER LE CODE

Tout le monde a sa manière de faire. Les outils et les techniques que vous utilisez sont ceux que vous avez essayés soit pour résoudre un problème ou corriger une erreur, soit parce que quelqu'un vous en a vanté les mérites. Lorsque j'ai commencé à développer, j'utilisais Dreamweaver. J'avais de nombreuses fonctionnalités qui me permettaient de construire des sites HTML statiques rapidement et efficacement. Après avoir vu un collègue travailler sur Ultraedit et constaté sa vitesse de travail, j'ai commencé à l'apprendre, comme un moyen de compléter mon outil existant. La même chose s'est produite dans ma manière de coder. Je vois la technique ou le style que quelqu'un d'autre utilise, et je les intègre dans ma manière personnelle de travailler.

Ce chapitre, Formater le code, est un bref aperçu de ma manière de coder. Elle a été éprouvée et semble bien pratique pour ceux qui doivent continuer à travailler sur mon code après moi.

Une seule ligne vs plusieurs lignes

Pendant des années, j'ai codé mes CSS en utilisant l'approche de la ligne unique[24]. C'est-à-dire que toutes les propriétés d'une règle donnée étaient écrites sur la même ligne. Cette manière de faire permet de passer rapidement en revue les sélecteurs sur la gauche. Cela était plus important pour moi que de pouvoir visualiser facilement les propriétés. Il y a encore quelques années, la liste des propriétés affectées à une règle était encore relativement courte ; il n'y en avait en général pas plus que quelques-unes. Ainsi, je pouvais

24.http://orderedlist.com/resources/html-css/single-line-css/

repérer mon sélecteur, et toutes les propriétés étaient visibles à l'écran.

Avec l'arrivée de CSS3 – et sa myriade de préfixes constructeurs spécifiques – les choses se sont rapidement corsées. Entre cette réalité et celle du travail en grande équipe, il est devenu plus facile pour tout le monde de placer chaque paire propriété/valeur sur une seule ligne.

Le CSS3, avec sa myriade de préfixes constructeurs spécifiques, peut devenir trop complexe à lire si toutes les propriétés sont regroupées sur une seule ligne.

```
.module {
    display: block;
    height: 200px;
    width: 200px;
    float: left;
    position: relative;

    border: 1px solid #333;
    -moz-border-radius: 10px;
    -webkit-border-radius: 10px;
    border-radius: 10px;

    -moz-box-shadow: 10px 10px 5px #888;
    -webkit-box-shadow: 10px 10px 5px #888;
    box-shadow: 10px 10px 5px #888;

    font-size: 12px;
    text-transform: uppercase;
}
```

Dans cet exemple, il y a 11 propriétés déclarées et nous pourrions facilement en avoir une demi-douzaine de plus si nous ajoutions une ou deux autres fonctionnalités au module. Les avoir toutes sur une seule ligne rendrait les premières propriétés visibles à l'écran mais nécessiterait ensuite de scroller de gauche à droite pour pouvoir lire les autres. Dans cette configuration, il est difficile

d'analyser rapidement le document pour y trouver les propriétés définies.

Il y a encore quelques points à relever dans cet exemple :

- Il y a un espace après les deux-points.
- Il y a quatre espaces avant chaque déclaration (pas de tabulation).
- Les propriétés sont groupées par type.
- L'accolade ouvrante est sur la même ligne que le nom de la règle.
- Les déclarations de couleur utilisent la forme raccourcie.

Ces différents points sont des préférences personnelles et je ne vous en voudrai pas de ne pas les utiliser. Cette approche est simplement celle qui me semble naturelle et qui a le plus de sens pour moi.

Regroupement des propriétés

Certaines personnes organisent les propriétés de manière alphabétique, d'autres ne les organisent pas du tout, et d'autres encore utilisent des systèmes arbitraires. Je fais partie de cette dernière catégorie. Mais ce n'est pas complètement arbitraire, vous pensez bien. Je décrirais cela comme une organisation « du plus important au moins important », mais qu'est-ce qui est important lorsqu'il s'agit de déclarer les styles d'un élément ?

Voici l'ordre dans lequel je classe les propriétés de style d'un élément :

1. Le modèle de boîte
2. Les bordures
3. Le fond
4. Le texte
5. Le reste

Le modèle de boîte comporte toutes les propriétés qui affectent l'apparence et la position du conteneur comme `display`, `float`,

`position`, `left`, `top`, `height`, `width` etc. Ces propriétés sont les plus importantes pour moi car elles influent sur le reste du document.

Les bordures comportent l'attribut `border`, l'attribut peu fréquent `border-image` et `border-radius`.

Ensuite vient le fond. Avec l'arrivée des dégradés du CSS3, les déclarations de fond peuvent devenir assez prolixes. Encore une fois, les préfixes constructeurs ne font qu'empirer les choses.

