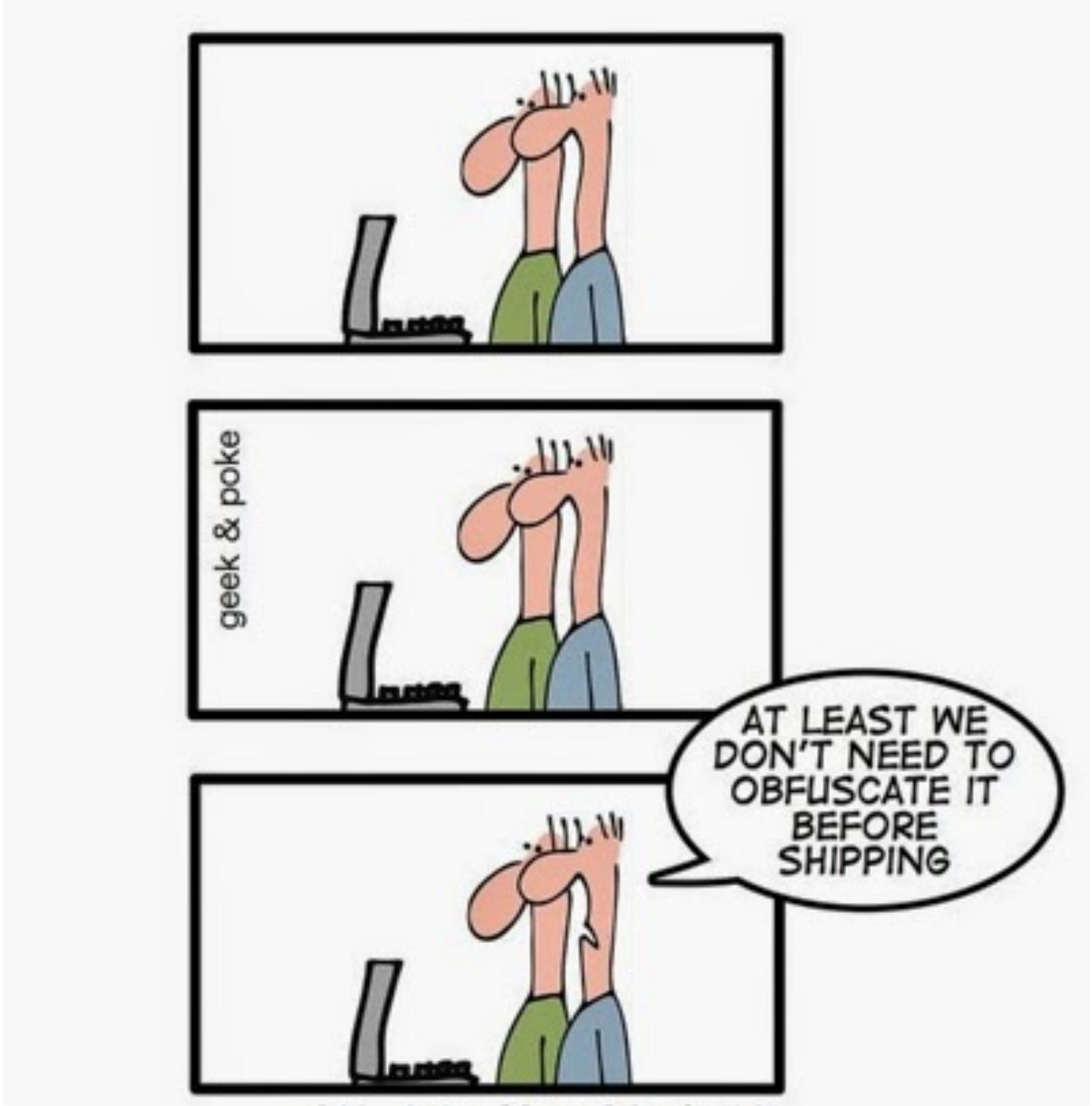
