Beauty is not for its own sake... Good style improves maintainability of code, improves team efficiency, etc. We have already made extensive use of ESLint and its “style checking”. Today we will learn about some other tools for evaluating SW style.

I want to distinguish between formatting and the "style" we are talking about here. Formatting is indeed part of style, but what we are talking about here are implementation choices that produce correct results but are nonetheless fraught because that approach makes your code difficult to maintain, is likely to introduce bugs when you others change the code, is more likely to have subtle bugs (i.e. code only appears to work) or more.

Note that we also automate formatting (e.g. with prettier). Why? Easier to read code in a common format, don’t want to argue with teammates over something that can be very personal.

Image from https://www.npmjs.com/package/eslint-config-airbnb-bundle

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
A code smell doesn’t mean that something is wrong, i.e. code smells are not bugs. The program may correctly function. Instead a code smell is a “surface indication” or “hint” that deeper problems might exist. Think of a code smell as a warning sign.

“Short” and “Do one thing” tend to be correlated. It is obvious why those might be smells, what about “few arguments” and “levels of abstraction”?

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
Why “lots of arguments” smells

- Hard to get good testing coverage
- Hard to mock/stub while testing
- Boolean arguments should be a “yellow flag”
  If function behaves differently based on Boolean argument, maybe it should be 2 functions
- If arguments “travel in a pack”, maybe you need to extract a new object/class
  Same argument for a “pack” of methods

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
The same attributes that make for good news stories, also make for good code.

Don’t have one method that does everything. Divide into understandable pieces, and have methods call others. That is one method/function that orchestrates all the work.

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
Quantitative: ABC Software Metric

Counts **Assignments, Branches, Conditions:**

\[ score = \sqrt{A^2 + B^2 + C^2} \]

```javascript
function foo()
  const a = eval("1+1");
  if (a === 2) {
    console.log("yay");
  }
}
```

\[ \sqrt{1 + 2^2 + 2^2} = 3 \]

Guidance: ≤20 per method

ABC is strictly a software size metric, although it has often been misconstrued as a complexity metric. Designed in part as an alternative to LOC. Rules are language-specific…, e.g. here functions are considered “branches”.

**Fun fact:** Flog is a Ruby-specific version of the ABC metrics. The highest Flog score ever seen on Code Climate for a single method is 11,354 (https://codeclimate.com/blog/deciphering-ruby-code-metrics/).

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
Quantitative: Cyclomatic complexity

Linearity-independent paths thru code

score = E − N + 2P

E edges, N nodes, P connected components

function myFunction {
    while(...) {
        ....
    }
    if (...) {
        do_something
    }
}

A graphical measurement of the number of possible paths through the normal flow of a program. The graph is the control flow graph (take compilers); each node is a basic block. If you just had “straight line” code, the complexity would be 1.

Here, E=9, N=8, P=1, so CC=3 (there is only one connected component, the entire program).

Image from Wikipedia
Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
Quantitative: Metrics

<table>
<thead>
<tr>
<th>Metric</th>
<th>Tool</th>
<th>Target score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code-to-test ratio</td>
<td>Plato/Jest</td>
<td>$\leq 1:2$</td>
</tr>
<tr>
<td>C0 (statement) coverage</td>
<td>Jest</td>
<td>$70%$+</td>
</tr>
<tr>
<td>Assignment-Branch-Condition score</td>
<td>None for JS</td>
<td>$&lt; 20$ per method</td>
</tr>
<tr>
<td>Cyclomatic complexity</td>
<td>Plato, ESLint</td>
<td>$&lt; 10$ per method (NIST)</td>
</tr>
</tbody>
</table>

Use metrics “holistically”

- Better for identifying where improvement is needed than for signing off
- Look for “hotspots”, i.e. code flagged by multiple metrics (what services like CodeClimate do...)

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
Refactoring

- Start with code with smells
- Through a series of *small steps*, transform code eliminate those smells
- Protect each step with tests
- *Minimize time during which tests are “red”*

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
Refactoring has common patterns too

Fowler et al. created a **catalog** of common refactorings

**Decompose Conditional**

You have a complicated conditional (if-then-else) statement. Extract methods from the condition, then part, and else part.

```plaintext
if (date.before(SUMMER.START) || date.after(SUMMER.END))
    charge = quantity * _winterRate * _winterServiceCharge;
else charge = quantity * _summerRate;
```

```plaintext
if (notSummer(date))
    charge = winterCharge(quantity);
else charge = summerCharge(quantity);
```

https://refactoring.com/catalog/decomposeConditional.html

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
Which of the following is not a goal of method level refactoring?

