Orientation in Objects

) Modernes Webdesign mit CSS 3)

Modulares CSS

Version 1.1.0 (01.03.2018, Autor: Frank Bongers, Webdimensions.de)

Weinheimer Straße 68
68309 Mannheim
http://www.oio.de

Inhaltsverzeichnis

1. Modulares CSS

Es gibt verschiedene Ansätze, um die **Organisation** von CSS für größere Webprojekte oder eine Vielzahl von miteinander verwandten Projekten zu bewältigen. Die beiden bekanntesten sind **SMACSS** und **BEM**, die hier kurz vorgestellt werden sollen.

Die Organisation der CSS-Daten wird analog zur objektorientierten Programmierung vorgenommen. Man versucht die Paradigmen „Decoupling“, des „Single Responsibility Principle“, des „Separation of Concern“ und der „Encapsulation“ auf CSS zu übertragen. Als reine Präsentationssprache ist CSS von Haus aus nicht objektorientiert und fordert keines dieser Paradigmen ein. Hier (zugegeben etwas unscharf nachformuliert) nochmals in Kürze als Erklärung:

- **Decoupling**
  Konzepte sollen nicht untrennbar verbunden und somit nur gemeinsam verwendbar sein

- **Single Responsibility**
  Ein Konzept soll nur für einen Zuständigkeitsbereich und nicht für mehrere (unterschiedliche) eingesetzt werden

- **Separation of Concern**
  Ein Konzept soll nicht eine Vielzahl von Unterkonzepten bündeln. Ist dies der Fall, sollte darüber nachgedacht werden, diese zu extrahieren.

- **Encapsulation**
  Ein Konzept soll alle Aspekte beinhalten, die es definieren bzw. einsetzbar machen. Es kann so einfach als Gesamtheit in einen anderen Kontext gebracht werden.

1.1. Grundidee – modulares CSS

Unabhängig davon, *wie* die Ideen im Einzelnen formuliert werden, wird ein Dokument in logische **Layouteinheiten** (aka „Module“, „Blocks“) untergliedert, die einzeln (und unabhängig voneinander!) gestylt werden.

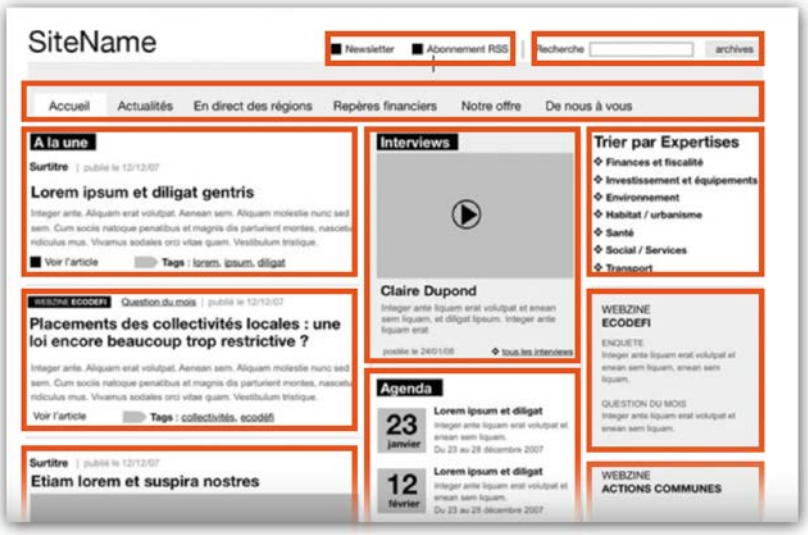
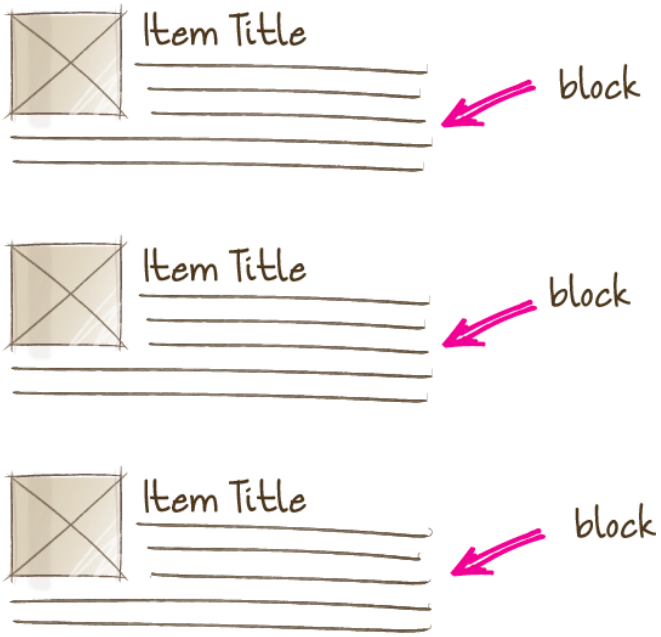


Abb.: Ein Dokument in Module untergliedert

Ein Grundprinzip der Modularität ist die Widerholbarkeit. Ist ein Modul entsprechend definiert, kann es bleibig oft in einem Dokument auftreten.



Module können hierbei verschachtelt sein. Eine Headerstruktur kann aus Untermodulen bestehen, die ihrerseits wiederum komplex aufgebaut sind.

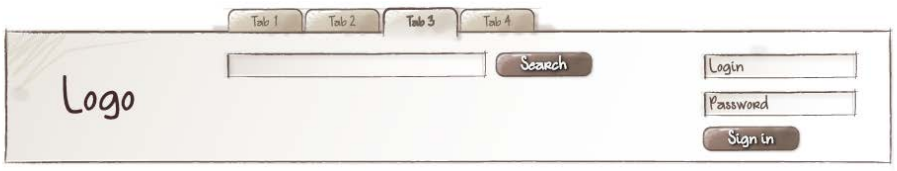


Abb: Mögliche Headerkonfiguration

Analysiert man den Aufbau der Struktur, so lässt sich diese in Untermodule zerlegen, die eigenständig definiert werden.

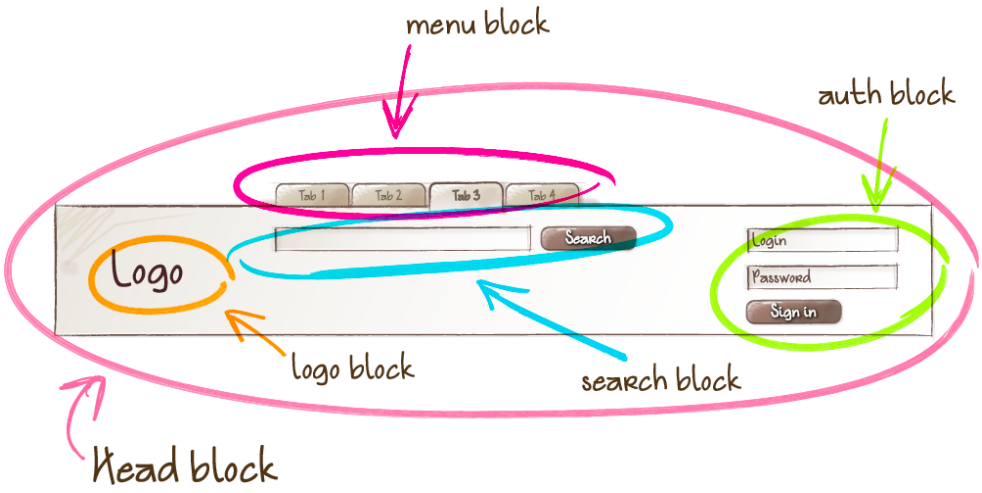


Abb.: Header, modular betrachtet

Ein Modul kann ebenfalls eine innere Struktur besitzen. Diese Struktur kann aus wiederholbaren Unterstrukturen bestehen, die jedoch nicht eigenständig genutzt werden. So wird die Definition eines Tabs nur innerhalb eines Tabmenüs verwendet werden. In diesem Fall wird man hier kein Modul, sondern ein „Element“, eine abhängige Substruktur definieren.

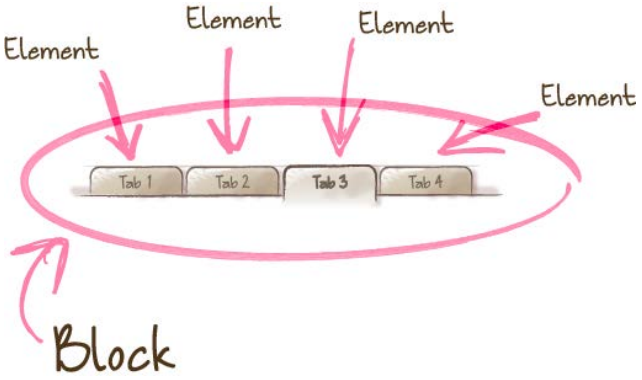


Abb.: Elemente als innere Substruktur eines Moduls (bzw. Blocks)

Module lassen sich, je nach aktuellem Bedarf, alternativ gruppieren und zu neuen Konfigurationen zusammensetzen. Sind die Module autark definiert, so können sie beliebig angeordnet und gegebenenfalls sogar verschachtelt werden.

