The First Modern Programming Language
FLOW-MATIC

FLOW-MATIC, originally known as B-0, was the first English-like data processing language. It was developed for the UNIVAC I at Remington Rand under Grace Hopper during the period from 1955 until 1959.
The First Programming Language

FLOW-MATIC.
Developed by Grace Hopper.
wikipedia.org/wiki/FLOW-MATIC
... Rand management considered the idea unfeasible. In early 1955, she and her team wrote a specification for such a programming language and implemented a prototype.
A 37-Year Head Start

All that wisdom!

We should get as much from their head start as we can.

The time between the first modern programming language and CSS is greater than the time between CSS and now.

Software engineers have been writing code for 37 years longer than front-end developers have.
DRY/Single Source of Truth
Don’t Repeat Yourself

“Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.

— wikipedia.org/wiki/Don't_repeat_yourself
Don’t Repeat Yourself

Every discrete piece of information should exist only once. You shouldn’t need to make the same change several times. Repetition is extra overhead: more to maintain, to go wrong. Increases cognitive overhead. Contributes to bloat.
Don’t Repeat Yourself
Don’t Repeat Yourself

We’ve typed the same value four times.
Don’t Repeat Yourself

$unit: 12px;

.u-margin-top { margin-top: $unit; }
.u-margin-right { margin-right: $unit; }
.u-margin-bottom { margin-bottom: $unit; }
.u-margin-left { margin-left: $unit; }

Much better!
A single source of truth.
Don’t Repeat Yourself

.page-title {
  font-family: "Custom Font", sans-serif;
  font-weight: 700;
}

.btn {
  font-family: "Custom Font", sans-serif;
  font-weight: 700;
}

.pagination {
  font-family: "Custom Font", sans-serif;
  font-weight: 700;
}
Don’t Repeat Yourself

Thematically related but repeated.
@mixin custom-font() {
  font-family: "Custom Font", sans-serif;
  font-weight: 700;
}

.page-title {
  @include custom-font();
}
.
.btn {
  @include custom-font();
}
.
.pagination {
  @include custom-font();
}
Don’t Repeat Yourself

@mixin custom-font() {
  font-family: "Custom Font", sans-serif;
  font-weight: 700;
}

.page-title {
  @include custom-font();
}

.btn {
  @include custom-font();
}

.pagination {
  @include custom-font();
}

Gives the exact same output, but at least we haven’t duplicated anything manually.
Single Source of Truth

"[...] the practice of structuring information models and associated schemata such that every data element is stored exactly once.

— wikipedia.org/wiki/Single_Source_of_Truth
Single Source of Truth

The more philosophical principle behind DRY.
Key data should only exist once in source.
Increases confidence.
Prevents anomalies and disparity.
Makes changes simpler.
Keeps your house in order.
Confusion

DRY in source, not in production.

Not about avoiding repetition…

It’s about avoiding repeating yourself.

Automation of repetition is fine.
Confusion

If you manually type a declaration 50 times in a project, you are repeating yourself: this is not DRY. If you can generate that declaration 50 times without having to manually repeat it, this is DRY: you are generating repetition without actually repeating yourself. This is quite a subtle but important distinction to be aware of.

— csswz.it/1ytQkxp
Confusion

.btn {
  color: white;
  font-weight: bold;
}

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

.message {
  font-weight: bold;
}
Confusion

This is purely coincidental. Don’t try to DRY it out.
Confusion

Don’t DRY if it’s repeated coincidentally.

Repetition in compiled code is fine.

Just avoid duplicating data in source.

Going too far creates awkward and confusing structures in your code.
DRY/Single Source of Truth

Use a preprocessor to store key data in variables.

Make use of mixins to generate repetition for you.

Abstract design patterns out into reusable objects.

Do not DRY anything that is purely coincidental.

Repetition is better than the wrong abstraction.
The Single Responsibility Principle
The Single Responsibility Principle

“[…] the single responsibility principle states that every class should have responsibility over a single part of the functionality provided by the software, and that responsibility should be entirely encapsulated by the class.

— wikipedia.org/wiki/Single_responsibility_principle
The Single Responsibility Principle

AKA: Do one thing, one thing only, and one thing well.
Break bigger monoliths down into individual concerns.
Easier to reason about.
Provides higher composability.
The Single Responsibility Principle

Subway is the epitome of SRP.

