Beyond Correctness

Can we give feedback on software *beauty*?

- Guidelines on what is beautiful?
- Qualitative evaluations?
- Quantitative evaluations?

What tools are available for "higher level" evaluation of our code?

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 or 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? It is easier to read code in a common format, and we 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

Qualitative: “Code smells”

SOFA captures symptoms that often indicate code smells in functions/methods:

- Is it Short?
- Does it do One thing?
- Does it have Few arguments?
- Is it at a consistent level of Abstraction?

A code smell doesn’t mean that something is wrong, i.e., code smells are not bugs. The program may function correctly. 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 code that doesn’t meet that criteria might “smell”, but what about “few arguments” and “levels of abstraction”?

Why “lots of arguments” smells

- Hard to get good testing coverage
- Hard to mock 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

Single level of abstraction

- Complex tasks need divide & conquer
- Like a good news story, classes, methods, etc. should read "top down"
  - + Start with a high-level summary of key points, then go into each point in detail
  - + Each paragraph deals with 1 topic
  - – Rambling, jumping between "levels of abstraction" rather than progressively refining
- Want to avoid "leaky abstractions"

What are some attributes of a good news story (and a not so good news story)? … The same attributes that make for good news stories, also make for good code.

In the context of code, that means don't have one function/method that does everything. Divide it into understandable pieces, and have methods call others. That is one method/function orchestrates the work of many others. Recall we saw a similar idea in the context of React components where we talked about components implementing or composing, i.e., that components should generally either implement specific functionality or compose (group) other components together. From the blog post: 'A component should be described either as a "component that implements various stuff" or as a "component that composes various components together", not both. (https://www.developerway.com/posts/components-composition-how-to-get-it-right)

Quantitative: ABC Software Metric

Counts **A**ssignments, **B**ranches, **C**onditions:

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

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

```
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., in JS functions are considered “branches”. <click> Thus we would annotate this code as: 1 assignment, 2 branches, and 2 conditions for an ABC metric of 3.

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/).

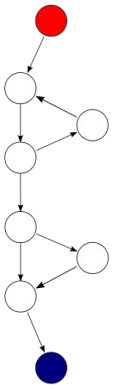
Quantitative: Cyclomatic complexity

Linearly-independent paths thru code

score = E – N + 2P

E edges, *N* nodes, *P* connected components

```
function myFuntion {
  while (...) {
    ....
  }
  if (...) {
    do_something
  }
}
```



E	9
N	8
P	1
CC	**3**

National Institute of Standards and Technology
U.S. Department of Commerce

Guidance: ≤10 per method

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

Quantitative: Metrics

Metric	Tool	Target score
Code-to-test ratio	Plato/Jest	≤ 1:2
C0 (statement) coverage	Jest	70%+
Assignment-Branch-Condition score	? for JS	< 20 per method
Cyclomatic complexity	Plato, ESLint	< 10 per method (NIST)

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…)

Refactoring

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

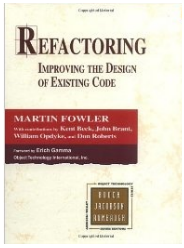
Which of the following is **not** a goal of method level refactoring?

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.

Refactoring has common patterns too



Fowler et al. created a catalog of common refactorings

Decompose Conditional

Name

You have a complicated conditional (if-then-else) statement.

When to use

Extract methods from the condition, then part, and else parts.

Mechanics

```
if (date.before (SUMMER_START) || date.after(SUMMER_END))
  charge = quantity * _winterRate + _winterServiceCharge;
else charge = quantity * _summerRate;
```

Example

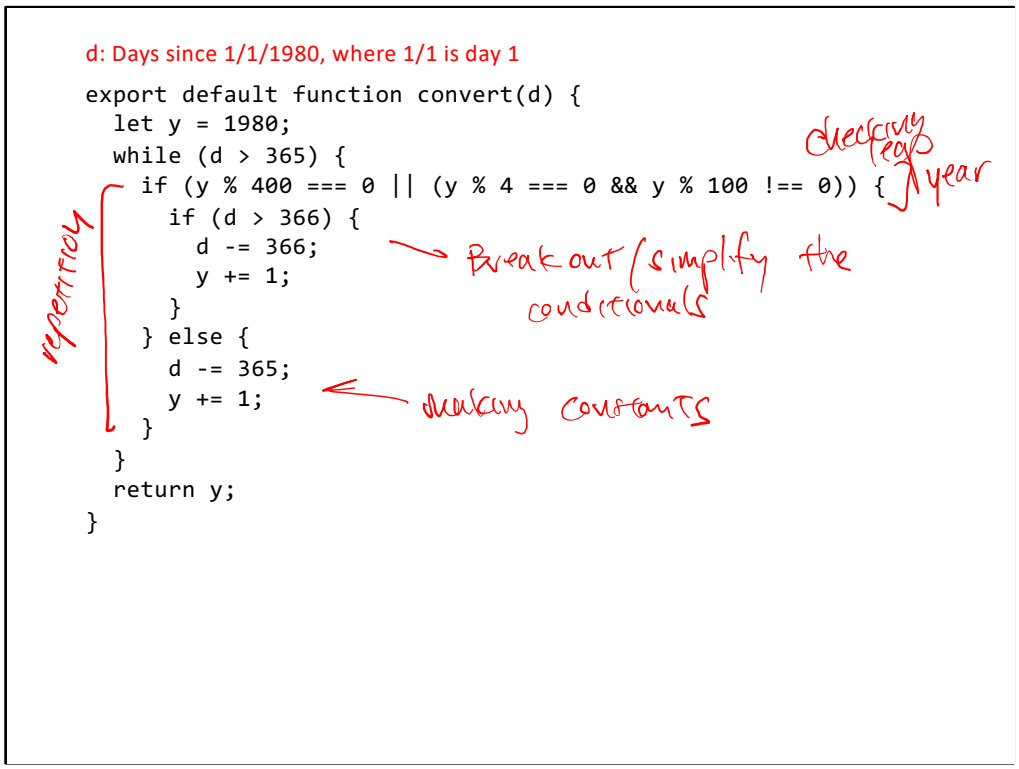
```
if (notSummer(date))
  charge = winterCharge(quantity);
else charge = summerCharge (quantity);
```

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

A blast from the past!

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



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).

let {default: convert} = await import ("./index.js")
convert(400)
convert(10593)

How do you want to tackle this refactoring/debugging?
1. Pull out leap year as a separate function to enable testing.

Recall the rules for leap years, There is a leap year every year whose number is perfectly divisible by four, except for years which are both divisible by 100 and not divisible by 400.

function isLeapYear(year) { return year % 400 === 0 || (year % 4 === 0 && year % 100 !== 0);}

2. We have a lot of duplication in the conditionals and subtraction. We would simplify

that to consolidate incrementing the year and checking and subtracting the days.

The key is to expose the bug where d is 366 and it's a leap year and thus the loop never ends (since d is > 365 but not decrementing).

// Intermediate