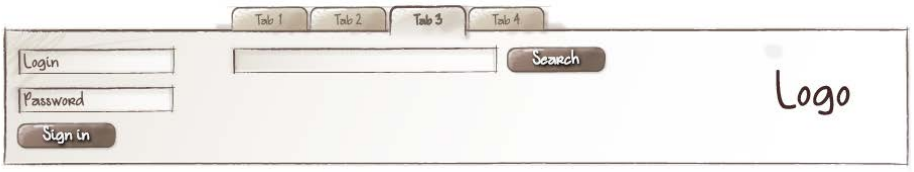


Abb.: Alternative Headerkonfiguration

Module können durchaus in verschiedenen Rollen mehrfach in einem Dokument erscheinen. So kann ein Tabmenü sowohl im Header als auch im Footerbereich einer Page auftreten. Das Menü im Footer könnte mit Hilfe einer Modifierklasse anders dargestellt werden.

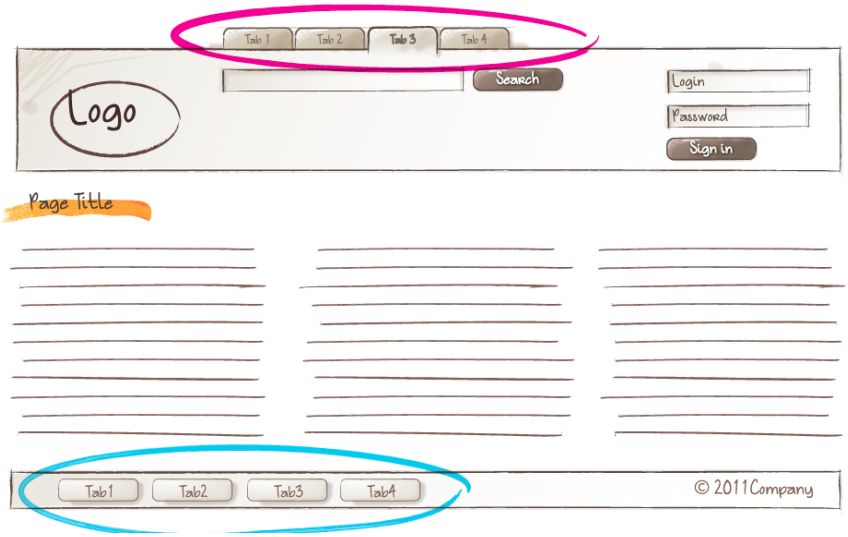


Abb: Zwei Tabmenümodule in verschiedenen Rollen

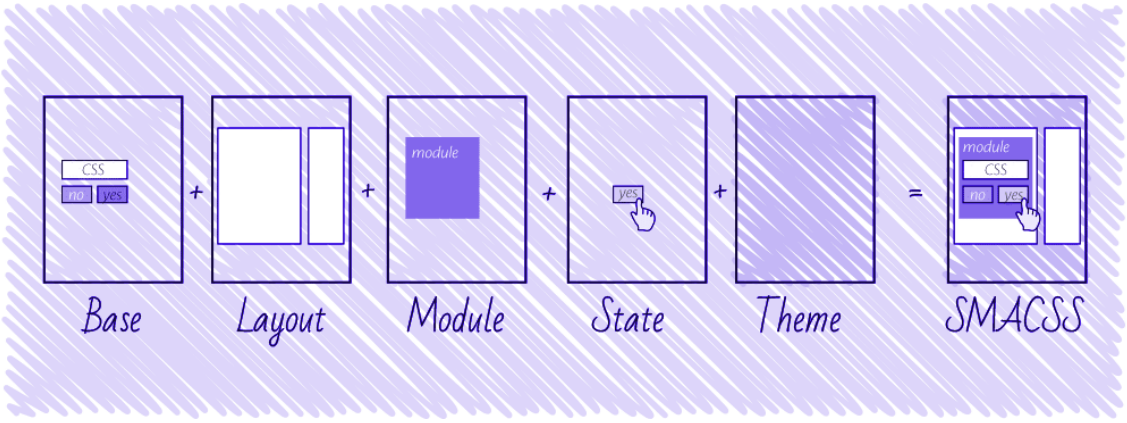
2. SMACSS

Die Idee zu **SMACSS** (**S**calable **M**odular **A**rchitecture for **CSS**) stammt von Jonathan Snook (http://snook.ca/about/). Die Ziele seines Ansatzes sind:

1. **Bessere Semantik** für HTML-Abschnitte (im Prinzip ist das Ergebnis eine Modularität durch semantische Benennung)

2. **Minimale Kopplung** zwischen CSS und der durch dieses geforderten HTML-Struktur (die Styles sollen so geschrieben

werden, dass das HTML möglichst frei gestaltet werden kann, ohne Erwartungshaltung oder Vorgaben seitens der Selektoren).



In SMACSS wird das CSS grob in fünf Abteilungen untergliedert:

✔ **Base**
enthält Styles, die sich auf das gesamte Dokument beziehen, also dessen Grundstyling betrifft. In diesem Rahmen können unter anderem *Resets* definiert werden. Allgemeine *typographische Einstellungen* (wie `font-family`, Basisschriftgrößen etc.) können hier platziert werden.

Die Base-Regeln wirken auf **HTML-Elementtypen**; sie enthalten daher keine Selektoren, die auf Klassen oder IDs aufbauen.

✔ **Layout**
enthält Styles für die *Großstrukturen* eines Dokuments, wie Header, Footer, Sidebar. Es wird hierbei bevorzugt der Layoutrahmen (Geometrie) beschrieben. Befürwortet wird der Einsatz von Klassen gegenüber IDs (um die Flexibilität zu wahren), jedoch ist der Einsatz von IDs auf dieser Ebene zu rechtfertigen.

Eventuell können hier auch modulunabhängige Utilityklassen (wie Bootstraps `.pull-right` oder vergleichbar) platziert werden. Es ist üblich, diese Klassen mit einem Präfix `.l-` zu versehen, der sie von Modulklassen unterscheidbar macht, wie `.l-pull-right` (statt nur `.pull-right`). Auch *Grid-Regeln* wären hier an der richtigen Stelle.

✔ **Module**
enthält Definitionen für *Strukturblöcke*, die beliebig im Inneren aller Layoutstrukturen einsetzbar und wiederholbar sein sollen. Module

werden grundsätzlich über Klassen realisiert, die das Modul *semantisch* benennen (z.B. `.card, .callout`).

Kindelemente innerhalb von Modulen erhalten eine Klasse hauptsächlich dann, wenn ein Elementtyp dort in verschiedenen Rollen auftreten kann. Ansonsten ist es üblich, der Rolle den Modulnamen voranzustellen wie folgt: `.card--label`.

Erklärung:
Der Modulname stellt den **„Namespace“** für die Bezeichner der Elementklassen für die Modulinhalte dar: `.modulname--element`.

Es ist Konvention in SMACSS, die Abtrennung mit *doppelten Bindestrichen* `--` vorzunehmen (BEM verwendet den doppelten Unterstrich in dieser Rolle). Der *einfache* Bindestrich bleibt so frei als Teil von Namen.

Da man eine **„flache“ Selektorhierarchie** bevorzugt, wird Namespacing anstelle einer *Verschachtelung* der Definitionen eingesetzt (wie beispielsweise `.modulname .element`), die zu Spezifizitätsproblemen führen könnte.

✔ **State**
dient dazu, Defaulteinstellungen für (beispielsweise) Elemente innerhalb von Modulen bei Bedarf zu überschreiben. Die *state*-Klassen werden in der Regel per JavaScript programmatisch hinzugefügt und wieder entfernt.

States werden durch das Präfix `.is-` gekennzeichnet, wie `.is-selected`, `.is-disabled` etc.

✔ **Theme** (optional)
kann bei Bedarf verschiedene visuelle Umsetzungen der Module bündeln, sodass beispielsweise zwischen unterschiedlichen Farbschemata gewechselt werden kann.

Theme-Klassen werden mit `.theme-` gepräfixt, beispielsweise also `.theme-red`, `.theme-green`, `.theme-big-margin` etc.

2.1. Base Rules

Die Base-Regeln werden üblicherweise über Elementselektoren zugewiesen. Hierbei können gelegentlich auch Child- oder Descendant-Selektoren oder (wenn nötig) Pseudoklassen verwendet werden.

Hingegen sollen im Rahmen der Selektoren für Base-Regeln **keine Klassenselektoren und keinesfalls ID-Selektoren** eingesetzt werden. Der Grund besteht darin, dass in diesem Fall ein Modulstyliing nicht in der Lage sein könnte, ein Base-Styling zu überschreiben.

