HOLA, ARGENTINA!
HI, I’M HARRY

@csswizardry

Consultant Front-end Architect

CSS Architecture, Performance, Scale
HI, I’M HARRY

@csswizardry
Consultant Front-end Architect
CSS Architecture, Performance, Scale
And I really hate refactoring CSS
WHAT DO WE TALK ABOUT WHEN WE TALK ABOUT REFAC TORING?
“...the process of changing a software system in such a way that it does not alter the external behaviour of the code, yet improves its internal structure.”

— Martin Fowler
THIS ISN’T* FOR OUR USERS
THIS ISN’T (DIRECTLY) FOR OUR USERS
THREE KINDS OF REFACTORIZING

1. **As-You-Go**: I hard-coded it all to see if it would work; now I need to make it production-ready.

2. **Technical Debt**: We built what we could in the time we had; we want to tidy things up.

3. **Rewrites and Overhauls**: It’s become too difficult and expensive to maintain; we need to rewrite.
As-You-Go  

Technical Debt  

Rewrites and Overhauls  

Project Lifespan  

Cost/Scale of Refactor
TECHNICAL DEBT
“A ‘debt’ means that you [acquired] something now for a long-term financial burden. This burden is not just about repaying what you got: there is ‘interest’. It means that, even if you pay your debt timely, you’ll pay more than you took, and if you don’t, your debt will keep increasing [...] if you ignore a debt long enough, it will become unpayable and you’ll go ‘bankrupt’.”

— Maiz Lulkin, csswz.it/2adZH3M
TECHNICAL DEBT

We’re going to incur some of it, fact.
It’s vitally important that we keep up repayments.
People forget that debt repayments incur interest.
Schedule in bug-fixing and tech-debt cleanup every sprint.
Make and prove the business case for refactoring.
WHEN (AND WHAT) TO REFACTOR
YOU SHOULD ALWAYS BE REFACTORIZING IN SOME FORM OR ANOTHER
WHEN TO REFACTOR

If the (projected) cost of maintenance is higher than rewriting.
If the current version is slowing you down.
If the new version provides tangible benefit.
<table>
<thead>
<tr>
<th></th>
<th>Existing Code</th>
<th>Refactored Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cost Now</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>Cost Next Time</td>
<td>2</td>
<td>0.25</td>
</tr>
<tr>
<td>Cost Next Time</td>
<td>2.5</td>
<td>0.25</td>
</tr>
<tr>
<td>Cost Next Time</td>
<td>2.5</td>
<td>0.25</td>
</tr>
<tr>
<td>Total Cost</td>
<td>9</td>
<td>7.75</td>
</tr>
</tbody>
</table>
INVESTMENT IS THE OPPOSITE OF DEBT
WHEN NOT TO REFACTOR

If you’re not actually being slowed down by something.
If it’s something that can be ignored or avoided.
If it’s something that can be captured by a rewrite later on.
If a rewrite is the better solution.
“We want to rewrite our CSS onto BEM.”
NEW THINGS

Honestly, I love BEM as much as the next person...
...but taking two weeks out to refactor your CSS onto it is not going to pay itself back very quickly.
Is it actually worth it?
“We think the code for this nav is pretty ugly.”
UGLY CODE

Is it causing actual problems for you right now?
How often do you have to actually work with this code?
Can we just leave it to keep working as it is?
REFACTORING TUNNELS
$ git reset --hard origin/master
AVOID LONG REFACTORING TUNNELS
REFACTORING TUNNELS

Avoid refactoring anything that runs through the entire project. Takes too long, and leaves things messy. Huge deltas to merge (i.e. conflicts). Easy to (inadvertently) introduce more problems. Instead, pick off things with a limited and clear scope.
REFACTORING TUNNELS

Find a short tunnel (e.g. refactoring just the nav).
Get the work completed.
Either get back onto features if needed...
...or pick another short tunnel.
Rinse and repeat.
The site refactors itself.
REFACTOR IN ISOLATION
Don’t (re)build features into a dirty codebase. 
You’ll be relying on a stale environment. 
Fire open JSFiddle and build the new one there. 
Port it back into your project. 
Do any tidy-up work there.
<header id="header" class="1-header u-cf js-header html">
  <div class="js-navigation-header navigation-container--collapsed">
  </div>
  <div class="gs-container 1-header__inner">
    
    .caption:before,
    figcaption:before,
    blockquote.quoted:before {
      background-image: url(https://assets.gum.co.uk/images/global/5fefd518a4541617ad095547a3beb74/sprite.png);
    }
    .commercial--soulnates .--commercial {
      background-image: url(https://assets.gum.co.uk/images/global/5fefd518a4541617ad095547a3beb74/sprite.png);
    }

    At Hired, top companies apply to hire you. Get 5+ job offers in 1 week!
  
  ads via Carbon
SAFE TO ASSUME WE’RE GOING TO EXPERIENCE SOME BREAKAGES
ALL: INITIAL;
ALL: INITIAL;

Effectively stops inheritance.
Prevent legacy styles from leaking into fresh work.
A very progressive way of defending against legacy.
Hello!
Hello!
These rules will not get inherited from an ancestor anymore.
So we have to define them on the leaf node.
## CSS all property

A shorthand property for resetting all CSS properties except for `direction` and `unicode-bidi`.