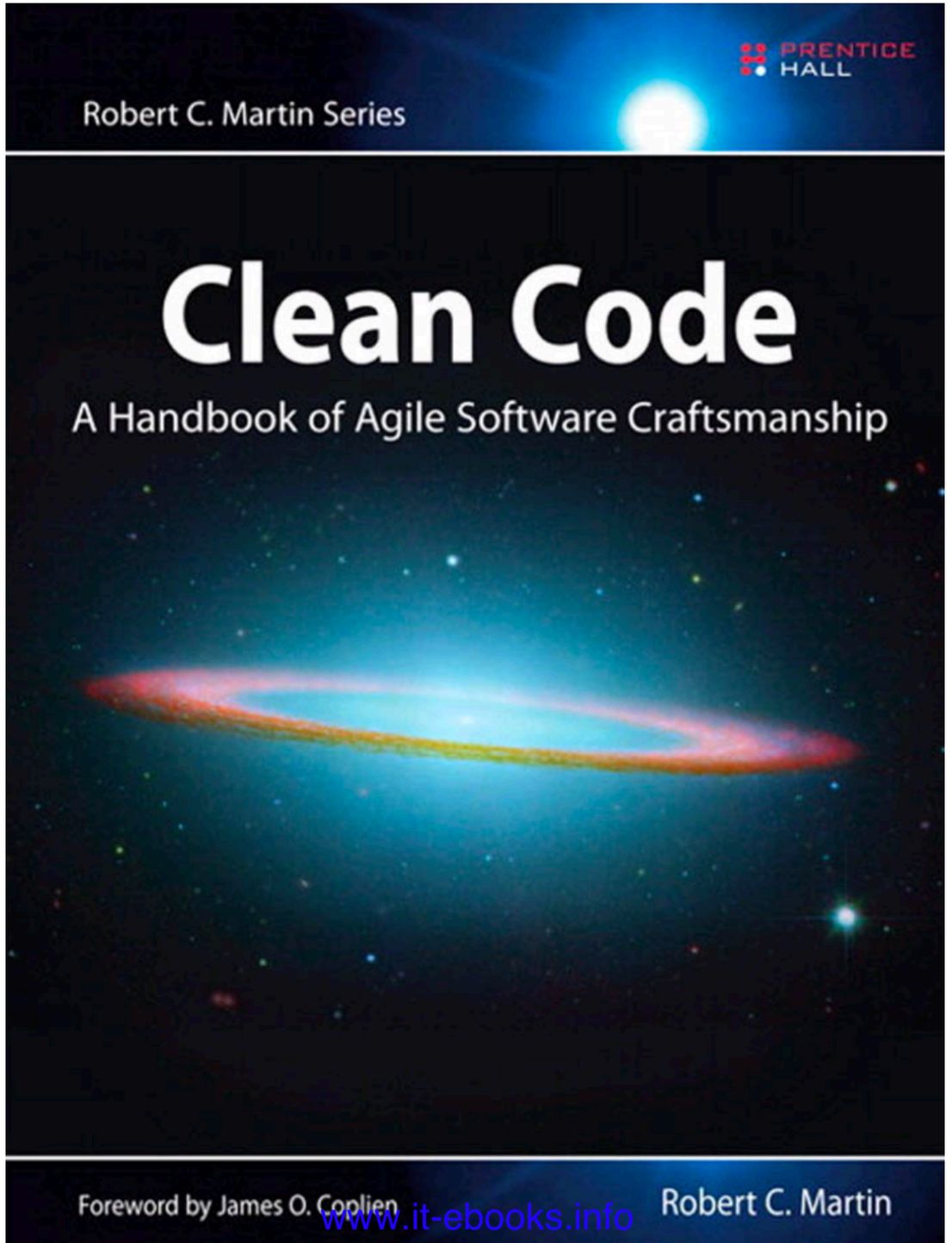
Workshop #5: **Clean Code.**