Die Aufgabe der Base-Regeln besteht daher darin, für ein Element ein Grundstyling zu definieren, das unabhängig davon ist an welcher Position das Element im Dokument auftritt (in logischen Blöcken in Modulen oder anderweitig).

✔ Der Einsatz des `!important`-Flags in Base-Regeln sollte unterlassen werden.

Beispiel für Base-Rules

```
body, form {
   margin: 0;
   padding: 0;
}

a {
   color: #039;
}

a:hover {
   color: #03F;
}

::-moz-selection,
::selection {
    background: #38a2df;
    text-shadow: none;
    color: #fff;
}
```

2.2. Layout Rules

Die Layout-Regeln beziehen sich auf Großstrukturen des Dokuments wie Header, Footer und Contentbereiche und deren Unterregionen (beispielsweise Spalten oder Grids).

✔ Für die Großregionen kann der Einsatz von **ID-Selektoren** angemessen sein. Ein sorgfältige Planung der Benennung ist anzuraten.

Wiederkehrende kleinere Strukturen mit Layoutzweck sollten hingegen mit **Klassen** beschrieben werden. Die Abgrenzung zu Modulen kann hier etwas unscharf werden.

Beispiel für Layout-Deklarationen:

```
#header, #main, #footer {
   width: 960px;
   margin: auto;
}

#main {
   border: solid #CCC;
   border-width: 1px 0 0;
}
```

Generell sollte ein Selektor für eine Layout-Regel kein zusammengesetzter Selektor sein. Es kann jedoch erforderlich sein, dies zu durchbrechen, wenn eine Modifikation einer Layoutregel notwendig ist (die Modifikation sollte über die Spezifizität forciert werden).

Wir bezeichnen solche Layout-Regeln als „higher level“ Regeln.

Angenommen wir besitzen eine Schalter-Klasse um die Richtung des Layouts be Bedarf umzukehren. Die Defaultrichtung sei im Rahmen der gewöhnlichen Layoutregeln definiert:

```
#main {
   float: left;
}

#sidebar {
   float: right;
}
```

Die “higher level”-Regel nimmt die Schalterklasse hinzu, die außerhalb der Layoutstrukturen im HTML platziert werden muss, üblicherweise im BODY oder HTML-Tag:

```
.l-flipped #main {
   float: right;
}

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

Wird die Schalterklasse gesetzt, so tauschen Sidebar und Inhaltsbereich die Plätze.

✔ **Präfix für Layoutklassen:**
In diesem Beispiel ist angedeutet, dass Layoutstyles, deren Selektoren auf Klassen basieren mit `l-` gepräfixt werden. Dies dient dazu Layoutklassen auf den ersten Blick von Modulklassen unterscheiden zu können.

Bei ID-basierenden Layoutstyles ist ein Präfix nicht zwingend nötig, da IDs generell nur im Rahmen von Layoutdefinitionen eingesetzt werden. Wer dies als konsistenter empfindet, darf natürlich Selektoren wie `#l-header` einsetzen.

2.3. Module Rules

Module sollen per Definition *wiederholbare Einheiten* in einem Dokument sein. Deshalb ist es zwingend, als Selektor eine Klasse einzusetzen. Ein Modul kann eine innere Struktur besitzen, in einfachen Fällen, wie für Buttons oder Badges aber auch aus einem einzelnen Element bestehen. Entscheiden ist, das ein Modul eine fest definierte Rolle besitzt.

Beispiel für ein Modul zur Darstellung eines Elements als Button:

```
.btn {
  display: inline-block;
  padding: 6px 12px;
  font-size: 14px;
  text-align: center;
  cursor: pointer;
  border: 1px solid transparent;
```

```
  border-radius: 4px;
}
```

Ein Button:

```
<button class="btn"> ... </button>
```

Statt ein Modul über die Hinzunahme einer Klasse zu stylen wie beispielsweise so:

```
<button class="btn sucess"> ... </button>
```

... was folgende (problematische) CSS-Selektorstruktur erfordert (ein Button *muss* ein Button sein *und* die `success`-Klassen besitze):

```
/* Spezifizität: 0|2|0 */
.btn.success { ... }
```

... verwendet man eine Submodulklasse:

```
<button class="btn btn-sucess"> ... </button>
```

... die dann einfach separat definiert werden kann, ohne dass anschließend Spezifizitätsprobleme erscheinen:

```
/* Spezifizität: 0|1|0 */
.btn-success {
  color: #fff;
  background-color: #5cb85c;
  border-color: red;
}
```

... zumal in diesem Falle auch einfach Zustände dieser Submodulklasse gesetzt werden können:

```
.btn-success:hover {
  color: #fff;
  background-color: #449d44;
  border-color: #398439;
}
```

Wichtig bei diesem Ansatz ist es, die Basismodulklasse frei von Eigenschaften zu lassen, die durch Modifikatoren gesetzt werden sollen. Statt dem Basismodul ein Styling mitzugeben und dieses je nach Bedarf überschreiben zu müssen (was Probleme durch die Reihenfolge der

Definitionen ergeben könnte), lagert man auch das **Grundstyling** in einen Modifikator aus.

Dieser Button soll ein Success-Button sein:

```
<button class="btn btn-sucess"> ... </button>
```

Die Klasse `.btn-success` sei so definiert:

```
.btn-success {
  border-color: red;
}
```

Die Klasse `.btn` hingegen so:

```
.btn {
  ...
  border-color: gray;
}
```

Ist die Definitionsreihenfolge im Stylesheet falsch, so bleibt die Border grau!

Die Lösung besteht darin, konsequent auch die Grundeigenschaften in eine Modifierklasse auszulagern und bei `.btn` zu entfernen:

```
.btn-default {
  border-color: gray;
}
```

Ein normaler Button sieht dann so aus:

```
<button class="btn btn-default"> ... </button>
```

Ein Modul kann jedoch auch eine **innere Struktur** besitzen, wie beispielsweise ein Content-Pane oder Panel. Es wird aus einem äußeren Container bestehen, dem die Rolle zugeschrieben wird und einen (möglicherweise schwer zu reglementierenden) Inhalt.

Wir würden also ein Modul `.panel` erzeugen.

```
/* Spezifizität: 0|1|0 */
.panel { ... }
```

Wir wären versucht, es auf beispielsweise folgende Struktur anzuwenden:

```
<div class="panel">
    <h3>Titel des Panels</h3>
    <p>Inhalte des Panels ...</p>
</div>
```

... und würden nun vielleicht als Children von `.panel` Element-Selektoren schreiben, um die Panelinhalte zu stylen:

```
/* Spezifizität: 0|1|1 */
.panel > h3 { ... }

.panel > p { ... }
```

Leider ist es vielleicht nicht sicher, ob die Überschrift in Form einer H3 *gewährleistet* werden kann, oder dass nicht mehrere H3-Überschriften im Panel sein könnten. Dasselbe gilt für die P-Tags.

Ist die HTML-Struktur klar, und die *Rolle* der betreffenden Elemente im Modul eindeutig, so ist dieser Ansatz vollkommen okay!

Man erstellt im Allgemeinen besser Rollenzuschreibungen in Form von Klassen, auch wiederum als Submodulklassen:

```
.panel-heading { ... }
.panel-content { ... }
```

Dies ergäbe (neben der Angleichung der Spezifizität) folgende etwas saubere Lösung:

```
<div class="panel">
    <h3 class="panel-heading">Titel des Panels</h3>
    <p class="panel-content">Inhalte des Panels ...</p>
</div>
```

Der Ansatz hat den Nachteil, dass er im Grunde davon ausgeht, dass der Content nur aus P-Elementen besteht (und wenn nicht, dass das betreffende Element ebenso gestylt werden könnte). Zudem zwingt er uns, *jedem einzelnen* Element die Klasse zuzuweisen.

Besser, man definiert eine Containerklasse:

```
.panel-body { ... }
```

Was uns zu folgender Struktur bringt:

```
<div class="panel">
  <h3 class="panel-title">Titel des Panels</h3>
  <div class="panel-body">
    <p>Inhalte des Panels ...</p>
     ...
  </div>
</div>
```

Aus Symmetriegründen kann man gleiches für einen Header- und einen (optionalen) Footer-Bereich vorsehen:

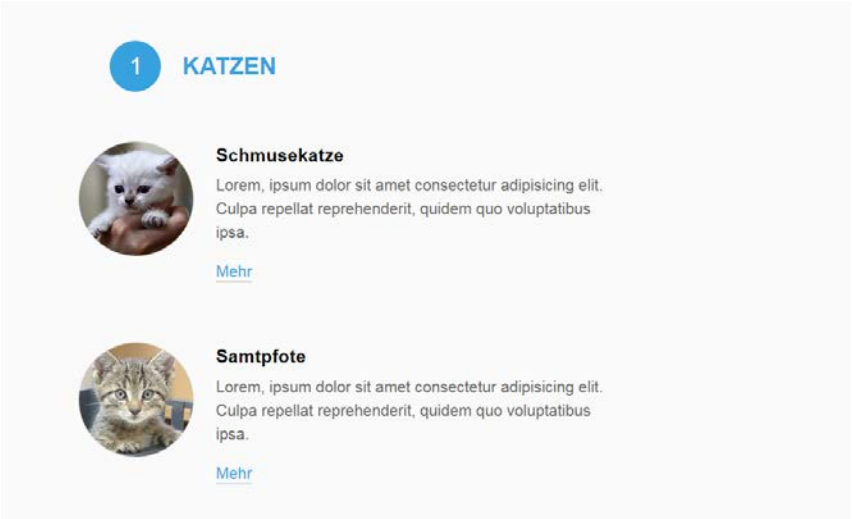
```
.panel-header { ... }
.panel-footer { ... }
```

Womit die HTML-Struktur des Moduls auch wie folgt aussehen könnte (aber nicht muss!):

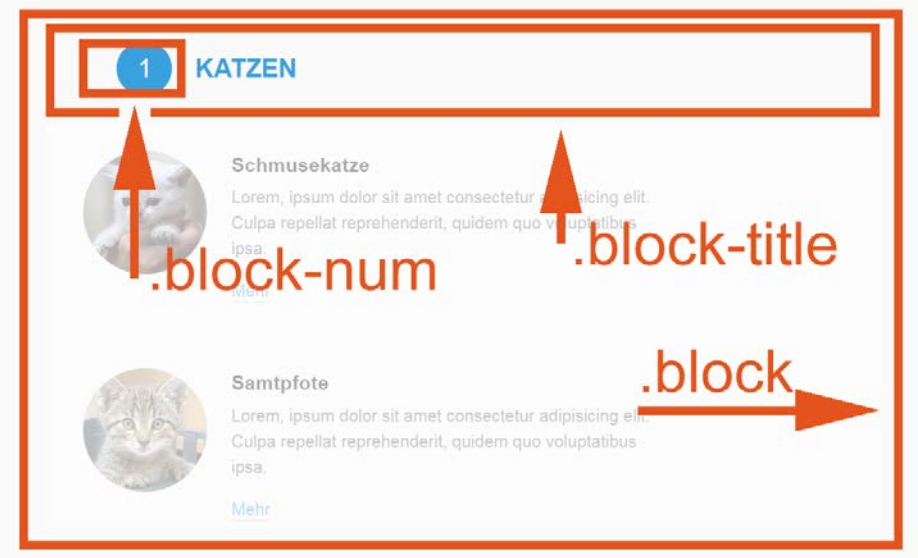
```
<div class="panel">
  <div class="panel-header">
    <h3 class="panel-title">Titel des Panels</h3>
  </div>
  <div class="panel-body">
    <p>Inhalte des Panels ...</p>
  </div>
</div>
```

✔ Die im Inneren der Modulbereiche auftretenden Element können nun (wenn angebracht) wiederum mit Childselektoren gestylt werden

Hier ein Beispiel für Moduldefinitionen, mit denen eine Reihe von Posts (als Module) in Blöcken (ebenfalls Module) gelayoutet werden können.



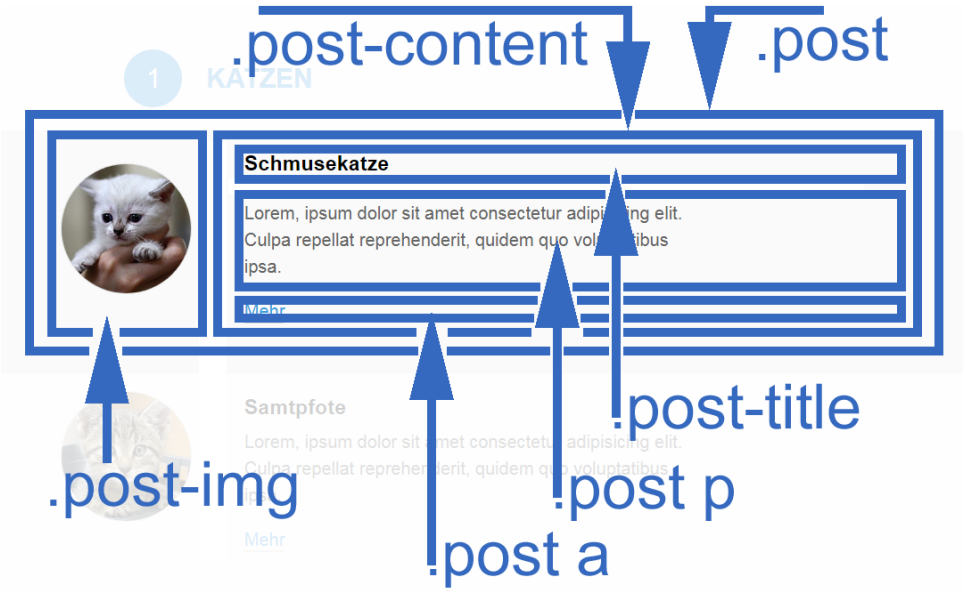

```
.block {
  padding: 66px 0 66px 127px;
  border-bottom: 1px solid #eee;
  background-color: rgba(245, 245, 245, .5);
}

.block:nth-child(even) {
  background-color: rgba(255, 255, 255, .5);
}

.block-title {
  font-size: 24px;
  font-weight: 700;
  text-transform: uppercase;
  color: #38a2df;
}

.block-num {
  font-weight: 400;
  background-color: #38a2df;
  color: #fff;
  padding: 0 19px 0 20px;
  display: inline-block;
  text-align: center;
  line-height: 52px;
  margin: 0 15px 0 32px;
  border-radius: 30px;
}
```



Hier die Regeln für die Posts innerhalb eines Blocks:

```
.post {
    padding: 30px 15px 30px 0;
    max-width: 550px;
}

.post-img {
    border-radius: 116px;
    float: left;
    width: 116px;
    height: 116px;
    border: 1px solid #e1e1e1;
}

.post-content {
    margin-left: 140px;
}

.post-title {
    font-weight: 600;
    margin-top: 0;
    margin-bottom: 5px;
}

.post p {
    font-weight: 300;
    color: #666;
    margin-top: 0;
}
```

```
.post a {
    color: #38a2df;
    text-decoration: none;
    border-bottom: 2px solid #ddd;
}

.post a:hover {
    border-color: #38a2df;
}
```



2.4. State Rules

Ein State-Rule ist ein Modifier, der im Zweifelsfall andere Regeln überschreiben wird.

Es kann sich hierbei um einen allgemeinen Modifier handeln, wie einen, der die Sichtbarkeit oder Nichtsichtbarkeit eines Blocks steuert, oder einen modulbezogenen Modifier, der Zustände im Rahmen eines Moduls beschriebt.

Die SMACSS-Nomenklatur schlägt vor, solche Regeln mit einem `-is-` Präfix zu versehen:

```
.is-hidden {
  display: none;
}
```

Wird eine solche Klasse jedoch eingesetzt und trifft auf eine Konfiguration mit höherer Spezifizität (oder eine mit gleicher Spezifizität, die später notiert ist, was wahrscheinlicher ist), so bleibt sie unwirksam.

Aus diesem Grund ist für solche Klassen der Gebrauch des `!important`-Flags nicht nur erwägenswert, sondern angebracht:

```
.is-hidden {
  display: none !important;
}
```

2.5. Theme Rules

Es gibt mehrere Möglichkeiten, die Regeln für eine Website zu „themen“, also abhängig von Darstellungswünschen zu modifizieren.

Die Themes können in des Stylekontext über die Reihenfolge eingebracht werden (es bietet sich an, derartige Regeln einfach als letzte zu inkludieren:

```
.module {
  border: 1px solid black;    <~ default theme rule
  ...
}

/* Theme Rules */
.module {
  border: 5px dotted blue;    <~ theme rule override
}
```

In diesem Fall müssen im HTML keine Maßnahmen getroffen werden, indem beispielsweise eine Themeklasse gesetzt wird. Das Theme wird ins Setup inkludiert oder nicht (oder ein anderes Theme wird inkludiert). Das Theming erfolgt also als Teil des Buildprozesses.

Ein anderer Ansatz besteht darin, die Themes, wo benötigt, mit Hilfe von Klassen zu aktivieren.

