ANTEPRIMA GRATUITA

CSS SEMANTICO

Scrivere CSS
che comunica,
scala e dura
nel tempo.



ROBERTO BECCARI
robertobeccari.it

Cosa trovi nel libro completo

PARTE I — IL NUOVO CSS

1. Perché non ti serve più un preprocessore
2. **Il nesting nativo e il CSS semantico**
   LO STAI PER LEGGERE
3. Custom properties: le variabili che vivono nel browser

PARTE II — LOGICA E CALCOLO NELLO STANDARD

4. Le funzioni CSS: la matematica che il browser fa per voi
5. Colori di nuova generazione
6. Selettori potenti: `:has()`, `:is()`, `:where()` e `:not()`
7. Architettura senza tool: `@layer`, `@scope` e i moduli nativi

PARTE III — LAYOUT E COMPONENTI MODERNI

8. Container queries: componenti che rispondono al proprio spazio
9. Layout avanzato: subgrid, `aspect-ratio` e le griglie senza framework
10. Interazioni e animazioni native

PARTE IV — PROGETTI GUIDATI

11. Progetto 1: un design system minimale
12. Progetto 2: il refactoring di un progetto Sass reale
13. Progetto 3: una landing page senza media query

APPENDICI

A. Compatibilità browser e fallback progressivo · B. Cheat sheet: da Sass a CSS standard · C. Glossario

> **COME È FATTO OGNI CAPITOLO**
>
> Riepilogo in cinque punti ed esercizio risolto in chiusura. I box "Per approfondire" sono per chi ha già esperienza; quelli "Nota per chi arriva da Sass" mappano le vecchie abitudini sulla nuova pratica.

Capitolo 2 — Il nesting nativo e il CSS semantico

> **DOVE SIAMO**
>
> Secondo capitolo della Parte I. Nel Capitolo 1 abbiamo visto che lo standard ha assorbito quasi tutto ciò che Sass offriva; qui affrontiamo la feature più amata, il nesting, e scopriamo che la sua versione nativa non è solo comodità sintattica: cambia il modo stesso di progettare le classi. Questo è il capitolo-manifesto del libro.

Il nesting è il motivo per cui molti di noi hanno installato Sass la prima volta. Poter scrivere la struttura di un componente *dentro* il componente, invece di ripetere il selettore a ogni riga, è una di quelle comodità a cui non si torna indietro. Dal 2023 il nesting è nello standard, Baseline in tutti i browser principali.

Ma questo capitolo non si ferma alla sintassi. Nella prima metà impariamo a usare il nesting nativo — incluse le differenze da Sass che possono sorprendervi. Nella seconda metà tiriamo le conseguenze: se gli stati e le varianti di un componente possono vivere *dentro* la sua regola, non serve più codificarli *nel nome* della classe. È la fine dell'era `class="button button--primary button--disabled"` e l'inizio di `class="button primary disable"`. Vedremo perché questo è un progresso per l'HTML, per il JavaScript e per l'accessibilità — e, onestamente, dove l'approccio ha dei limiti.

2.1 La sintassi del nesting nativo

Il principio è semplice: una regola può contenerne altre. Il selettore annidato si combina con quello del genitore come farebbe un selettore discendente.

```
/* Una card con titolo e link, senza ripetere .card */
.card {
  padding: 1.5rem;
  border-radius: 8px;
```

```
  .title {
    font-size: 1.25rem;
    margin-block-end: 0.5rem;

    a {
      color: inherit;
      text-decoration: none;
    }
  }

  img {
    width: 100%;
    aspect-ratio: 16 / 9;
    object-fit: cover;
  }
}
```

Questo equivale a scrivere `.card`, `.card .title`, `.card .title a`, `.card img` come regole separate. La struttura del componente è finalmente leggibile nel codice, non solo intuibile dai nomi.