Dan Larremore
Assistant Professor
University of Colorado Boulder


geek & poke
AT LEAST WE DON'T NEED TO OBFUSCATE IT BEFORE SHIPPING



Coding is always collaborative.

often with others, but often with your future self.

Bad code is hard to read, maintain…

… modify, debug, translate, explain, etc.

Today: Writing code is like **writing anything else.**

The same content can be implemented with clarity or without.

Tips for writing clean code.

1. Organizing a project.
2. Naming variables.
3. Writing functions.

Tips for writing clean code.

1. Organizing a project.
2. Naming variables.
3. Writing functions.

First, folder organization

Many of us **clean and analyze data**, so keep separate:

raw_inputs to your project

cleaned_inputs that have been processed

code that processes data

output files post-analysis

figures generated from analysis

Second, use github

Even if your project is currently solo, using github allows it to become collaborative with a click, plus it enables version control and branches.

Third, prioritize clarity and stability

In organization (just like naming variables and writing functions) we want things to be intuitive and understandable.

Analogy: reading and writing scientific papers. There is no absolutely “correct” way to write a paper, but some are easier to understand than others. *This comes after revision, not first draft!*

Tips for writing clean code.

1. Organizing a project.
2. Naming variables.
3. Writing functions.

1 pick a style and stick with it

camelCaseVariable or snake_case_variable

CapitalizedClass

2 booleans answer questions

isParsed
hasPublications

This improves readability. Consider:

```
if hasPublications:
    getPubs(pubSection)
```

Contrast with:

```
if publications:
    getPubs(pubSection)
```

3 names reveal intentions

The name of a variable, function, or class, should tell you why it exists, what it does, and how it is used.

If a name requires a comment, then the name does not reveal its intent.

```
int d; // elapsed time in days
```

Instead, write

```
int elapsedTimeInDays;
```

4 avoid disinformation

Avoid using system or language keywords, as well as standard package names. For instance:

```
np = 50
```

Don’t mix colloquial and coding words. For instance:

```
publicationList
```

had better be of type *List.* Better yet, avoid encoding container types in variable names. Why not just:

```
publications
```

5 beware small variations

How long does it take to spot the difference between

```
XYZControllerForEfficientHandlingOfStrings
```

and

```
XYZControllerForEfficientStorageOfStrings
```

Again, just like writing prose, we should always be asking what can improve clarity and readability.

Do not fear (and instead embrace) the rewrite!

6 just don’t

lower-case L

```
l
```

upper-case o

```
O
```

consider the effect on readability:

```
int a = l;
if ( O == l )
    a=O1;
else
    l=01;
```

7 use pronounceable names

```
class DtaRcrd102 {
    private Date genymdhms;
    private Date modymdhms;
    private final String pszqint = "102";
    /* ... */
};
```

```
class Customer {
    private Date generationTimestamp;
    private Date modificationTimestamp;
    private final String recordId = "102";
/* ... */
};
```

We pronounce our variables out loud, or in our heads while reading. Make it easy!

8 use searchable names

For instance, don't name a variable **sum** since you may use **np.sum(…)** elsewhere.

And, consider how hard it would be to find instance of variable **robot** if you also have named variables **robots**, **robotHouse** and **robotFactory**

This is starting to get complicated. Just like writing prose, don't expect code to be perfect after the first draft, but *do* edit for clarity!

Note: single-letter names should *only* be used as local variables inside short methods.

Note: **i j k** are traditional counters. No problem.

9 classes are nouns, methods are verbs

```
Customer
WikiPage
Account
AddressParser
```

But again, for clarity, avoid words in classes like:

```
Manager
Processor
Data
Info
```

```
cvitae.getSectionBreaks()
cvitae.setAuthorNickname('Ralphie McQuabberberg')
cvsection.removeExtraHTMLTags()
publication.isParsed()
```

10 don’t change a variable’s type

don’t change horses midstream etc.

```
cvitae.getSectionBreaks()
cvitae.setAuthorNickname(‘Ralphie McQuabberberg’)
cvsection.removeExtraHTMLTags()
publication.isParsed()
```

11 python: pep8

ask tcy re above

linters are a thing too.
suyog says: set up your config file and use autoformatters. A cleanerupper.

```
cvitae.getSectionBreaks()
cvitae.setAuthorNickname('Ralphie McQuabberberg')
cvsection.removeExtraHTMLTags()
publication.isParsed()
```

Tips for writing clean code.