Break things down into their smallest possible parts.

Ensure each part fulfils its responsibility very well.

Combine responsibilities to create complex components.

Swap out, remove, or add discrete parts.

Helps you Separate your Concerns.

Gives you incredible opportunity and flexibility.
6,442,450,944
6,442,450,944

6.4bn possible sandwich combinations.
All by offering individual ingredients.
Smaller pieces allow for greater composition.
csswz.it/1XKydI8
The Single Responsibility Principle

#sandwich {
  bread: white;
  meat: chicken;
  salad: lettuce, onion, tomato;
  sauce: mayonnaise;
}

<div id="sandwich">...</div>
The Single Responsibility Principle

```javascript
#sandwich {
  bread: white;
  meat: chicken;
  salad: lettuce, onion, tomato;
  sauce: mayonnaise;
}

<div id="sandwich">...
</div>
```

This is a monolith. It’s difficult to change, swap, or remove things.
The Single Responsibility Principle

.bread, .bread--white {}
.chicken {}
.lettuce {}
.onion {}
.tomato {}
.mayonnaise {}

Perfect! Now we can make our own sandwich from the ingredients we want.
The Single Responsibility Principle

```html
.bread, .bread--white {}
.chicken {}
.lettuce {}
.onion {}
.tomato {}
.mayonnaise {}

<div class="bread bread--white chicken lettuce onion mayonnaise">...</div>

No tomato? No problem!
```
The Single Responsibility Principle

Provide developers with the ingredients.
Let them make the meals.
Let’s look at a more realistic example…
The Single Responsibility Principle

.btn-login {
  display: inline-block;
  padding: 2em;
  background-color: green;
  color: white;
}

The Single Responsibility Principle

.btn-login {
  display: inline-block;
  padding: 2em;
  background-color: green;
  color: white;
}

Mixing responsibilities.
Base.
Structural.
Cosmetic.
The Single Responsibility Principle

.btn {
    display: inline-block;
}

.btn--large {
    padding: 2em;
}

.btn--positive {
    background-color: green;
    color: white;
}
The Single Responsibility Principle

.btn {
  display: inline-block;
}

.btn--large {
  padding: 2em;
}

.btn--positive {
  background-color: green;
  color: white;
}
The Single Responsibility Principle

```css
.btn {
    ... 
}

.btn--large {
    ... 
}

.btn--small {
    ... 
}

.btn--positive {
    ... 
}

.btn--negative {
    ... 
}

.btn--full {
    ... 
}
```

Plus now we can combine these classes with others to make lots of varieties of button.
The Separation of Concerns
The Separation of Concerns

[...] It is, that one is willing to study in depth an aspect of one’s subject matter in isolation for the sake of its own consistency [...] But nothing is gained—on the contrary!—by tackling these various aspects simultaneously. It is what I sometimes have called ‘the separation of concerns’ [...] it does not mean ignoring the other aspects, it is just doing justice to the fact that from this aspect’s point of view, the other is irrelevant. It is being one- and multiple-track minded simultaneously.
The Separation of Concerns

Each thing responsible for itself and nothing more.
Reason about and study features in isolation.

In CSS:
Only bind CSS onto CSS-based classes only.
Don’t write DOM-like selectors.
Don’t bind CSS onto `data-*` attributes.
Don’t bind JS onto CSS classes.
The Separation of Concerns

// Binding CSS onto accessibility hints.
[role="navigation"] { ... }

// Putting DOM information into our CSS.
header nav ul li a { ... }

// Using HTML to provide cosmetics.
<font color="red">

// Binding JS onto styling hooks.
document.getElementsByClassName('nav');
The Separation of Concerns

<nav role="navigation">
  <ul>
    <li>
      <a>...</a>
    </li>
    <li>
      <a>...</a>
    </li>
  </ul>
</nav>
The Separation of Concerns

<nav role="navigation">
  <ul>
    <li>
      <a>...</a>
    </li>
    <li>
      <a>...</a>
    </li>
  </ul>
</nav>
The Separation of Concerns

