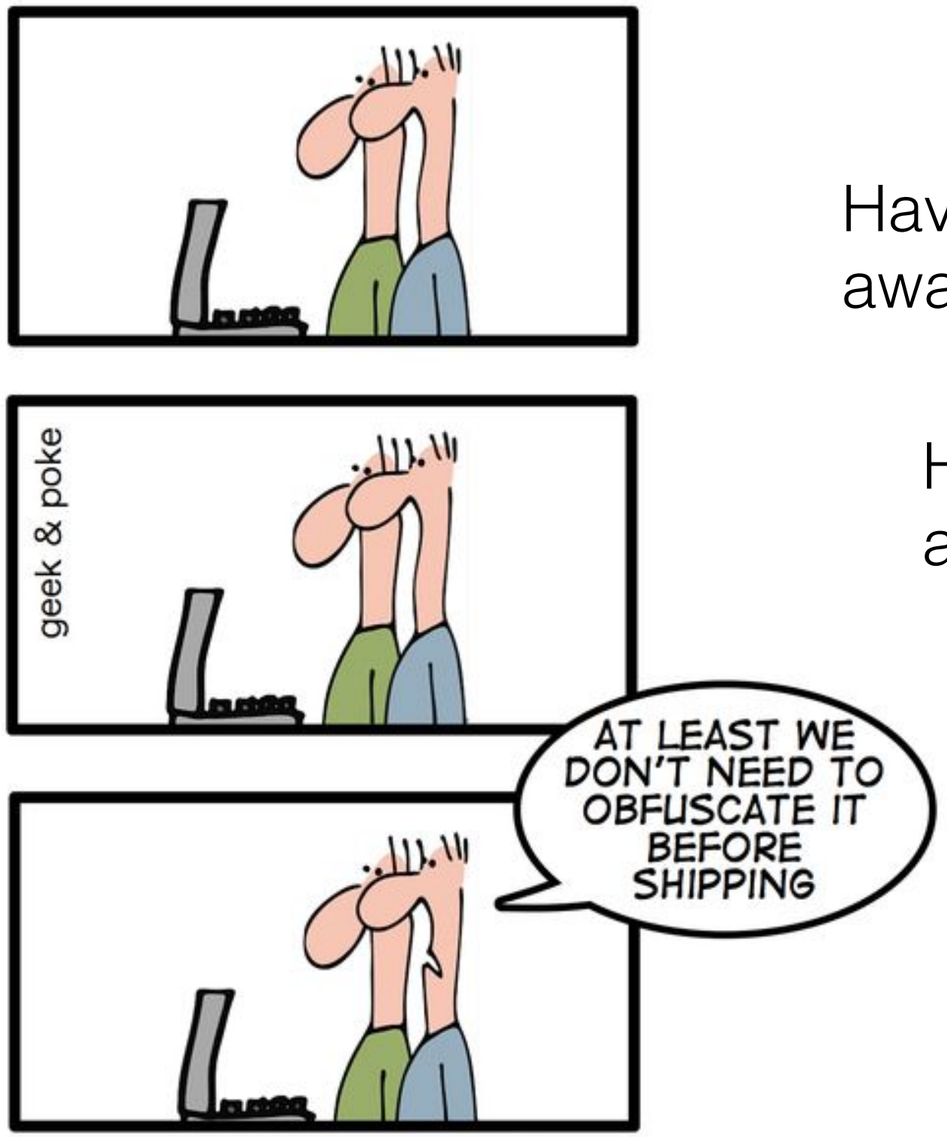
Workshop #5: **Clean Code.**

Dan Larremore
Assistant Professor
University of Colorado Boulder



Have you returned to a coding project after a month away, only to be **confused by your past work**?

Have you asked for the code from a **cool new paper** and received a confusing pile of **unreadable garbage**?

Have you filled your code with **`print(variable)`** statements trying to diagnose a weird bug?