Si possono annidare anche le **at-rule**, ed è una delle comodità maggiori: la media query vive accanto agli stili che modifica, non in fondo al file.

```
.sidebar {
  display: none;

  @media (width >= 64rem) {
    display: block;
    inline-size: 18rem;
  }
}
```

(Notate di passaggio la sintassi a intervalli delle media query, `width >= 64rem`, anch'essa Baseline dal 2023: più leggibile di `min-width`.)

Supporto: il nesting nativo è Baseline 2023. La prima implementazione di Chrome richiedeva `&` davanti ai selettori di elemento (`& img`); la sintassi "rilassata" che vedete qui sopra, dove `img` funziona da solo, è arrivata in tutti i browser entro la fine

del 2023. Se dovete supportare browser del 2022, il fallback è semplicemente... scrivere i selettori per esteso, o lasciare quel compito a un post-processore durante la transizione.

2.2 Il selettore `&`

`&` rappresenta il selettore del genitore. Serve nei casi in cui la combinazione non è quella discendente:

```
.button {
  background: var(--color-primary);

  /* Pseudo-classi: .button:hover, non .button :hover */
  &:hover {
    background: var(--color-primary-dark);
  }

  /* Classi composte sullo stesso elemento: .button.compact */
  &.compact {
    padding: 0.25rem 0.75rem;
  }

  /* Inversione di contesto: .theme-dark .button */
  .theme-dark & {
    background: var(--color-primary-light);
  }
}
```

I tre usi da memorizzare:

1. **`&:pseudo`**, senza `&`, `:hover` da solo dentro una regola verrebbe comunque interpretato come `.button :hover` (discendente in hover), che non è quasi mai ciò che volete. Con le pseudo-classi, `&` è di fatto obbligatorio.
2. **`&.classe`**, l'elemento che ha *entrambe* le classi. È il costrutto su cui si regge tutta la seconda metà del capitolo.
3. **`contesto &`** — il genitore usato *a destra*: utile per temi e contesti (`.theme-dark &`), da usare con parsimonia perché inverte la lettura.

2.3 Dove il nesting nativo diverge da Sass

Se arrivate da Sass, la sintassi vi sembrerà familiare al 95%. È il restante 5% a causare i bug più subdoli, quindi affrontiamolo subito.

Niente concatenazione di nomi

In Sass, `&` è una stringa: potete concatenarci sopra un suffisso.

```
/* Sass: funziona */
.button {
  &--primary { ... }   /* genera .button--primary */
  &__icon { ... }      /* genera .button__icon */
}
```

Nel nesting nativo questo non esiste. `&` non è testo da concatenare: è un selettore vivo che rappresenta gli elementi selezionati dal genitore. `&--primary` non è una sintassi valida e non genererà mai `.button--primary`.

Fermiamoci un attimo, perché questa non è una lacuna: è una presa di posizione del linguaggio. La concatenazione di Sass esisteva quasi esclusivamente per servire convenzioni come BEM, cioè per *costruire nomi di classi composte*. Lo standard ha scelto di non supportarla, e nella seconda metà del capitolo vedremo che questa scelta ci spinge esattamente nella direzione giusta: se non potete generare `.button--primary` per concatenazione, la strada naturale è `&.primary` — una classe componibile scopata dentro il componente.

> **NOTA PER CHI ARRIVA DA SASS**
>
> Questo è il punto in cui i refactoring automatici da SCSS a CSS si rompono. Se la vostra codebase usa massicciamente `&__` e `&--`, non potete limitarvi a rinominare i file in `.css`: quei costrutti vanno ripensati, non tradotti. Il lato positivo: il Capitolo 12 mostra la migrazione passo-passo, e il risultato finale è un CSS più semplice di quello di partenza, non un adattamento di ripiego. Nel frattempo, la regola mentale è: in Sass `&` è una stringa, nello standard `&` è un selettore.