1. Organizing a project.
2. Naming variables.
3. Writing functions.

1 functions should be *small*

The first rule of functions is that they should be small. The second rule of functions is that *they should be smaller than that.*

In the old days, the rule of thumb was that a function should fit on a single screen without scrolling. That was when screens were 24 lines by 80 columns, and editors used 4 lines for administrative purposes.

Still, today, functions should hardly ever be 20 lines long.

Why? Again, clarity is our rule: each function should tell a story that is easy to understand and transparently obvious.

Placed in sequence, functions should tell a story in a compelling order.

2 do one thing

Functions should do one thing. They should do it well. They should do it only.

The LOGO language used the keyword TO in the same way that Ruby and Python use def. So every function began with the word TO. This had an interesting effect on the way functions were designed!

The **TO paragraph** is the equivalent of a topic sentence:

TO RenderPageWithSetupsAndTeardowns, we check to see whether the page is a test page and if so, we include the setups and teardowns. In either case we render the page in HTML.

Does this function “do one thing” ?

3 one level of abstraction per function

One way to see if a function is going to do one thing is to see if statements are all a the same level of abstraction.

```
getHtml()
String pagePathName = PathParser.render(pagePath);
quab.append(‘\n’)
```

Each of the above is at a different level of abstraction, from high to low. These should not be in the same function. If they are, it is a hint that the function doesn’t do one thing.

4 reading code top-to-bottom

We want the code to read like a top-down narrative

Put differently, we want to be able to read the program as though it were a set of TO paragraphs, each of which is **describing the current level of abstraction** and then **referencing subsequent TO paragraphs** at the next level down.

```
TO parse a CV, we convert the CV to html, then we segment the html, then we profit from the segments.
    TO convert a CV to html, we generate raw html with an apache function, and then we clean the raw html.
        TO generate raw html with an apache function…
        TO clean the raw html…
    TO segment the html…
```

Note: staying at one level of abstraction is hard.

5 use descriptive names

A long descriptive name is better than a short enigmatic name.

Corollary: A long descriptive name is better than a long descriptive comment.

Choosing a descriptive name will clarify the design of your code *in your mind* by forcing you to think about it. It is not uncommon that hunting for a good name results in a favorable restructuring of the code.

Be consistent with your names.

6 minimize function arguments

The ideal number of arguments is zero (niladic). Then one (monadic), then two (dyadic). Three-argument functions (triaic) should be avoided if possible.

Why? Any function with an argument needs to be tested for robustness to that argument, to make sure the function works as designed. The more inputs, the more combinations of those inputs that will need to be tested.

A function with zero arguments is therefore trivial to test. On the other hand, to robustly test a function with many arguments is daunting.

7 flag arguments

Flag arguments are ugly. Passing a boolean into a function is bad practice. It immediately complicates the signature of the method, loudly proclaiming that this function does more than one thing!

Solution: make two functions.

Reasonable Objection: seriously??