[role="navigation"] { ... }

[role="navigation"] > ul { ... }

[role="navigation"] > ul > li { ... }

Ewww! Loading our CSS with DOM information.
The Separation of Concerns

<nav class="site-nav js-site-nav" role="navigation">
  <ul class="site-nav__list">
    <li class="site-nav__item">
      <a class="site-nav__link">...</a>
    </li>
    <li class="site-nav__item">
      <a class="site-nav__link is-active">...</a>
    </li>
  </ul>
</nav>
The Separation of Concerns

```html
<nav class="site-nav js-site-nav" role="navigation">
  <ul class="site-nav__list">
    <li class="site-nav__item">
      <a class="site-nav__link">...</a>
    </li>
    <li class="site-nav__item">
      <a class="site-nav__link is-active">...</a>
    </li>
  </ul>
</nav>
```

Semantic concerns.
The Separation of Concerns

<nav class="site-nav js-site-nav" role="navigation">
  <ul class="site-nav__list">
    <li class="site-nav__item">
      <a class="site-nav__link">...</a>
    </li>
    <li class="site-nav__item">
      <a class="site-nav__link is-active">...</a>
    </li>
  </ul>
</nav>

Accessibility concerns.
The Separation of Concerns

Stylistic concerns.
The Separation of Concerns

```html
<nav class="site-nav js-site-nav" role="navigation">
  <ul class="site-nav__list">
    <li class="site-nav__item">
      <a class="site-nav__link">...</a>
    </li>
    <li class="site-nav__item">
      <a class="site-nav__link is-active">...</a>
    </li>
  </ul>
</nav>

Behavioural concerns.
The Separation of Concerns

Grid systems are a great example.
Handle your layout completely separately to your components.

Writing CSS in JS breaks the Separation of Concerns.

Can’t reconsider your JS architecture without having to reconsider your CSS architecture.
The Separation of Concerns

“If in 14 months you find a new view library or framework you want to try out, you’re out of luck. You will have to invest a lot of time into pulling styles back out of JavaScript modules and into stylesheets again.”

— keithjgrant.com/posts/against-css-in-js.html
Immutability
Immutability

"...an immutable object is an object whose state cannot be modified after it is created."

— wikipedia.org/wiki/Immutable_object
Immutability

Provides confidence.
Makes things predictable.
Helps debugging.
Reduces cognitive overhead.
Removes caveats, states, and conditions.
Immutability

col-6 {
  width: 50%;
}

Immutability

```css
.col-6 {
  width: 50%;
}

@media screen and (max-width: 480px) {
  .col-6 {
    float: none;
    width: 100%;
  }
}
```
Immutability

.col-6 {
  width: 50%;
}

@media screen and (max-width: 480px) {
  .col-6 {
    float: none;
    width: 100%;
  }
}

This has mutated!
Immutability

.col-6 has one input, but two potential outputs. Outcome depends on how/when you observe it. It has been mutated. Mutable state leads to confusion and unexpected outcomes. Particularly common in CSS.
Immutability

.col-6 {
    width: 50%;
}

@media screen and (max-width: 480px) {
    .col-6@sm {
        float: none;
        width: 100%;
    }
}

Use a different class.
Immutability

.section class="sub-content">
  <h2 class="u-text-center">...</h2>
</section>
Immutability

Specificity mismatch. This will be aligned left!
Immutability

Parts of the codebase are able to mutate other parts.

Unpredictable outcomes.

Unexpected side effects.

This is fixable.
Immutability

The only time to use it.
Immutability

.btn {
    font-size: 1em;
}

.promo .btn {
    font-size: 1.2em;
}
Immutability

.btn {
    font-size: 1em;
}

.promo .btn {
    font-size: 1.2em;
}

One class, two outcomes.
Immutability

Two classes, two outcomes.
Immutability

Don’t have several states of the same thing.
Use Modifiers or Responsive Suffixes.
Use !important to force immutability.
Brings us nicely onto…
Cyclomatic Complexity
Cyclomatic complexity is a software metric used to indicate the complexity of a program. It is a quantitative measure of the number of linearly independent paths through a program’s source code.

— wikipedia.org/wiki/Cyclomatic_complexity
Cyclomatic Complexity

\[ M = E - N + 2P \]

Basically just the number of IFs/ELSEs.

A form of static analysis.

Counting the number of paths through a program.

The amount of potential outcomes given certain conditions.

Higher complexity is bad: simpler is always better.
Cyclomatic Complexity

\[ M = E - N + 2P \]

Basically just the number of IFs/ELSEs.

A form of static analysis.

Counting the number of paths through a program.

The amount of potential outcomes given certain conditions.

Higher complexity is bad: simpler is always better.
Cyclomatic Complexity
Cyclomatic Complexity