La specificità segue le regole di `:is()`

Il nesting nativo si comporta, ai fini della specificità, come se il selettore del genitore fosse avvolto in `:is()`. E `:is()` ha una proprietà particolare: la sua specificità è quella del selettore *più specifico* nella lista, per tutti.

```
.alert, #banner {
  .title {
    color: red;
  }
}
```

In Sass questo genererebbe `.alert .title, #banner .title`: due selettori con specificità diverse. Nello standard equivale a `:is(.alert, #banner) .title`: **un** selettore la cui parte iniziale vale quanto un id, anche quando matcha `.alert .title`. Se altrove avete scritto `.alert .title { color: blue }`, in Sass vincerebbe l'ordine di apparizione; nello standard vince sempre la regola annidata, perché `:is(.alert, #banner)` pesa (1,0,0).

Nella pratica quotidiana la differenza emerge di rado (se non mescolate id e classi nelle liste di selettori, e non dovreste) ma quando emerge è sconcertante se non la conoscete.

> **PER APPROFONDIRE**
>
> Due dettagli per i senior. **Primo:** il modello a `:is()` significa anche che i selettori del genitore non vengono "riespansi" per ogni combinazione, con benefici sul peso del CSS generato... che non esiste: non c'è CSS generato, ed è questo il punto — il browser valuta il nesting direttamente, e i DevTools moderni mostrano le regole annidate così come le avete scritte, con la catena di genitori esplicitata. **Secondo:** l'ordine tra dichiarazioni e regole annidate. Nelle prime implementazioni, le dichiarazioni scritte *dopo* una regola annidata venivano risalite in cima alla regola, alterando l'ordine effettivo; le implementazioni attuali (tramite le `CSSNestedDeclarations`) preservano l'ordine di scrittura. Resta una buona pratica di leggibilità, che adottiamo in tutto il libro, mettere sempre tutte le dichiarazioni prima delle regole annidate.

2.4 Gli errori comuni

Replicare il DOM nel CSS. Il nesting invita a rispecchiare la struttura HTML livello per livello. Resistete: ogni livello di annidamento è accoppiamento tra CSS e markup. Se spostate il titolo fuori dal wrapper, `.card > .header > .title` smette di funzionare; `.card .title` no; una classe diretta ancora meno.

```
/* Fragile: accoppiato alla struttura esatta del markup */
.card {
  > .header {
    > .title {
      > a { ... }
    }
  }
}

/* Robusto: seleziona per ruolo, non per posizione */
.card {
  .title a { ... }
}
```

Dimenticare il punto in `&.classe`. `& .primary` (con spazio) e `&.primary` (senza) sono selettori diversi: discendente il primo,

composto il secondo. È l'errore di battitura più costoso del capitolo.

Annidare per abitudine ciò che è globale. Gli stili tipografici di base, i reset, le utility non appartengono a nessun componente: annidarli dentro il primo componente che li usa li rende invisibili agli altri.

2.5 La regola dei 2-3 livelli

Fissiamo la regola che useremo in tutti i progetti del libro: **massimo due, eccezionalmente tre livelli di annidamento.** Il primo livello è il componente; il secondo sono i suoi elementi, stati e varianti; il terzo, quando serve, è una pseudo-classe o una media query su un elemento interno.

Non è una regola estetica. Ogni livello aggiunge specificità e accoppiamento: un selettore profondo è più difficile da sovrascrivere, da spostare e da capire fuori contesto. Con Sass la community aveva imparato questa lezione a proprie spese (la "inception rule" dei tempi d'oro); il nesting nativo non cambia la fisica del problema.

Se vi trovate al quarto livello, quasi sempre lì dentro c'è un componente che chiede di nascere: estraetelo, dategli una classe, ripartite da uno.

2.6 Il problema che BEM risolveva davvero

Ora tiriamo le conseguenze, e per farlo dobbiamo prima rendere giustizia a BEM. Le convenzioni come BEM (Block, Element, Modifier: `.card__title`, `.button--primary`) non sono nate per estetica ma per risolvere due problemi reali del CSS pre-moderno:

1. **Namespace globale.** Ogni classe vive in un unico spazio dei nomi. `.title` scritto da due team collide; `.card__title` no, perché il nome della classe *contiene* il suo scope.