```
.module {
  border: 1px solid black;    <~ default theme rule
  ...
}

/* Theme Rules */
```

```
.theme-borders {
  border: 5px dotted blue;    <~ theme rule over-ride
}
```

Dies erfordert die Vergabe einer entsprechenden Klasse im HTML:

```
<div class="module theme-border"> … </div>
```

2.6. CSS-Frameworks mit SMACSS-Ansatz

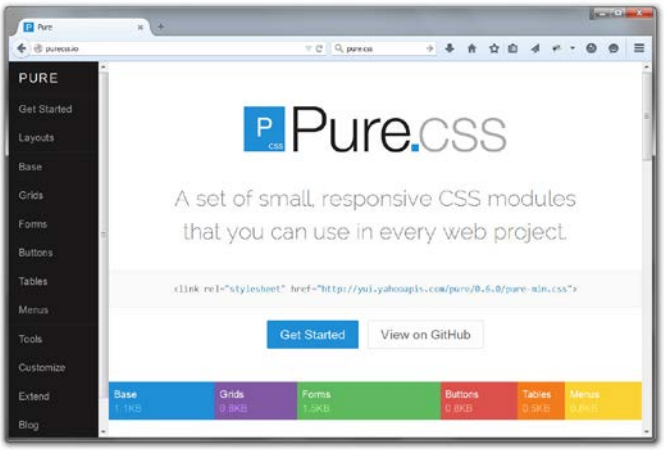
Zwar ist SMACSS “lediglich” ein konzeptioneller Ansatz, doch gibt es ein paar (und nicht gerade unwichtige) Vertreter unter den CSS-Frameworks, die dem SMACSS-Ansatz folgen.

✔ Es biete sich an, eines dieser Framewoks als Grundlage einzusetzen wenn die eigenen benötigten Module gemäß von SMACSS-Regeln gestylt werden sollen.

2.6.1. Pure CSS

Pure CSS ist ein schlankes, responsives und komponentenorientiertes Framework, das auch Gridfunktionalität enthält. Es ist vergleichbar mit Bootstrap, wenn auch weniger mächtig und punktet über seine geringe Dateigröße (3,8 kB in gezippter Fassung).

✔ Als Reset wird NormalizeCSS eingebunden.



✔ Quelle: `purecss.io`

Pure CSS sagt von sich offiziell, dass es „den Konventionen von SMACCS folgend“ entworfen wurde. Eine **Erweiterbarkeit** durch eigene Regeln wird hierdurch erleichtert, wie die Pure-Dokumentation ausdrücklich darlegt:

✔ Siehe: `https://purecss.io/extend/`

Kooperation zwischen Pure und Bootstrap:

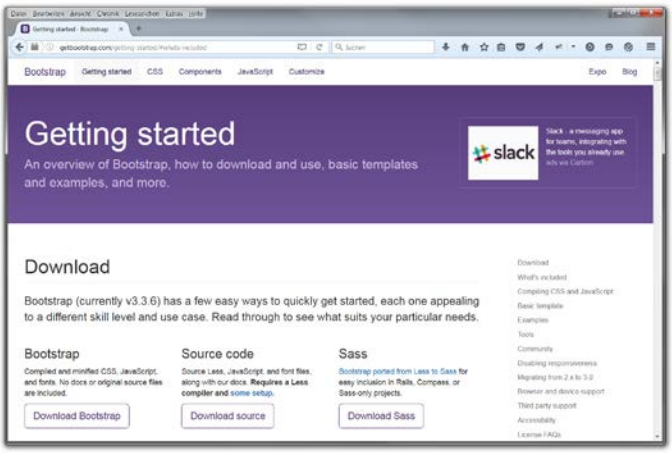
Hochinteressant ist die Tatsache, dass Pure als *Basis* für den Einsatz von **Bootstrap-Komponenten** dienen kann, was die CSS-Datenmenge deutlich verringert. Hierfür wird nicht das gesamte Bootstrap-Framework geladen, sondern lediglich das CSS der gewünschten Bootstrap-Komponente (+ jQuery).

Siehe wiederum: `https://purecss.io/extend/`

2.6.2. Bootstrap

Bootstrap ist der Platzhirsch unter den CSS-Frameworks und zeichnet sich durch seine Vielseitigkeit (Responsive Layouts, Grids, Komponenten, Widgets etc.) und häufige Verwendung aus. Der Nachteil besteht in der recht umfangreichen Kollektion von Regeln, die sich in einer entsprechend großen Datei widerspiegelt.

✔ Da Bootstrap auf **LESS** (in Version 4 auf *SASS*) aufbaut, lässt sich jedoch ein optimierter Build erstellen, der nicht benötigte Komponenten ausschließt.



✔ Quelle: `https://getbootstrap.com`

Bootstrap ist nicht offiziell gemäß SMACSS-Kriterien erstelt, jedoch bescheinigt selbst Jonathan Snooks dessen Konformität. Im Prinzip gilt daher das vorhin bereits für *Pure CSS* gesagte.

3. BEM CSS

Bei **BEM** ("**B**lock", "**E**lement", "**M**odifier") handelt es sich um einen *methodischen Ansatz* zur Definition effizienter CSS-Regeln für größere und häufigen Veränderungen unterworfenen Websites. Das System wurde vom russischen Internetunternehmen *Yandex* (www.yandex.ru) entwickelt. Der Sitz des Unternehmens befindet sich in Amsterdam. Die von *Yandex* angebotenen Dienste (z.B. eine Suchmaschine) entsprechen in etwa denen von Google.

3.1. Ziele von BEM

Yandex schlägt zur Verwendung als Selektoren für BEM-Strukturen die ausschließliche Verwendung von **CSS-Klassen** vor. Die Namen der Klassen sollen hierbei semantisch gewählt werden. Die Bennenung der Klassen für Substrukturen folgen einer Semantik (dazu später).

Ziele sind:

- **Modularität** und **Wiederverwendbarkeit**
  Die Beschränkung auf Klassen und hiermit die geringe Variabilität der Spezifizität der Selektoren erhöht die Interoperabilität. Styledefinitionen für Strukturen sind ohne weiteres in andere Projekte eingliederbar.

- **Nomenklatur** zur Erleichterung von **Teamarbeit**
  Strukturen können (in gewissem Maße) unabhängig vom Kontext gestaltet werden, in dem sie später eingesetzt werden. Die Benennung der Klassen impliziert ihren Einsatz und ihre Zugehörigkeit. Versehentliche Duplizierung oder Konflikte sind (weitestgehend) ausgeschlossen.

Der **Nachteil** (wenn man so will) des Ansatzes besteht in einem erhöhten Aufwand bei der Vergabe von Klassen innerhalb der HTML-Strukturen, sowie die (der Nomenklatur geschuldeten) Länge der einzusetzenden Klassenbezeichner.

3.2. Block, Element, Modifier

Beim BEM-Ansatz gliedert man die zu stylenden Teile des Dokuments grob in

- **Blocks**
  Eine zu stylende Einheit beliebiger Komplexität. Block sind im Allgemeinen *autark* (stehen für sich) und *wiederholbar*, können also in verschiedener Zahl und an verschiedenen Positionen im Dokument erscheinen (dabei auch verschachtelt sein). Beispiele sind Header- oder Footer-Konfigurationen, aber auch Navigationen oder einzeln eingesetzte Buttons oder Formularfelder. Ein Block wird durch eine Klasse benannt, z.B. `menu` oder `container`.

Blöcke eines BEM-Designs sind stets **unabhängig** voneinander.

- **Elemente**
  Ein HTML-Container innerhalb eines Blocks, der dort eine bestimmte Rolle spielt, die aber durch seine Zugehörigkeit zum Block (als Teil dessen Semantik) festgelegt ist und nicht über seinen Typ. Ein und dasselbe Element kann also in unterschiedlichen Blöcken verschiedene Rollen übernehmen. Ein Element wird ebenfalls über eine Klasse gekennzeichnet, die den umgebenden Block im Namen nennt, z.B. `menu__item`, `container__caption`.

Um für einen Block bzw. ein Element innerhalb eines Blocks verschiedene Zustände definieren zu können, kommt ein drittes Konzept hinzu:

- **Modifier**
  Ein Modifier ist eine Klasse, die den Zustand eines Blocks oder eines Elements innerhalb eines Blocks widerspiegelt. Üblicherweise wird ein Modifier per JavaScript zugewiesen (Pseudoklassen sind ebenfalls denkbar). Ein Beispiel ist der `disabled`-Zustand eines Buttons oder der *highlighted*-Zustand eines Textabsatzes. Ein Modifier kann frei einsetzbar definiert werden oder sich (über den Namen symbolisiert) auf einen Block oder ein Element eines Blocks beziehen.
  Beispiele: `--state-success` oder `button--state-success`.

Beispiel für den Einsatz einer Blockdefinition für Buttons:

```
<button class="button">
  Normal button
</button>
<button class="button button--state-success">
  Success button
</button>
<button class="button button--state-danger">
  Danger button
</button>
```

Hier wird pro Button die Block-Klasse `.button` vergeben. Der zweite und dritte Button erhält zusätzlich je eine Modifier-Klasse: `.button--state-XXX`.

Das CSS könnte beispielsweise wie folgt aussehen:

```
/* Der Block: */

.button {
  display: inline-block;
  border-radius: 3px;
  padding: 7px 12px;
  border: 1px solid #D5D5D5;
  background-image: linear-gradient(#EEE, #DDD);
  font: 700 13px/18px Helvetica, arial;
}

/* Modifier für den Block: */

.button--state-success {
  color: #FFF;
  background: #569E3D linear-gradient(#79D858, #569E3D)
repeat-x;
  border-color: #4A993E;
}

.button--state-danger {
  color: #900;
}
```

Zwei Dinge sind zu bemerken: Die Modifier-Klassen sind über den Namen als zum Button-Block gehörig gekennzeichnet. Die Nomenklatur folgt der

Syntax **`block--modifier-value`**. Die Buttonklasse ist als freie Klasse *unabhängig* von einem festen Element definiert. Sie könnte daher (ähnlich Bootstrap) auf Input-Buttons `<input type="submit">`, normale Buttons `<button>` oder Links `<a>` angewendet werden.

3.3. BEM-Nomenklatur

BEM setzt ausschließlich CSS-Klassen ein. Vorgeschlagen wird eine Nomenklatur für deren Benennung wie folgt – der Einsatz **doppelter Unterstriche** und **doppelter Bindestriche** als Trenner bzw. Verbinder ermöglicht den Einsatz von Unterstrich und Bindestrich als Teil der Namen.