Déclarations de fonds multiples avec CSS3.
Exemple provenant de Lea Verou's CSS3 Pattern Gallery[25]

```
background-color: #6d695c;
background-image: url("/img/argyle.png");
background-image:
  repeating-linear-gradient(-30deg,
rgba(255,255,255,.1), rgba(255,255,255,.1) 1px,
transparent 1px, transparent 60px),
  repeating-linear-gradient(30deg,
rgba(255,255,255,.1), rgba(255,255,255,.1) 1px,
transparent 1px, transparent 60px),
  linear-gradient(30deg, rgba(0,0,0,.1) 25%,
transparent 25%, transparent 75%, rgba(0,0,0,.1)
75%, rgba(0,0,0,.1)),
  linear-gradient(-30deg, rgba(0,0,0,.1) 25%,
transparent 25%, transparent 75%, rgba(0,0,0,.1)
75%, rgba(0,0,0,.1));
background-size: 70px 120px;
```

Les modèles complexes sont possibles avec les dégradés CSS3, mais créent des déclarations plus longues alors que l'exemple ne comporte même pas de préfixes CSS3. Imaginez donc la longueur de cette déclaration si elle en comportait !

Les déclarations de texte comportent les attributs `font-family`, `font-size`, `text-transform`, `letter-spacing` et toutes les autres propriétés CSS affectant l'apparence du texte.

25.http://leaverou.me/css3patterns/

Tout ce qui n'entre pas dans l'une de ces catégories est ajouté à la fin.

Les déclarations de couleur

Ce paragraphe peut sembler inutile à mentionner, mais j'ai vu différents développeurs s'y prendre de différentes manières. Certains aiment utiliser les mots-clefs comme `black` et `white` mais j'ai toujours essayé, autant que possible, de me conformer au format numérique comportant 3 à 6 chiffres après le dièse. `#000` et `#FFF` sont plus courts que leurs homologues mots-clefs. De même, je n'utilise pas `rgb` ou `hsl`, puisque la forme hexadécimale est plus courte. Bien entendu, `rgba` et `hsla` n'ont pas ne format numérique et sont donc utilisés.

Soyez cohérent

Au final, ce qui compte – comme le décrit en grande partie ce livre – est d'établir une convention, de la documenter et de vous y tenir. Au fur et à mesure que le projet évolue, cela rendra les choses plus simples pour vous et pour les autres.

RESSOURCES

Voici d'autres ressources et de nombreux outils excellents. Certains d'entre eux sont directement liés aux sujets discutés dans ce livre et d'autres sont simplement des outils utiles à avoir dans notre répertoire.

Les ressources SMACSS

Depuis la parution initiale de ce livre, des ressources autour l'approche SMACSS ont commencé à émerger :

- Middleman SMACSS[26]
- SMACSS for Drupal[27]
- SCSS Toolkit[28]. Un boîte à outils basée sur SMACSS pour démarrer.
- Kickstart SMACSS[29]

Les préprocesseurs CSS

- LESS[30]
- Sass[31]

26.https://github.com/nsteiner/middleman-smacss
27.http://drupal.org/sandbox/johnalbin/1704664
28.https://github.com/davidrapson/scss-toolkit
29.https://github.com/Anderson-Juhasc/kickstart-smacss
30.http://lesscss.org/
31.http://sass-lang.com/

Modèles/Méthodologies basés sur les composants

- Object-Oriented CSS (OOCSS)[32]
    - OOCSS for JavaScript Pirates Slides[33]
    - MailChimp UI Library based on OOCSS[34]
- BEM[35]

Autres Frameworks

- HTML5 Boilerplate[36]
- normalize.css[37]
- Bootstrap[38]
- 960.gs[39]
- Eric Meyer CSS Reset[40]

Documentation

- Front-end Style Guides[41]
- Knyle Style Sheets[42]

32.http://oocss.org/
33.http://speakerrate.com/talks/4642-oocss-for-javascript-pirates
34.http://www.flickr.com/photos/aarronwalter/5579386649/
35.http://bem.github.com/bem-method/html/all.en.html
36.http://html5boilerplate.com/
37.https://github.com/necolas/normalize.css/
38.http://twitter.github.com/bootstrap/
39.http://960.gs/
40.http://meyerweb.com/eric/tools/css/reset/
41.http://24ways.org/2011/front-end-style-guides
42.http://warpspire.com/posts/kss/

Autres ressources

- mustache[43] est un langage de template logic-less. C'est le format que nous avons choisi pour Yahoo!.
- Pattern Primer[44] est un script PHP qui vous permet de visualiser vos snippets HTML sur une seule page.
- Terrifically[45] est un modèle JavaScript/jQuery pour travailler avec OOCSS.

43.http://mustache.github.com/
44.https://github.com/adactio/Pattern-Primer
45.http://www.terrifically.org/