2. **Specificità piatta e prevedibile.** Una sola classe per selettore, sempre specificità (0,1,0), niente battaglie con la cascata.

BEM codifica lo scope e lo stato **nel nome**, perché il linguaggio non offriva un posto migliore dove metterli. Era la risposta giusta a un CSS senza nesting, senza `@layer`, senza `@scope`. La domanda di questo capitolo è: ora che quei costrutti esistono, il nome della classe è ancora il posto giusto?

2.7 La tesi: classi componibili

La nostra risposta è no. Con il nesting nativo, lo scope di uno stato o di una variante può essere espresso *nella struttura del CSS* invece che *nel nome della classe*. Confrontate:

```
<!-- Classi composte (BEM): componente e stato fusi nel nome -->
<button class="button button--primary button--disabled">Salva</
button>

<!-- Classi componibili: componente, variante e stato separati -->
<button class="button primary disable">Salva</button>
```

```
/* Il CSS che rende sicuro il secondo markup */
.button {
  /* stili base */
  padding: 0.5rem 1.25rem;
  border-radius: var(--radius);
  background: var(--color-surface);

  &.primary {
    /* variante */
    background: var(--color-primary);
    color: white;
  }

  &.disable {
    /* stato */
    opacity: 0.5;
    cursor: not-allowed;
  }
```

```
  &:hover:not(.disable) {
    /* interazione, che rispetta lo stato */
    filter: brightness(1.1);
  }
}
```

La parola chiave è `&.primary`: la classe `primary` **da sola non fa nulla**. Produce effetti solo composta con `.button`, perché il selettore richiede entrambe le classi sullo stesso elemento. Lo scoping che BEM otteneva con il nome (`button--primary`), qui lo ottiene la struttura. Vediamo cosa ci guadagniamo, punto per punto.

HTML più leggibile e semantico. `class="button primary disable"` si legge come una frase: è un bottone, primario, disabilitato. Le classi tornano a essere aggettivi, com'erano pensate. Niente prefissi ripetuti tre volte per elemento, niente doppi trattini da decifrare.

Stati riusabili tra componenti. `disable` è un *concetto*, non una proprietà del bottone. Componenti diversi possono dargli ciascuno la propria interpretazione:

```
.button {
  &.disable { opacity: 0.5; cursor: not-allowed; }
}

.text-field {
  &.disable { background: var(--color-surface-dim); pointer-events:
none; }
}

.nav-item {
  &.disable { display: none; }
}
```

Un solo nome nel vocabolario del progetto, tre comportamenti scopati. Con BEM sarebbero tre nomi distinti (`button--disabled`, `text-field--disabled`, `nav-item--disabled`) che ripetono lo stesso concetto senza poterlo condividere: se domani il team decide

che gli stati disabilitati devono anche ignorare i click, con le classi componibili potete aggiungere una regola condivisa; con BEM li rincorrete uno per uno.

Niente esplosione combinatoria. Un componente con 3 varianti e 3 stati, in BEM, tende a generare fino a nove nomi (o a costringervi a combinazioni multiple di modificatori nel markup). Con le classi componibili avete 6 classi che si combinano liberamente: `class="button primary loading"`, `class="button outline disable"`. Le combinazioni vivono nel markup, non nel foglio di stile, e quelle che richiedono stili propri si esprimono con la composizione dei selettori (`&.primary.loading`), solo quando servono davvero.