```
.blockname

.blockname__element

.blockname__element--modifier

.blockname--modifier
```

Beispiel für den Einsatz von BEM-Klassen in einer HTML-Struktur:

```
<article class="card">
   <header class="card__image-wrapper">
      <img src="placeholder.gif">
   </header>
   <section class="card__information-wrapper">
      <div class="card__information">
         <p>Lorem ipsum... </p>
      </div>
      <button class="button button--success">
         Mehr
      </button>
   </section>
</article><!-- Ende card-Block -->
```

Hier wird ein **Block** “card” definiert, der drei **Elemente** beinhaltet, nämlich den Image-Wrapper, den Information-Wrapper und den eigentlichen Inhalt, als Information bezeichnet.

Die Card enthält einen Button-Block als autark definierte Substruktur. Der Button besitzt eine Modifier-Klasse.

✔ **Anmerkung:**
Das Element `card__information` tritt hier zwar als *Child* des Elements `card__information-wrapper` auf, wird aber nicht als dessen Child definiert (oder evtl. mit `card__information-information` benannt). Es geht primär darum, ein Element einem Block zuzuordnen und nicht gleichzeitig dessen hierarchische Position im Block festzulegen. Der Wrapper könnte somit auch entfallen.

Die CSS-Definitionen könnten entsprechend wie folgt angelegt werden:

```
/* Blocks */
.card {
}

.button {
}

/* Elemente */
.card__image-wrapper {
}

.card__information-wrapper {
}

.card__information {
}

/* Modifier */
.button--sucess {
}

.button--danger {
}
```

Wichtig: Die Selektoren werden nicht verschachtelt definiert, also nicht beispielsweise als

```
.card .card__image-wrapper { ... }
```

Hierdurch wird möglichen Problemen mit der Spezifizität vorgebeugt. Auch würde eine Definition wie

```
.card .card__information-wrapper .card__informantion { ... }
```

die Anwesenheit des Wrappers im Code *voraussetzen* und die Flexibilität in der Anwendung einschränken.

3.4. Beispiele für BEM

Wir betrachen einige (relativ einfache) Beispiele für den BEM-Ansatz.

3.4.1. Einfache BEM-Komponente

Auch hier stellt ein Button ein klassisches Beispiel dar. Die **Komponente** ist stets eine einfache CSS-Klasse.

```
.btn { ... }
```

```
<button class="btn">
   BEM-Button
</button>
```

3.4.2. BEM-Komponente mit Modifier

Erscheint eine Komponente in verschiedenen Auswirkungen, so wird hierfür eine **Modifier-Klasse** erstellt. Diese Klasse wird zusätzlich zur Komponentenklasse vergeben, ist also *nicht* autark. Sie kann die Regeln der Komponente ergänzen oder überschreiben (modifizieren).

Anstelle der Konfiguration Komponente + Modifier zwei unabhängige Komponenten zu erstellen ist ungünstig und widerspricht der BEM-Idee.

```
.btn {
    display: inline-block;
    color: blue;
}

.btn--secondary {
    color: green;
}
```

```
<button class="btn btn--secondary">
   BEM-Button mit Modifier
</button>
```

✔ Es ist essentiell, dass die Modifier-Klasse im CSS-Quelltext *nach* der Komponentenklasse steht. Dies muss gewährleistet sein, da beide Selektoren dieselbe Spezifizität besitzen (was wiederum ein Überschreiben erlaubt, also gewünscht ist).

3.4.3. BEM-Komponente mit Elementen

Ist eine Komponente komplexer, so wird sie über eine Containerklasse mit Kindelementen (BEM-**Elements**) beschrieben. Ein Element, das eine *Rolle* besitzt, die ein Styling erfordert, erhält eine *CSS-Klasse*.

Es entspricht nicht den BEM-Regeln, an dieser Stelle auf Typselektoren zurückzugreifen. Dies koppelt die Komponente zu eng an eine HTML-Konfiguration und widerspricht der Idee der semantischen Auszeichnung. Zudem ändert sich hierbei die Spezifizität.

```
/* Komponentenklasse - Spezifizität 0 1 0*/
.photo { ... }

/* Elementklassen - Spezifizität 0 1 0*/
.photo__img { ... }
.photo__caption { ... }
```

Dies wird wie folgt auf die HTML-Struktur der Komponente angewandt:

```
<figure class="photo">
  <img class="photo__img" src="bild.jpg">
  <figcaption class="photo__caption">
     Sonnenblumen
  </figcaption>
</figure>
```

Es ist ein Missverständnis, auf die Klassen bei den Elementen zu verzichten, auch wenn hierdurch der berüchtigten “Klassitis” vorgebeugt wird. Zwar sieht dies hier einfacher aus:

```
<figure class="photo">
  <img src="bild.jpg">
  <figcaption>
     Sonnenblumen
  </figcaption>
</figure>
```

...und die Selektoren ließen sich auch leicht schreiben:

```
/* Komponentenklasse - Spezifizität 0 1 0*/
.photo { ... }

/* Elementklassen - Spezifizität 0 1 1 !! */
.photo > img { ... }
.photo > figcaption { ... }
```

Jedoch erhalten die Elementselektoren nun unversehens eine **höhere Spezifizität**. Dies erschwert das kontrollierte Überschreiben und widerspricht der BEM-Idee.

Oben wurden Kindselektoren eingesetzt. Ist die Schachteltiefe einer Komponente größer, so soll sich dies jedoch nicht in den Elementeklassen widerspiegeln.

3.4.4. Tiefe BEM-Komponente mit Elementen

Das Prinzip von BEM besteht darin, die Selektoren von der Hierarchie abzukoppeln. Auch Elemente in tiefer Schachtelung werden durch *einfache Klassen* dargestellt.

> Hinweis: Tritt *dieselbe* Rolle mit *verschiedenem* Styling in unterschiedlicher Tiefe auf, so sollte eine zweite Klasse eingeführt werden.

```
/* Komponentenklasse - Spezifizität 0 1 0*/
.photo { ... }

/* Elementklassen - Spezifizität 0 1 0*/
.photo__img { ... }
.photo__caption { ... }
/* tiefes Element: */
.photo__quote { ... }
```

Dies wird wie folgt auf die HTML-Struktur der Komponente angewandt:

```
<figure class="photo">
  <img class="photo__img" src="me.jpg">
  <figcaption class="photo__caption">
    Sonnenblumen
    <q class="photo__quote">
      Vincent van Gogh
    </q>
  </figcaption>
</figure>
```

Hingegen sollte *nicht* das tiefe Element in seiner Klasse das übergeordnete BEM-Element berücksichtigen, weder in seinem Namen, noch im Selektor.

```
/* Komponentenklasse - Spezifizität 0 1 0*/
.photo { ... }

/* Elementklassen - Spezifizität 0 1 0*/
.photo__img { ... }
.photo__caption { ... }

/* tiefes Element - schlechte Lösung (a) */
.photo__caption__quote { ... }

/* tiefes Element - schlechte Lösung (b) */
.photo__caption .photo__quote { ... }
```

Der zweite Ansatz fällt aufgrund seiner Spezifizität durch. Der erste schränkt durch die Benennung die HTML-Hierarchie ein.

3.4.5. Modifikationen an BEM-Elementen

Gelegentlich ist es angebracht, anstelle einer gesamten Komponte lediglich eines der Elemente zu modifizieren. In diesem Fall besteht ein möglicher Ansatz darin, Modifierklassen anzubieten, die dies übernehmen.

Soll beispielsweise in der Komponente ein Bild mit einem *Rahmen* versehen werden oder eine Bildunterschrift *größer* dargestellt werden, so *kann* dies über Modifier geschehen.