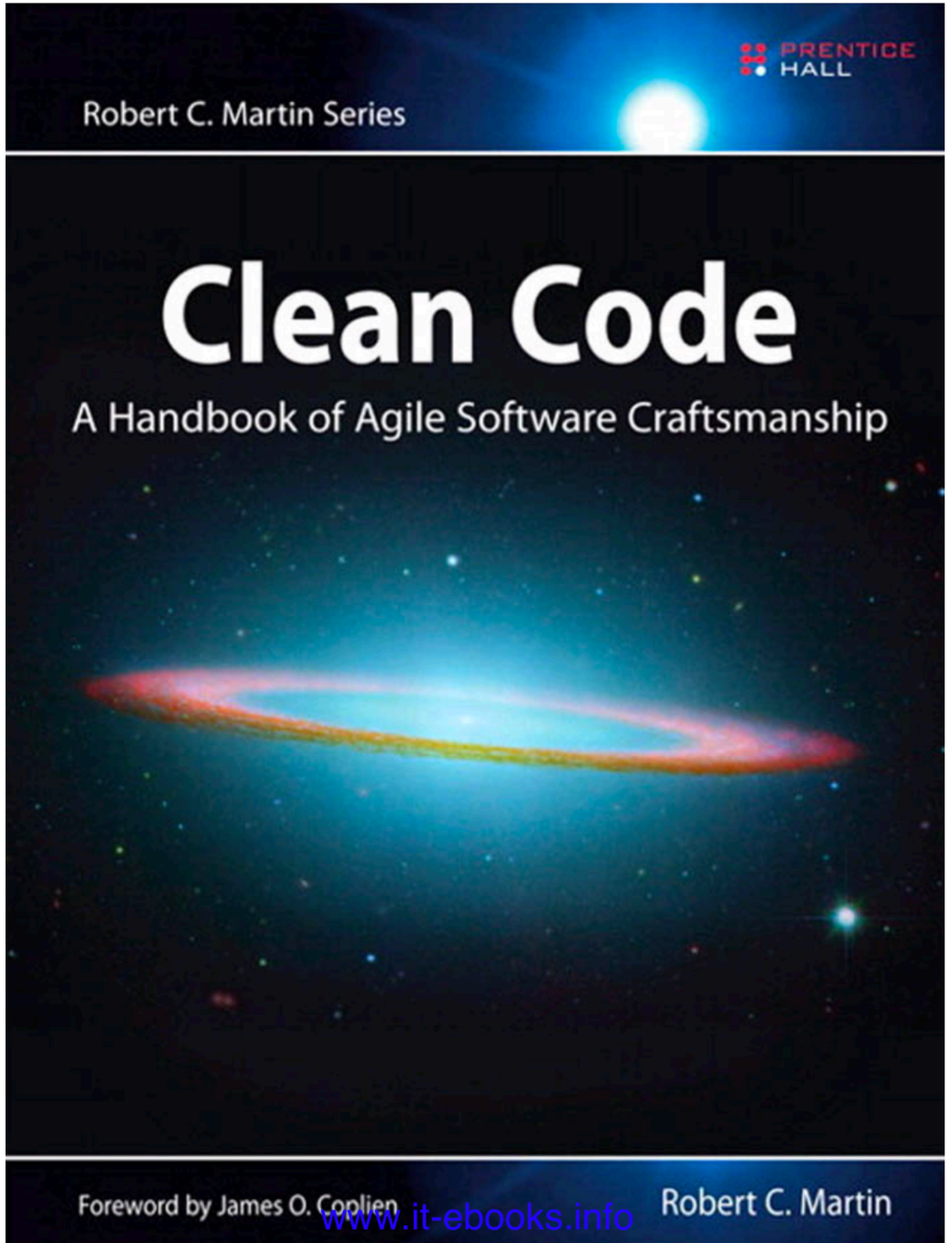
Answer: Yes. We aren't mathematicians trying to fit a proof in a single, complicated, "elegant" line. We want to make code as readable as possible.

8 verb your nouns

If functions are verbs, and arguments are nouns, try to make the verbs and nouns “read well” together.

```
writeField(name);
assertFirstEqualsLast(first,last);
```

```
field(x);
assertEquals(x,y);
```



Workshop #5: **Clean Code.**

Dan Larremore
Assistant Professor
University of Colorado Boulder

Dan Larremore

Assistant Professor
University of Colorado Boulder

More from this series in PDF at http://danlarremore.com

Workshop #3: **Data Visualization.**

Workshop #4: **Giving a Talk.**

Workshop #5: **Clean Code.**

Workshop #6: **Peer Review.**