JavaScript banale. Questo, nella pratica quotidiana, è il vantaggio che si sente di più:

```
/* Con classi componibili: lo stato è una classe atomica */
button.classList.toggle('disable', isSaving);

/* Con BEM: lo stato è un nome da costruire */
button.classList.toggle('button--disabled', isSaving);
/* ...e se il componente è configurabile, il prefisso va
parametrizzato:
   `${blockName}--disabled` — stringhe da assemblare per manipolare
uno stato */
```

`classList` è un'API pensata per classi atomiche. Le classi componibili la usano come è stata progettata; le classi composte la costringono a lavorare con nomi ricostruiti.

Allineamento con ARIA e gli attributi di stato. La piattaforma web esprime già gli stati come proprietà separate dall'identità dell'elemento: `disabled`, `aria-expanded="true"`, `aria-busy="true"`, `:checked`. Nessuno propone `<button-disabled>`: lo stato è un attributo *sul* componente, non una variante *del* componente. Le classi componibili seguono lo stesso modello mentale, e la coerenza si può spingere oltre, selezionando direttamente gli attributi:

```
.button {
  &.disable,
  &[aria-disabled="true"],
  &:disabled {
    opacity: 0.5;
    cursor: not-allowed;
  }
}
```

Un'unica regola scopata copre lo stato espresso via classe, via ARIA o via attributo nativo. Con i nomi composti, l'equivalente esiste ma il parallelismo si perde: `button--disabled` e `[aria-disabled]` restano due mondi lessicali che dicono la stessa cosa.

2.8 Refactoring guidato: da BEM a classi componibili

Mettiamo tutto in pratica su un componente realistico: una card prodotto con variante "in evidenza" e stato "esaurito". Prima la versione BEM:

```
<article class="product-card product-card--featured product-card--
soldout">
  <img class="product-card__image" src="cuffie.jpg" alt="Cuffie
wireless">
  <h3 class="product-card__title">Cuffie wireless</h3>
  <p class="product-card__price">89 €</p>
  <button class="product-card__button button button--primary"
disabled>
    Aggiungi al carrello
  </button>
</article>
```

```
/* Versione Sass/BEM */
.product-card {
  border: 1px solid $color-border;
  border-radius: 8px;

  &--featured { border-color: $color-accent; box-shadow: $shadow-
lg; }
  &--soldout { opacity: 0.6; }
```

```
  &__image { width: 100%; aspect-ratio: 4 / 3; object-fit: cover; }
  &__title { font-size: 1.1rem; }
  &__price { font-weight: 700; }

  &--soldout &__price { text-decoration: line-through; }
}
```

Procediamo in tre passi.

Passo 1 — Gli elementi perdono il prefisso. `product-card__image` esisteva per evitare che `.image` collidesse nel namespace globale. Ma dentro il nesting, lo scope è strutturale: `.product-card { img {...} .title {...} }` seleziona solo ciò che sta nella card. Dove possibile usiamo direttamente l'elemento (`img`, `h3`); dove serve una classe, basta il nome semantico (`.price`).

Passo 2 — Varianti e stati diventano classi componibili. `--featured` diventa `&.featured`, `--soldout` diventa `&.soldout`. Il markup si alleggerisce di conseguenza.

Passo 3 — Le interazioni compongono gli stati. La regola più interessante del componente BEM era `&--soldout &__price`: prezzo barrato quando esaurito. Nella versione componibile diventa una composizione naturale.

Il risultato:

```
<article class="product-card featured soldout">
  <img src="cuffie.jpg" alt="Cuffie wireless">
  <h3 class="title">Cuffie wireless</h3>
  <p class="price">89 €</p>
  <button class="button primary" disabled>Aggiungi al carrello</
button>
</article>
```

```
/* Versione CSS standard */
.product-card {
  border: 1px solid var(--color-border);
  border-radius: 8px;

  img {
```

```
    width: 100%;
    aspect-ratio: 4 / 3;
    object-fit: cover;
  }

  .title { font-size: 1.1rem; }
  .price { font-weight: 700; }

  &.featured {
    border-color: var(--color-accent);
    box-shadow: var(--shadow-lg);
  }

  &.soldout {
    opacity: 0.6;

    .price { text-decoration: line-through; }
  }
}
```