```
/* Komponentenklasse - Spezifizität 0 1 0*/
.photo { ... }
```

```
/* Elementklassen - Spezifizität 0 1 0*/
.photo__img { ... }
.photo__caption { ... }
/* tiefes Element: */
.photo__quote { ... }

/* Element-Modifier: */
.photo__img--framed {
    /* zusätzliche Styles für Element */
}
.photo__caption--large {
    /* zusätzliche Styles für Element */
}
```

Hier wirft sich die Frage auf, inwieweit das Auftreten oder die Kombination solcher Modifier reglementieren lässt. Am HTML zwingt uns dieser Ansatz dazu, weitere Klassen am Element hinzuzunehmen. Modifikatoren an mehreren Elementen sind nicht gekoppelt (können also vergessen oder falsch gesetzt werden).

```
<figure class="photo">
  <img class="photo__img
    photo__img--framed" src="bild.jpg">
  <figcaption class="photo__caption
    photo__caption--large">
     Sonnenblumen
  </figcaption>
</figure>
```

Der Ansatz ist durchaus legitim, wenn die Unahängigkeit zwischen den Modifikationen gewollt ist.

Besteht jedoch ein Zusammenhang zwischen den Modifikationen innerhalb einer Komponente, so ist für diesen Zweck tatsächlich ein Modifier an der Komponente selbst vorzuziehen.

```
/* Komponentenklasse - Spezifizität 0 1 0*/
.photo { ... }
/* Modifier */
.photo--highlighted { ... }

/* Elementklassen - Spezifizität 0 1 0*/
.photo__img { ... }
.photo__caption { ... }
/* tiefes Element: */
```

```
.photo__quote { ... }

/* Element-Modifier jetzt hierarchisch: */
.photo--highlighted .photo__img {
    /* zusätzliche Styles für Element */
}
.photo--highlighted .photo__caption {
    /* zusätzliche Styles für Element */
}
```

Das HTML ist jetzt wieder einfacher. Die Modifikationen an den Elementen wird jetzt über die Stylesheethierarchie erzwungen, muss also nicht an den Elementen selbst mit Hilfe von Klassen festgelegt werden.

```
<figure class="photo photo--highlighted">
  <img class="photo__img" src="bild.jpg">
  <figcaption class="photo__caption">
     Sonnenblumen
  </figcaption>
</figure>
```

✔ **Zwei Dinge sind anzumerken:**
Ist keine Modifikation am Komponentencontainer selbst erforderlich, so kann die autarke Definition des Komponenten-Modifiers entfallen. Die Spezifizität des Selektors für die Elementmodifikation ist höher. Sollen *mehrere* solcher Modifikationen gleichzeitig über die Komponente getriggert werden, so kann dies Probleme mit der Überschreibung von Propertywerten zur Folge haben.

Es ist also in Ordnung, wenn ein Modifier die Wirkung eines *BEM-Elementselektors* (oder eines Blocks) überschreibt. Andersherum ist es jedoch **nicht ratsam**, einen *Modifier* durch den Kontext eines Block- oder Elementselektors zu verändern. Es ist dann nicht mehr im Einzelfalle vorhersehbar, was der Modifier tun wird.

4. Modulares CSS und CSS-Präprozessoren

Der Einsatz einer **CSS-Präprozessorsprache** wie LESS oder SASS erleichtert die Arbeit mit modular organisiertem CSS (wie auch die Verwaltung größerer Stylesheetprojekte generell - siehe den Einsatz von LESS bei Bootstrap).

Beide Sprachen haben gemeinsam, dass es sich bei ihne um Supersets von reinem CSS handelt - das bedeutet, dass jedes CSS automatisch auch gültiges LESS bzw. SASS ist.

✔ Wir werfen gleich einen kurzen Seitenblick auf LESS.

Außer diesen beiden Ansätzen ist auch **Stylus** ein verbreiteter Präprozessor. Es existieren weitere, weniger bedeutende Ansätze wie *Myth*, *Clay* (in Haskell geschrieben) oder *Rework*

✔ **LESS:** http://lesscss.org/

✔ **SASS:** https://sass-lang.com/

Die, für manchen Entwickler, hässlich anmutende und unbequeme Schreibweise der BEM-Klassen mit doppelten Unter- oder Bindestrichen sowie die gemeinsame Verwaltung aller Klassen eines Moduls lassen sich in SASS und LESS (bei gleicher Syntax!) sehr einfach realisieren.

Das folgende Beispiel in LESS (dieselbe Syntax funktioniert ebenfalls in SASS) beruht auf der Möglichkeit der Verschachtelung von Definitionen und den Parent-Selektor `&`, der Bezug auf den an der Wurzel des Ausdrucks liegenden Selektor (hier die Modulklasse) nimmt. Somit werden alle nach dem &-Symbol stehenden Strings automatisch als separate Klassen definiert, die mit dem Modulnamen gepräfixt werden (aus `&--modifier` wird `.mein-block--modifier`):

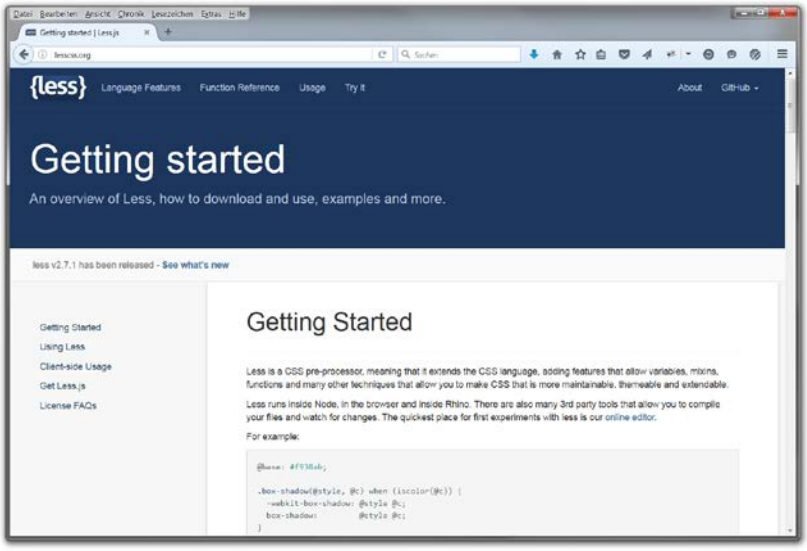
```
.mein-block {
   background-color: #aaa;

   &__element {
      border: 1x solid black;

      &--modifier-one {
         color: red;
      }

      &--modifier--two {
         color: blue;
      }
   }
}
```

Dies erzeugt folgendes CSS:

```
.mein-block {
   background-color: #aaa;
}
.mein-block__element {
   border: 1x solid black;
}
.mein-block__element--modifier-one {
   color: red;
}
.mein-block__element--modifier-two {
   color: blue;
}
```

4.1. LESS

LESS wird entwickelt von *Alexis Sellier*, besser bekannt als *cloudhead*. (`www.cloudhead.io`). LESS stellt eine *Erweiterung* (Superset) für CSS dar und ist deswegen nicht nur abwärtskompatibel, sondern nutzt CSS Syntax für zusätzliche Features.



✔ **Siehe:** http://lesscss.org

4.1.1. Was ist LESS?

LESS ist eine **CSS-Precompilersyntax** und erweitert die Möglichkeiten von CSS mit dynamischem Verhalten in Form von Variablen, Mixins, Berechnungen und Funktionen.

Üblicherweise wird LESS nach reinem CSS *kompiliert*, kann aber (unter Performanceverlust) auch mit Hilfe eines Interpreters sowohl client- als auch serverseitig *direkt* verwendet werden.

✔ Die aktuelle Version ist **LESS v2.7.2**

4.1.2. LESS kompilieren

Da eine LESS-Datei von der Syntax von einer CSS-Datei abweicht, kann sie nicht direkt genutzt werden, sondern muss kompliert, also in „echtes“ CSS umgewandelt werden. Dies ist die Aufgabe des LESS-Compilers, der als Node-Modul installiert werden muss:

```
npm install -g less
```

Hierfür muss bereits eine globale Installation von Node vorliegen. Erstens, weil sonst der NPM zum Laden der Komponente nicht verfügbar ist, zweitens, weil Node auch die Laufzeit für den Compiler darstellt.

Überprüfen Sie die korrekte Installation, indem Sie einfach die Version des Compilers abfragen:

```
lessc -version
> lessc 2.7.2 (Less Compiler) [JavaScript]
```

Ein LESS-Stylesheet kann bei globaler Installation des Compilers von jedem Verzeichnis aus an der Kommandozeile kompiliert werden:

```
lessc styles.less styles.css
```

Dies kompiliert eine Datei *styles.less* und erzeugt dabei im gleichen Verzeichnis eine Datei *styles.css*.

4.1.3. Variablen in LESS

Mit Variablen können mehrfach im Stylesheet gebrauchte Werte an einer zentralen Stelle definiert und anschließend überall referenziert werden. Änderungen sind dadurch schneller und fehlerfrei möglich.

