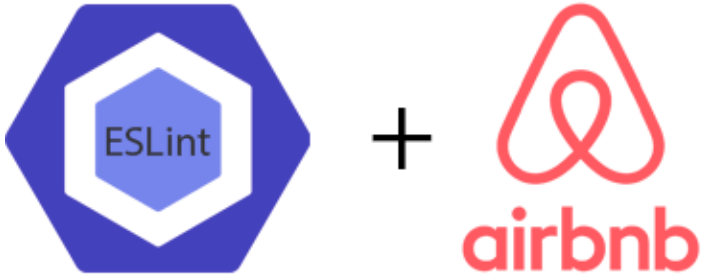
Beyond Correctness

Can we give feedback on software *beauty*?

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



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

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“);
  }
}
```

$$\sqrt{1 + 2^2 + 2^2} = 3$$

Guidance: ≤20 per method

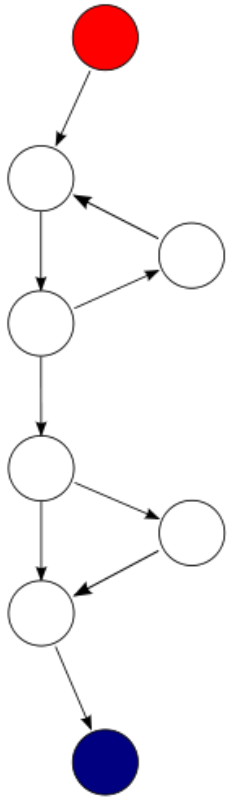
Quantitative: Cyclomatic complexity

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

Quantitative: Metrics

Metric	**Tool**	**Target score**
Code-to-test ratio	Plato/Jest	≤ 1:2
C0 (statement) coverage	Jest	70%+
Assignment-Branch-Condition score	None 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…)

Qualitative: “Code smells”

SOFA captures symptoms that often indicate code smells:

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

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

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

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”*

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

Which SOFA guideline is the **most** important for unit testing?

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

Other 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	Move method/move field to collect related items into one DRY place Aspect Oriented Programming
Too many comments	Extract method Introduce assertion Replace with internal documentation
Inconsistent level of abstraction	Extract methods & classes

Summary

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

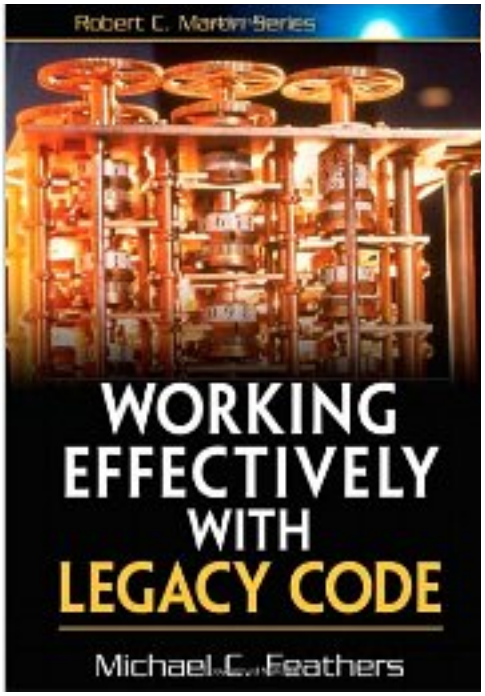
1. Use metrics as a guide to where you can improve your code
2. Apply *refactorings* (found on previous slide, in Refactoring books, on line, etc.)
3. At each step, test newly-exposed *seams*, then stub/mock them out in higher-level tests

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



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

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…

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

Exploring legacy codebases: 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

Exploring legacy codebases: 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, CRC cards

   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
```

(Bad) comments: The scourge of legacy code

Which of the following are useful comments?

```
// Add one to i.
i++;
```

```
// Lock to protect against concurrent access.
SpinLock mutex;
```

```
// This function swaps the panels.
void swap_panels(Panel* p1, Panel* p2) {...}
```

Good(?) comments…

```
// Loop through every array index, get the
// third value of the list in the content to
// determine if it has the symbol we are looking
// for. Set the result to the symbol if we
// find it.
```

What is wrong with the comment above?

Comments should raise the level of abstraction. Describe *why* the code was written this way, not *how*…

```
// Scan array of tuples to find query symbol if
// present
```

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!

First and foremost 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