A. Reduce code complexity
B. Eliminate code-smells
C. Eliminate bugs
D. Improve testability

Answer: C

Recall that code smells are not bugs, but instead warnings signs that there might be deeper problems.

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
On December 31, 2008, all Microsoft Zune MP3 players that were booted up on that day mysteriously froze. If you rebooted on January 1, 2009, it would work again. This example includes the buggy code, as explained in this [blog post](http://www.zuneboards.com/forums/showthread.php?t=38143), transliterated to ES6. Try 10593 (Dec 31, 2008) or 1827 (Dec 31, 1984) to trigger the bug (an infinite loop).

* `v0.js`: Original transliterated
* `v1.js`: Refactored with more relevant variable names, but no other changes
* `v2.js`: Extract the `isLeapYear` function to improve readability and introduce a test suite
* `v3.js`: Extracts `addLeapYear` and `addCommonYear` functions to reduce function complexity. Adds additional test cases for extracted functions.
* `v4.js`: Fixes logic error

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
Quantitative metrics

Mean Cyclomatic Complexity

Maintainability

v0  v1  v2  v3  v4

v0  v1  v2  v3  v4
What did we do?

Made date calculator easier to read and understand using simple refactorings

*Extract method* is one of most common, and allows testing “helper” methods separately

Found a bug!

If we had developed using TDD, might have been prevented...

Improved code metrics

[https://github.com/csci312-common/zune-refactor](https://github.com/csci312-common/zune-refactor)

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
Which SOFA guidelines is most important for unit testing?

A. Short  
B. Do One thing  
C. Have Few arguments  
D. Stick to one level of Abstraction

Answer: C

There are no absolute right or wrong answers. We can make arguments for several and could come up with counter-examples for specific programs why one is more important than the others. That said here is my take:

1. Short: Long, but straight-line, code could still be easy to test.
2. One Thing: If code is doing multiple things, but each is simple, could still be easy to test.
3. Few Arguments: My answer. If the arguments matter, then the code has lots of degrees of freedom to test.
4. One level abstraction: Hard to understand, but not necessarily hard to test.

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
Summary

**Goal**: Improve code structure (as measured by quantitative & qualitative measures) without changing functionality (as measured by tests)

1. Use metrics as a guide to where you can improve your code
2. Apply *refactorings* (found in following slide, in Refactoring books, on line, etc.)
3. At each step, test newly-exposed *seams*, then stub/mock them out in higher-level tests

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
Other smells and their remedies

<table>
<thead>
<tr>
<th>Smell</th>
<th>Refactoring that may resolve it</th>
</tr>
</thead>
<tbody>
<tr>
<td>Large class</td>
<td>Extract class, subclass or module</td>
</tr>
<tr>
<td>Long method</td>
<td>Decompose conditional</td>
</tr>
<tr>
<td></td>
<td>Replace loop with collection method</td>
</tr>
<tr>
<td></td>
<td>Extract method</td>
</tr>
<tr>
<td></td>
<td>Replace temp variable with query</td>
</tr>
<tr>
<td></td>
<td>Replace method with object</td>
</tr>
<tr>
<td>Long parameter list/data clump</td>
<td>Replace parameter with method call</td>
</tr>
<tr>
<td></td>
<td>Extract class</td>
</tr>
<tr>
<td>Shotgun surgery; Inappropriate intimacy</td>
<td>Move method/move field to collect related items into one DRY place</td>
</tr>
<tr>
<td>Too many comments</td>
<td>Extract method</td>
</tr>
<tr>
<td></td>
<td>Introduce assertion</td>
</tr>
<tr>
<td></td>
<td>Replace with internal documentation</td>
</tr>
<tr>
<td>Inconsistent level of abstraction</td>
<td>Extract methods &amp; classes</td>
</tr>
</tbody>
</table>

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
What makes code “legacy”?

Still meets customer need, *and*

- You didn’t write it, and it’s poorly documented
- You did write it, but a long time ago (and it’s poorly documented)

“Legacy code is simply code without tests” [regardless of who wrote it or how pretty it is]

-Michael Feathers

Here we mean “legacy” with its negative connotations. However we should remember that legacy SW is by definition successful, otherwise it would no longer be in use.