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 & writing functions.
3. Test-driven development.

Tips for writing clean code.

1. Organizing a project.
2. Naming variables & writing functions.
3. Test-driven development.

First, organize your project

Many of us **clean and analyze data**, so make folders.

For example, my typical projects look like this:

- **raw_inputs** to your project.
- **cleaned_inputs** that have been processed.
- **api** for the code itself.
- **output** files post-analysis.
- **figures** generated from analysis.
- **writing** for the notes that I'll turn into a paper.

Recommendation: Think about organization *before* coding.

Second, use github

Even if your project is currently solo, using **git[hub]** allows it to become collaborative with a click, and it enables **version control** and **branches**.

Translations:

git is a version control system.

github is a cloud-based hosting service that plays nice with git.

version control includes the ability to rewind your changes to undo your mistakes

branches are different parallel versions of your code.

[e.g. one branch “works”; another branch is where you’re halfway through rewriting the assembly pipeline; a third branch has some experiments with new ways of plotting your figures.]

Third, use the README.md

Your README.md should include three things:

1. An abstract.
   - What's the high level question?
   - What languages are used?
   - What are the data sources?
2. Usage
   - How to install.
   - Requirements.
   - How to run.
3. Worked Examples
   - At least one callable copy&paste-able example.

For example:

A+ : giggle (Layer)

A+ : webweb (Larremore & Wapman)

C : BESTest (Larremore)

Fourth, prioritize clarity and stability

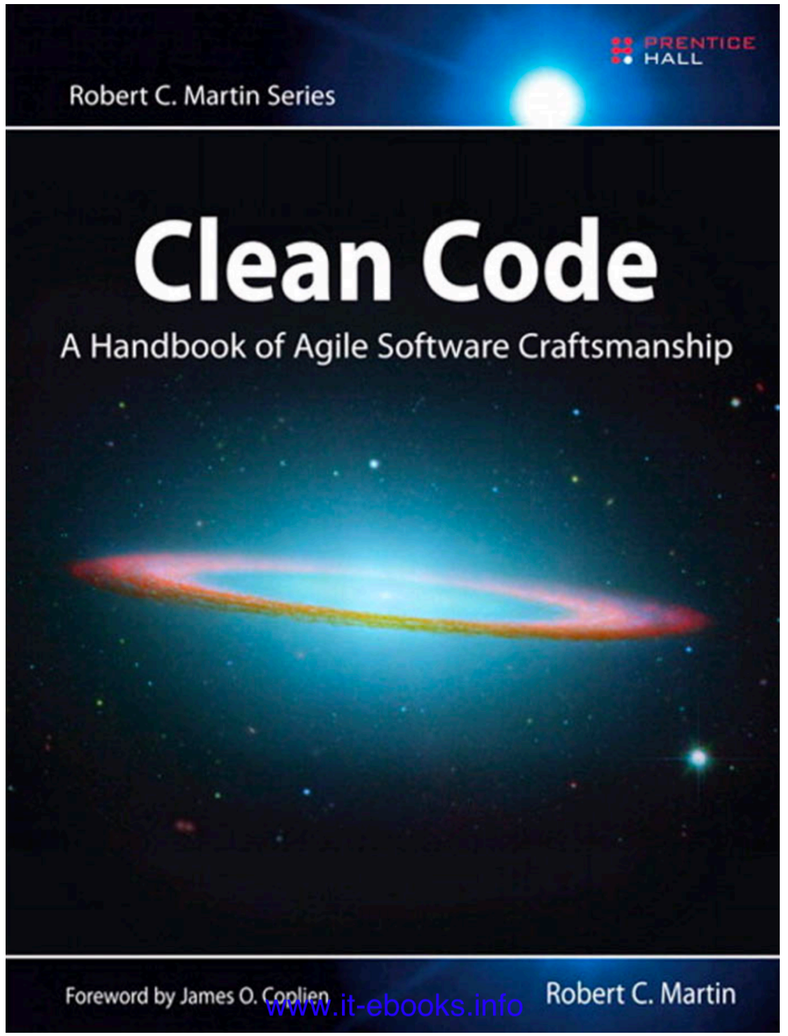
Organization goal: intuitive & understandable.