Contiamo i guadagni: il markup è passato da 5 classi prefissate a nomi corti e leggibili; il CSS esprime la relazione "quando esaurita, la card barra il prezzo" con un annidamento che si legge esattamente così; lo stato `soldout` è pronto per essere gestito da JavaScript con un `toggle`; e `featured` è un concetto che domani potrà riusare anche `.review` o `.article-card`, ciascuno con il proprio significato scopato. Notate anche che siamo rimasti dentro la regola dei livelli: componente, poi elementi/stati, e un solo terzo livello (`.price` dentro `&.soldout`) dove la relazione lo richiede.

2.9 I limiti, onestamente

Un manifesto credibile dichiara i propri confini. Eccoli.

Il namespace globale non è sparito. `&.featured` scopa gli *effetti* della classe, non il suo *nome*. Se due team usano `featured` con semantiche opposte e i loro componenti si annidano l'uno nell'altro, un `class="featured"` messo sull'elemento sbagliato può attivare la regola sbagliata. BEM rendeva impossibile questa collisione per costruzione. Mitigazioni moderne esistono — `@scope` per l'incapsulamento vero e `@layer` per governare i conflitti, li

vedremo nel Capitolo 7, ma richiedono disciplina architetturale, mentre BEM la incorporava nel nome.

La ricercabilità. `grep "button--primary"` trova ogni occorrenza di quella combinazione nel progetto. `grep "primary"` trova anche molto rumore. In codebase molto grandi, i nomi lunghi e univoci di BEM sono una forma di documentazione e di tracciabilità che le classi corte non offrono.

La specificità sale di un gradino. `&.primary` vale (0,2,0) contro il (0,1,0) del singolo `.button--primary`. In un progetto ordinato non è un problema, le composizioni battono giustamente le basi, ma se sopra ci costruite override disordinati, l'escalation parte da più in alto. `@layer` è la risposta strutturale, e nel Capitolo 7 la useremo proprio così.

Quando BEM resta una scelta valida: team molto numerosi o distribuiti dove la convenzione *è* la governance; codebase legacy dove convivono stili di epoche diverse senza layer né scope; librerie distribuite a terzi, dove non controllate l'ambiente in cui il CSS atterrerà e la collisione di nomi corti è una certezza statistica; contesti in cui il tooling (linting, ricerca, dead-code detection) è costruito attorno ai nomi BEM. In questi scenari, codificare lo scope nel nome è ancora un compromesso ragionevole, e nesting e BEM non sono nemici: potete annidare regole BEM e godervi comunque metà dei benefici di questo capitolo.

Per tutti gli altri progetti, la maggioranza, la nostra posizione è quella del titolo: lasciate che i nomi tornino semantici e che sia la struttura a fare scoping.

Riepilogo

1. Il nesting nativo è Baseline 2023: regole dentro regole, `&` per pseudo-classi (`&:hover`), composizioni (`&.classe`) e contesti (`.tema &`), at-rule annidabili accanto agli stili che modificano.

2. Due differenze da Sass da imparare subito: `&` è un selettore, non una stringa (niente `&--suffisso`), e la specificità segue il modello di `:is()` — il genitore pesa quanto il suo selettore più specifico.
3. Regola pratica: massimo 2-3 livelli di annidamento; selezionate per ruolo, non ricopiando la struttura del DOM; al quarto livello, estraete un componente.
4. Tesi del libro: con il nesting, stati e varianti si scopano nella struttura (`&.disable`) invece che nel nome (`--disabled`); ne guadagnano l'HTML (leggibilità), il CSS (stati riusabili, niente esplosione di nomi), il JavaScript (`classList.toggle('disable')`) e la coerenza con ARIA.
5. Limiti onesti: il namespace resta globale (mitigato da `@scope` e `@layer`, Capitolo 7), la ricercabilità dei nomi corti è inferiore, la specificità sale di un gradino. BEM resta valido per team molto grandi, librerie distribuite e legacy senza scoping.