```
/* ----------------- LESS ----------------- */
@color: #4D926F;
#header {
  color: @color;
}
h2 {
  color: @color;
}

/* ------------ Generiertes CSS ------------ */
```

```
#header {
  color: #4D926F;
}
h2 {
  color: #4D926F;
}
```

Im Zuge der Definition von Variablen können auch **Berechnungen** angestellt und andere Variablen einbezogen werden:

```
/* ----------------- LESS ----------------- */

@stahlblau: #5B83AD;
@hellblau: (@hellblau + #111);

#header {
   color: @hellblau;
}

/* ------------ Generiertes CSS ------------ */

#header {
   color: #6c94be;
}
```

Der Scope von Variablen funktioniert in LESS analog zu dem in Programmiersprachen üblichen Verfahren: Zuerst werden Variablen und Mixins *lokal* (im aktuellen **Block**) gesucht. Werden sie nicht gefunden, geht die Suche im *nächst höheren* Geltungsbereich weiter.

```
@var: red;

#page {
  @var: white;
  #header {
    color: @var; // white
  }
}

#footer {
  color: @var; // red
}
```

4.1.4. Kommentare in LESS

Die üblichen CSS Kommentare gelten in LESS ebenfalls.

```
/* Ich bin ein CSS Kommentar. Und ein LESS Kommentar */
.class {    color: black   }
```

✔ Ein solcher Kommentar wird ins erzeugte CSS ausgegeben, sofern nicht eine minifizierterte Datei erzeugt werden soll.

In LESS können auch **einzeilige Kommentare mit Doppelslash** eingesetzt werden, wie sie aus Javascript bekannt ist.

```
// Kommentar nur im LESS-Code sichtbar
.class {    color: white    }
```

✔ **Praktisch:** Die einzeiligen Kommentare werden **nicht** ins kompilierte CSS übernommen. Es ist daher möglich (und anzuraten) die LESS-Module großzügig zu kommentieren und mit erläuterungen zu versehen.

4.1.5. Verschachtelung von Regeln

Mit LESS kann man neben dem üblichen „cascading“ auch **„nesting“**, also das Verschachteln von Regeln, einsetzen. Anstatt für das Styling ineinander verschachtelter Strukturen entsprechend lange Selektorketten zu schreiben, werden LESS-Selektoren einfach ineinander verschachtelt.

```
/* ----------------- LESS ----------------- */

#header {
  h1 {
    font-size: 26px;
    font-weight: bold;
  }

  p { font-size: 12px;
    a { text-decoration: none;
      &:hover { border-width: 1px }
    }
```

```
  }
}
```

Wichtig ist hier der &-Kombinator. Dieser wird als **Platzhalter für das Elternelement** verwendet, was besonders für die Definition von Pseudoklassen, wie `:hover` und `:focus`, praktisch ist.

```
/* -------------- Generiertes CSS -------------- */

#header h1 {
   font-size: 26px;
   font-weight: bold;
}

#header p {
   font-size: 12px;
}

#header p a {
   text-decoration: none;
}

#header p a:hover {
   border-width: 1px;
}
```

✔ Zu beachten ist, dass das &-Symbol zwar hierarchisch im Inneren der Parentdefinition eingesetzt wird (also sichtbar diesem zugeordnet ist), die Ergebnisregel jedoch parallel auf gleicher Ebene mit der Parent-Regel erscheint. Es werden in diesem Falle also keine zusammengesetzten Selektoren generiert!

4.1.6. Media Queries mit LESS

LESS kann sehr gut mit Media-Queries umgehen und erleichtert das erstellen von CSS, das solche Blöcke einbezieht, enorm.

Media Queries können hierfür genauso wie Selektoren verschachtelt werden. Hier wird die Klasse `.one` für einen Breakpoint definiert und eine Zusatzquery für diesen Breakpoint definiert:

```
/* ----------------- LESS ----------------- */

.one {
    @media (width: 400px) {
        font-size: 1.2em;

        @media print and color {
            color: blue;
        }
    }
}
```

Im generierten Stylesheet wird dies nach `@media`-Blöcken getrennt ausgegeben:

```
/* -------------- Generiertes CSS -------------- */

@media (width: 400px) {
   .one {
     font-size: 1.2em;
   }
}

@media (width: 400px) and print and color {
   .one {
     color: blue;
   }
}
```

4.1.7. Import von Dateien in LESS

Analog zu CSS existiert auch in LESS ein `@import`-Statement, mit dem sowohl *.less*-Dateien oder *.css*-Dateien importiert werden können.

✔ Das Verhalten des Prozessors unterscheidet sich jedoch bei beiden Dateitypen voneinander.

Durch Importe werden beispielsweise Variablen und Mixins verfügbar gemacht. Die Nennung der Dateiendung `.less` ist optional:

```
@import "lib.less";
```

Dies entspricht in der Wirkung

```
@import "lib";
```

LESS unterscheidet dabei zwischen LESS- und CSS-Dateien. Eine CSS muss zwar den LESS-Prozessor nicht durchlaufen, kann aber dennoch importiert werden:

```
@import "beispiel.css";
```

Achtung:
Dies führt jedoch nicht, wie vielleicht erwartet wird, dazu, dass LESS den CSS-Code in das Ergebnisdokument *einfügt*, sondern lediglich, dass dieses *Importstatement* ins Ergebnisdokument kopiert wird.

Da das kopierte Importstatement dort, wie es sich gehört, am **Anfang** des Dokuments auftaucht, werden die Regeln an der diesem entprechenden Position (wohl also zu früh!) registriert und können keine der anderen Regeln überschreiben.

Ist die Position des Imports **relevant**, kann man den LESS-Prozessor veranlassen, die CSS-Daten „inline“, d.h. exakt an der Stelle ins Ergebnisdokument einzufügen, die der Position des CSS-Imports entspricht. Hierfür muss beim Import das `inline`-Statement angegeben werden:

```
@import (inline) "beispiel.css";
```

Hierdurch wird der CSS Code aus *beispiel.css* in die Ziel-Datei eingefügt und zwar an der Position, der dem Importstatement entspricht.

Zusammengefasst stellt sich die Lage wie folgt an:

- ✔ **Import von LESS-Dateien:**
  Eine importierte LESS-Datei durchläuft den LESS-Prozessor und das Ergebnis wird anstelle des Importstatements in der Ergebnisdatei eingefügt. Dies ermöglicht es, eine beliebige Anzal von LESS-Modulen in kontrollierter Reihenfolge (wichtig!) in ein Ergebnisdokument zu „plätten“.

- ✔ **Import von CSS-Dateinen mit inline-Statement:**
  Eine mit inline-Statement importierte CSS-Datei verhält sich wie eine LESS-Datein. Der Prozessor wird sie zwar nicht verarbeiten, plättet die

Datei jedoch in das Ergebnisdokument und platziert ihre Statements exakt dort wo gewünscht.

✔ **Import von CSS-Datei ohne inline-Statement:**
Die Datei bleibt separat erhalten und wird nicht in das Ergebnisdokument integriert. Vielmehr wird dort ein Importstatement eingefügt, der diese Datei im Kontext des Ergebnisdokuments zur Verfügung stellt. Die Statements des CSS-Imports liegen aber in der CSS-Kaskade vor den Statements des Ergebnisdokuments und können dieses zwar *ergänzen*, aber nicht dessen Regeln *überschreiben*.

Folgende Datei würde zu einer Ergebnisdatei *dist/styles.css* kompiliert, die einfach aus allen Statements der Module in Reihenfolge der Importe besteht:

```
/* src/styles.less */
@import "utils/variablen.less";
@import "utils/mixins.less";
@import "modules/button.less";
@import "modules/card.less";
@import "modules/accordion.less";
// HIER wird nur der CSS-Import durchgereicht:
// @import "base/text.css";
// So werden CSS-Regeln inkludiert:
@import (inline) "base/text.css";
```

Vor den eigentlichen Modulen werden die Hilfsdateien eingebunden, die Variablen und andere Definitionen (z.B. Mixinklassen) bereitstellen, die von den Modulen verwendet werden. Die Inhalte der Variablen- und Mixindateien werden jedoch nicht im Ergebnis erscheinen.

Eine denkbare Ordnerstruktur für das Projekt wäre wie folgt:

```
src/
   utils/
      variablen.less
      mixins.less
   base/
      basestyles.less
      text.css          <- besser wäre .less-Datei :-)
   layout/
      header.less
      footer.less
      grid.less
   modules/
      accordion.less
      button.less
      card.less
   themes/
      themes_kontrast.less   <- optional
   styles.less     <- diese wird kompiliert

dist/
   styles.css      <- die Ergebnisdatei
```

Die Ergebnisdatei sollte beim Buildvorgang selbstverständlich auch noch *komprimiert* werden.

5. Links

SMACSS
https://smacss.com/

BEM
https://en.bem.info/

6. Literatur

Scalable and Modular Architecture for CSS
Jonathan Snook
(Eigenverlag, über Website)