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
Feathers’ two ways to approach modifying legacy code

<table>
<thead>
<tr>
<th>Edit and Pray</th>
<th>Cover and Modify</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Familiarize yourself with the relevant code</td>
<td>1. Write tests that cover the code you will modify (creating a “safety blanket”)</td>
</tr>
<tr>
<td>2. Plan the changes you will make</td>
<td>2. Make the changes</td>
</tr>
<tr>
<td>3. Make the planned changes</td>
<td>3. Use tests to detect unintended effects</td>
</tr>
<tr>
<td>4. Poke around to make sure you didn’t break anything</td>
<td></td>
</tr>
</tbody>
</table>

‘Superficially, Edit and Pray, seems like “working with care”’ With that care exercised up front. ‘But safety isn’t solely a function of care. … Effective software change, like effective surgery, really involves deeper skills. Working with care doesn’t do much for you if you don’t use the right tools and techniques’ – Michael Feathers

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
An Agile approach to legacy code

1. Identify places you need to change (termed “change points”)
2. Add “characterization tests” to capture how the code works now (in TDD+BDD cycles)
3. Refactor the code to make it more testable or to accommodate the changes
4. When code is well factored and well tested, make your changes!
5. Repeat...

Think of this as embracing change on long time scales. Keep Baden-Powell’s motto in mind: “Try to leave the world a little better than you found it”.

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
If you’ve been assigned to modify legacy code, which of the following statements about that code base do you most hope will be true?

A. It was originally developed using Agile techniques
B. It is well covered by tests
C. It’s nicely structured and easy to read
D. Many of the original design documents are available

Answer: B

Recall that tests are your "safety blanket". Why are the others wrong? Agile techniques don't guarantee modifiable code. Similarly, just because the code is nicely structured also doesn't mean you can confidently make changes. How will you know if you broke something? While it would be nice there were original design documents available, in all likelihood those documents don't correspond to the current code and again don't help you know if you broke anything.
Exploring a legacy codebase: Step 1

Get the code to run!

- In a either production-like or development-like setting
- Ideally with something resembling a copy of production database
- A catch: Some systems may be too large to copy

Learn the user stories: Have customers show you how they use the application

The first should happen on a scratch branch and with a copy of DB (so you won’t break anything and can throw your experiments away).
The second helps you define the ground truth for the application. In the best case, these demos are consistent with the behavioral tests, in the worst case...

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
Exploring a legacy codebase: Step 2+

2. Inspect the database schema
3. Try to build a model interaction diagram
   Can be automated for some frameworks, e.g. Rails
4. Identify the key (highly connected) classes
   Recall Class-Responsibility-Collaborators (CRC) cards
5. (Extend) design docs as you go:
   Diagrams
   README, GitHub wiki, etc.
   Add JSDoc comments to create documentation automatically

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
Adding tests: Getting started

- You don’t want to write code without tests
- You don’t have tests
- You can’t create tests without understanding the code

How do you get started?

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
Characterization Tests

Establish the *ground truth about how the SW works today*

Repeatability tests ensure current behaviors aren’t changed (even if buggy)

Integration tests are a natural starting point (b/c they are typically “black box”)

*Pitfall: Don’t try to make improvements at this stage!*

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
Unit- and Functional-level characterization tests

Use the tests to help you learn as you go:

test('it should calculate sales tax', () => {
  const order = Order.fromJson({});
  expect(order.computeTax()).toBe(-99.99);
});

ValidationError: total is a required property

test('it should calculate sales tax', () => {
  const order = Order.fromJson({ total: 100.00 });
  expect(order.computeTax()).toBe(-99.99);
});

Expected value to be: -99.99 Received: 8

test('it should calculate sales tax', () => {
  const order = Order.fromJson({ total: 100.00 });
  expect(order.computeTax()).toBe(8.00);
});

✓ it should calculate sales tax

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
Which of the following is a difference between integration-level and unit-level characterization tests?

A. Unit tests can be created by automating user actions, integration tests cannot
B. Integration tests require more information about how the code works
C. Integration tests are more likely to depend on the production database

Answer: C

Because integrations tests don't involve isolated components, they are more likely to depend on the database somehow and may require a test database or a form of high-level mocking. We have already seen examples of automating user interactions, e.g. "clicking" (answer A) and know what integration tests require less information about how the code works since integration tests treat the code as a black box.
What is the best tool for detecting (and fixing) code smells/problems?

There is no best tool!
The primary enforcement mechanism is your self-discipline!
Beautiful code is the result of your professionalism to do the “Right Thing” not the easy thing. The tools just help along the way.

As another example, in the movie “Who Framed Roger Rabbit”, the actor bumped a lamp during filming. The animators went back and very carefully updated the animation to ensure the animated shadows match the movement of the lamp. That is they did the ”right thing” (definitely not the easy thing), but in doing so help maintained the illusions that makes the movie immersive.

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.