Esercizio: il refactoring dell'alert

Consegna. Questo componente di notifica è scritto in Sass/BEM. Riscrivetelo in CSS standard con nesting e classi componibili: l'HTML deve diventare `class="alert warning dismissible"`, e la chiusura via JavaScript deve poter usare `alert.classList.add('closing')`. Attenzione alla regola che lega icona e variante.

```
.alert {
  padding: 1rem;
  border-inline-start: 4px solid $color-info;

  &--warning { border-color: $color-warning; background: $bg-
warning; }
  &--dismissible { padding-inline-end: 3rem; }
  &--closing { opacity: 0; transition: opacity 0.3s; }

  &__icon { inline-size: 1.5rem; }
  &--warning &__icon { color: $color-warning; }
```

```
  &__close { position: absolute; inset-block-start: 1rem; inset-
inline-end: 1rem; }
}
```

Soluzione commentata.

```
.alert {
  position: relative; /* mancava nell'originale: serve al
posizionamento di .close */
  padding: 1rem;
  border-inline-start: 4px solid var(--color-info);

  /* Elementi: il prefisso &__ sparisce, lo scope lo dà il nesting
*/
  .icon { inline-size: 1.5rem; }

  .close {
    position: absolute;
    inset-block-start: 1rem;
    inset-inline-end: 1rem;
  }

  /* Variante: era &--warning, diventa una composizione.
     La regola "icona colorata quando warning" (&--warning &__icon)
     diventa un annidamento che si legge da solo. */
  &.warning {
    border-color: var(--color-warning);
    background: var(--bg-warning);

    .icon { color: var(--color-warning); }
  }

  /* Capacità: dismissible descrive cosa il componente PUÒ fare */
  &.dismissible { padding-inline-end: 3rem; }

  /* Stato transitorio: pensato per essere aggiunto da JS.
     La transition sta nella regola base, così parte quando
     la classe viene AGGIUNTA (se fosse solo in &.closing,
     funzionerebbe comunque in aggiunta ma non gestirebbe
     un'eventuale rimozione: buona abitudine metterla qui). */
  transition: opacity 0.3s;

  &.closing { opacity: 0; }
}
```

```
/* La chiusura, ora banale */
closeButton.addEventListener('click', () => {
  const alert = closeButton.closest('.alert');
  alert.classList.add('closing');
  alert.addEventListener('transitionend', () => alert.remove(),
{ once: true });
});
```

Tre punti da notare. **Primo:** la regola BEM a doppio riferimento `&--warning &__icon`, la più macchinosa dell'originale, è diventata `.icon` annidato dentro `&.warning`: la relazione tra variante ed elemento ora è visibile nella struttura. **Secondo:** abbiamo classificato `warning` come variante, `dismissible` come capacità e `closing` come stato transitorio: tre nomi, tre ruoli, nessun prefisso a distinguerli perché è il *significato* a farlo — coerentemente con il vocabolario fissato nel Capitolo 1. **Terzo:** siamo a due livelli di annidamento, tre solo per `.icon` dentro `&.warning`: dentro la regola.

Nel prossimo capitolo affrontiamo la seconda grande dipendenza da Sass: le variabili. E scopriremo che, anche lì, la versione nativa non è una traduzione, è una promozione.

{

Ti è piaciuto questo capitolo?

Era il capitolo-manifesto, ma la tesi, da sola, è metà del lavoro.

Nel libro completo la vedi *applicata*: un design system costruito da zero, la migrazione passo-passo di un vero progetto Sass fino a spegnere la build, e una landing page responsive senza una sola media query.

www.csssemantico.it

Lascia la tua email sul sito: ti avviso quando il libro esce. Niente spam, ho una sola cosa da annunciarti, ed è questa.

ROBERTO BECCARI

}