```
export function isLeapYear(year) {
  return (year % 400 === 0 || (year % 4 === 0 && year % 100 !==
0));
}

export default function convert(d) {
  let y = 1980;
  while (true) {
    const daysInYear = isLeapYear(y) ? 366 : 365;
    if (d <= daysInYear)
      break;
    d -= daysInYear;
    y += 1;
  }
  return y;
}
```

// Final version

```
export function isLeapYear(year) {
  return year % 400 === 0 || (year % 4 === 0 && year % 100 !==
0);
}

export function daysInYear(year) {
  return isLeapYear(year) ? 366 : 365;
}

export default function convert(days) {
  let year = 1980;
  let daysLeft = days;

  while (true) {
    const daysInCurrentYear = daysInYear(year);
    if (daysLeft <= daysInCurrentYear) {
      return year;
```

```
    }
    daysLeft -= daysInCurrentYear;
    year += 1;
  }
}
```

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 with tests, might have been prevented…

Improved code metrics

The initial version has a cyclomatic complexity of 6, that goes down to a little over 2 and the maintainability estimates (as reported by Plato increase)!

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 produce 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

Refactoring 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, online, etc.)
3. At each step, test newly-exposed *seams*, then stub/mock them out in higher-level tests

Example "smells" and their remedies

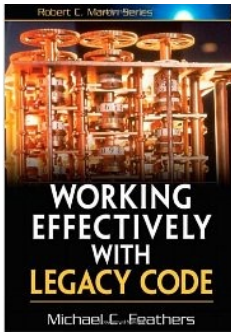
Smell	**Refactoring that may resolve it**
Large class	Extract class, subclass or module
Long method	Decompose conditional Replace loop with collection method Extract method Replace temp variable with query Replace method with object
Long parameter list/data clump	Replace parameter with method call Extract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

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



We are starting out projects, how could this be relevant? I suspect some these already apply to the code you wrote! A "long time ago" is not that long ago… like maybe last week!

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!

Feathers' two ways to approach modifying legacy code

Edit and Pray

1. Familiarize yourself with the relevant code
2. Plan the changes you will make
3. Make the planned changes
4. Poke around to make sure you didn't break anything

Cover and Modify

1. Write tests that cover the code you will modify (creating a "safety blanket")
2. Make the changes
3. Use tests to detect unintended effects

'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

What are those "right tools and techniques"? <click>

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", and the notion of strategic programming we discussed early on. We want to be making small investments throughput in improving our code base!

If you've been assigned to modify existing 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

E All of the above

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, likely 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…

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

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?

Characterization Tests

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

- Repeatable tests ensure current behaviors aren't changed (even if buggy)
- Integration tests are a natural starting point (b/c they are typically "black box")

Recall "Given-When-Then" tests

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

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
```

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 database or other similar infrastructure

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.

earthcam.com

The only people that can see the Statue of Liberty's hair are those that climb up the torch. Yet the artist(s) included hair anyway!

Our perspective can be more "glass half full" than this slide suggests. As Ousterhout notes in his book about software design, many aspect of good design slow you down in the beginning (create extra work). He writes "If the only thing that matters to you is making your current code work as soon as possible, then thinking about design will seem like drudge work that is getting in the way of your real goal." But …

"the investments you make in good design will pay off quickly. The modules you defined carefully at the beginning of a project will save you time later as you reuse them over and over. The clear documentation that you wrote six months ago will save you time when you return to the code to add a new feature[…] Good design doesn't really take much longer than quick-and-dirty design, once you know how."

"The reward for being a good designer is that you get to spend a larger fraction of your time in the design phase, which is fun. Poor designers spend most of their time chasing bugs in complicated and brittle code. If you improve your design skills, not only will you produce higher quality software more quickly, but the software development process will be more enjoyable."

Ousterhout, John K. . A Philosophy of Software Design, 2nd Edition (p. 176). Yaknyam

Press. Kindle Edition.