```html
@if div {
  @if .main {
    @if section {
      @if .content {
        @if h1 {
          @if a {
            @if span {
              // Do stuff.
            }
          }
        }
      }
    }
  }
}
```
Cyclomatic Complexity

Subject.

This is the bit we actually care about.
Cyclomatic Complexity

Conditions.

All of this is just complexity.
Cyclomatic Complexity

Start off explicitly.

Don’t add needless complexity.
Cyclomatic Complexity

Deeply nested or qualified selectors are bad. They carry a higher Cyclomatic Complexity.
Reduce by using much shorter selectors.
Get straight to the point.
Remove as many conditions and caveats as possible.
Start with the correct subject.
Cyclomatic Complexity
The Open/Closed Principle
The Open/Closed Principle

"Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification."

— wikipedia.org/wiki/Open/closed_principle
The Open/Closed Principle

Never change anything at its source.

Avoid the Domino Effect.

Doing so causes visual regressions.

Hard to keep track of the knock-on effects.

Always make changes via extension (i.e. addition).

Possibly the most useful principle for dealing with other peoples’ code.
The Open/Closed Principle

“ [...] once completed, the implementation of a class could only be modified to correct errors; new or changed features would require that a different class be created. That class could reuse coding from the original class through inheritance.

— wikipedia.org/wiki/Open/closed_principle
The Open/Closed Principle

```css
.btn {
    ...  
    padding: 1em 2em;
}
```
The Open/Closed Principle

.btn {
  ... padding: 1em 2em;
}
The Open/Closed Principle

.btn {
  ...
  padding: 1em 2em;
}

.promo .btn {
  padding: 1.5em 2.5em;
}
The Open/Closed Principle

\[
.btn \{
  ...
  padding: 1em 2em;
}\]

\[
.pomo .btn \{
  padding: 1.5em 2.5em;
}\]

Even this is risky as we’re still modifying the base button class.
The Open/Closed Principle

.btn {
  ...
  padding: 1em 2em;
}

.btn--large {
  padding: 1.5em 2.5em;
}

Perfect! A brand new class adds the changes that we want to safely opt in to.
The Open/Closed Principle

A safe way to make changes.

Everything gets opted into explicitly.

Prevents changes from happening one-sidedly; the developer has to add the class into the markup as well.

A second layer of safety: changes can’t be actioned from one place alone.

Build things forward.

Analogous to rewriting Git history.

Safe way of working with legacy.
Orthogonality
Orthogonality

Orthogonality in programming language design is the ability to use various language features in arbitrary combinations with consistent results.

— wikipedia.org/wiki/Orthogonality
Orthogonality

Reduces interdependence.
Improves composability.
Separates concerns.
Reduces collisions.
Removes side effects.

Good litmus test: can we reorder imports?
@import "elements.headings";
@import "elements.links";
@import "elements.lists";
@import "elements.images";
@import "elements.quotes";

@import "objects.wrappers";
@import "objects.layout";
@import "objects.headline";
@import "objects.media";
@import "objects.list-inline";

@import "components.page-head";
@import "components.page-foot";
@import "components.logo";
@import "components.nav-primary";
@import "components.nav-secondary";
@import "components.masthead";
@import "components.panel";
@import "components.score";
@import "components.buttons";
@import "components.avatar";
@import "components.testimonials";
@import "components.calendar";
@import "components.headline";
@import "components.promo";

@import "scope.prose";

@import "trumps.headings";
@import "trumps.widths";
Orthogonality

(How well) can we arbitrarily combine things?
The implication is that they don’t rely on one another.
The hallmarks of a flexible and modular system.
Orthogonality

Another good test: will it nest?

Can things be combined in the DOM?

Well-scoped selectors improve orthogonality.
A fun and vibrant city


Discount travel to Melbourne

A fun and vibrant city

Harry Roberts
Collisions!
A fun and vibrant city

Proper scoping provides orthogonality.
The Moustache Principle
The Moustache Principle

"Just because you can, it doesn’t mean that you should."

— Harry Roberts
Thank You

Harry Roberts

csswizardry.com

csswizardry@gmail.com

@csswizardry

speakerdeck.com/csswizardry