<table>
<thead>
<tr>
<th>Feature</th>
<th>IE</th>
<th>Edge</th>
<th>Firefox</th>
<th>Chrome</th>
<th>Safari</th>
<th>Opera</th>
<th>iOS Safari</th>
<th>Opera Mini</th>
<th>Android Browser</th>
<th>Chrome for Android</th>
</tr>
</thead>
<tbody>
<tr>
<td>Current aligned</td>
<td>8</td>
<td>13</td>
<td>46</td>
<td>51</td>
<td>29</td>
<td>45</td>
<td>49</td>
<td>50</td>
<td>4.3</td>
<td>4.4</td>
</tr>
<tr>
<td>Usage relative</td>
<td>11</td>
<td>14</td>
<td>48</td>
<td>53</td>
<td>52</td>
<td>10</td>
<td>9.1</td>
<td>38</td>
<td>4.4</td>
<td>4.4</td>
</tr>
<tr>
<td>Show all</td>
<td>14</td>
<td></td>
<td>49</td>
<td>54</td>
<td>TP</td>
<td></td>
<td>39</td>
<td>40</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
DEFENCE.CSS
What happens when you need to run refactored code and legacy code side-by-side?
SKY UI TOOLKIT

Had an existing/legacy toolkit.
Modernised design and architecture required.
New toolkit developed.
Had to be rolled out very gradually (big project).
Homepage was first candidate.
Site now using old and new toolkits.
WE HAD TO RUN OLD AND NEW IN TANDEM
DEFENCE.CSS

A whole new project.
Run OSS-style: other teams consume and contribute.
Exists only to fix temporary/transient fallout and breakages.
Stuffed full of crude fixes, !importants, hacks.
Just type until it looks okay.
This is the worst CSS you will ever write. And that’s okay.
And what’s in this file...?
.RF-* CLASSES
Prefix any refactored classes with rf-.
This means developers can see which classes are new…
But it also means we can do this:
/**
 * If it’s a class containing the string rf-,
 * put a green border around it.
 */

[class*="rf-"] {
    outline: 5px solid green;
}

Corbyn ballot challenge rejected by High Court.
THIS IS THE WORK WE’VE DONE
/**
 * If it’s a class, but isn’t a class containing
 * the string rf-, put a red border around it.
 */

[class]:not([class*="rf-"]) {
    outline: 5px solid red;
}
UK Today

Rain moving east, brighter with showers later for some.

Rain in the west will move eastwards, this persistent and locally heavy over Northern Ireland, northern England and southern Scotland. Brighter with showers over northern Scotland. Turning brighter across Wales and southern England later with a risk of heavy showers.

Find a Forecast

Enter a town, county or UK postcode

Favourite Locations 0

UK Forecast Video

LAST UPDATED 12:28 BST, THURSDAY 28 JUL
THIS IS THE WORK LEFT TO DO
REF-HACK-TORING SPECIFICITY
HACKING SPECIFICITY

Dealing with specificity on a legacy project.
Newly refactored work being overridden by existing selectors.
!important to manage it safely.
We can hack specificity with minimal side effects.
<a href="#" class="foo" id="bar">...</a>
#bar {
    color: blue;
}

.foo {
    color: green;
}

a {
    color: red;
}
HACKING SPECIFICITY

Three differently weighted selectors.
All working against their source order.
We could use !important to force different precedence.
Or we can hack our specificity around.
Element &
class equivalent

Element &
class
class equivalent
& element
```javascript
/**
 * Same specificity as one class.
 */
[id="bar"] {}

/**
 * Same specificity as two classes.
 */
.foo.foo {}

/**
 * Same specificity as a class and an element.
 */
:root a {}
```
THESE ARE HACKS
THESE ARE HACKS

Ideally: Refactor until you don’t need the hacks.

Realistically: Use one of these hacks to solve the problem.

Never: Use !important.
“The idea of shame.css is that you have a totally new stylesheet reserved just for your hacky code. Code you have to write to get the release out on time, but code that makes you ashamed.”

— Harry Roberts, csswz.it/113CPn2
A DUMPING GROUND FOR ALL THOSE BITS OF CSS YOU’RE A LITTLE BIT ASHAMED OF
Hacks are inevitable.
Isolate and signpost them.
Makes everyone else aware of them.
Easy to find and fix things.
/**
 * The `.promo a {}` selector keeps overriding the
 * button’s styles. Increase its specificity here
 * until I get chance to refactor the promo boxes.
 *
 * Harry Roberts <csswizardry@gmail.com> 2016-05-01
 */

.btn.btn {
    text-decoration: none;
}
Self-writing todo list.
Keeps good code nice and clean (Broken Windows Theory).
See which parts of the codebase are particularly problematic.
$ git blame shame.css
$ git blame shame.css
A SECOND CHANCE
“I’m a civil engineer by trade... we don’t get to rework our architecture.”
REFAC T O R I N G IS A  
SECOND CHANCE THAT MOST  
INDUSTRIES DON’T GET
REMEMBER...
REMEMBER

Prevention is cheaper than the cure.
Technical Debt is fine, just make sure you keep up repayments.
Only refactor once you can see tangible benefit.
Avoid long Refactoring Tunnels.
Isolate and highlight both hacks and refactored work.
“Always leave the campground cleaner than you found it.”

— Robert Baden-Powell
THANK YOU

Harry Roberts
csswizardry@gmail.com
speakerdeck.com/csswizardry
csswizardry.com
@csswizardry