Analogy: writing scientific papers.
There is no *correct* way to write a paper, but some papers sure are easier to understand than others!

And by extension: readability comes after revisions, not on draft #1.
Embrace the editing and rewriting process.

Tips for writing clean code.

1. Organizing a project.
2. Naming variables
   & writing functions.
3. Test-driven development.



1. Pick a style and stick with it

`camelCaseVariable`	[typical java style]
`snake_case_variable`	[typical-ish python style]
`CapitalizedClass`	[generally common]
`CAPSLOCK_CONSTANT`	[generally common]

2. Booleans answer questions

```
is_parsed
has_publications
```

This improves readability. Consider:

```
if has_publications:
    get_pubs(cv_pubs_section)
```

Contrast with:

```
if publications:
    get_pubs(cv_pubs_section)
```

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.

```
d = 5 # elapsed time in days
```

Instead, write

```
elapsed_time_in_days = 5
```

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:

```
publication_list
```

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

```
publications
```

5. Beware small variations

It takes a long time to spot the difference between

```
XYZControllerForEfficientHandlingOfStrings
```

and

```
XYZControllerForEfficientStorageOfStrings
```

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

Recommendation: Don’t fear the rewrite! Embrace it!

6. Just don’t

lower-case L looks like 1

```
l 1
```

upper-case o looks like 0

```
O 0
```

consider the effect on readability:

```
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 an instance of variable **`robot`** if you also have named variables **`robots`**, **`robotHouse`** and **`robotFactory`.**

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

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

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!

9. Classes are nouns; methods are verbs

```
Customer
WikiPage
Account
AddressParser
```

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

```
Manager
Processor
Data
Info
```

Can you guess what each line here might do without seeing the code?

```
cvitae.get_section_breaks()
cvitae.set_author_nickname('Ralphie McBuffaloface')
cvsection.detect_ordered_lists()
publication.is_parsed()
```

10. Don’t change a variable’s type

For instance, if you’ve defined

```
current_square = 0
```

on one line, never change the variable’s type

```
current_square = ‘jail’
```

on a later line.

Languages like Python allow you do this sort of thing, but please don’t!

Variables Summary:

1. Pick a style and stick with it.
2. Booleans answer questions.
3. Variable names reveal intentions.
4. Avoid disinformation.
5. Beware small variations.
6. Just don't.
7. Make names pronounceable.
8. Make names searchable.
9. Classes are nouns. Methods are verbs.
10. Don't change a variable's type.

```
snake_case, camelCase
is_parsed, has_dependencies
elapsed_time_in_days
np=5, sum=12, plot=True
seqReversedMap, seqReverserMap
l 1 O 0
alscr, alignment_score
```

…and find-&-replaceable!

Level up: use a linter

Languages have style guides.
Programming languages do too!

Code whose purpose is to check *other* code for style.

That's called a *linter*. Python's: github.com/PyCQA/pycodestyle

```
$ pycodestyle --first optparse.py
optparse.py:69:11: E401 multiple imports on one line
optparse.py:77:1: E302 expected 2 blank lines, found 1
optparse.py:88:5: E301 expected 1 blank line, found 0
optparse.py:222:34: W602 deprecated form of raising exception
optparse.py:347:31: E211 whitespace before '('
optparse.py:357:17: E201 whitespace after '{'
optparse.py:472:29: E221 multiple spaces before operator
optparse.py:544:21: W601 .has_key() is deprecated, use 'in'
```

Note: pycodestyle used to be called pep8. Guess why they renamed it…

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.

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. Can you read your code, top to bottom? How does the story sound?

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 clean_and_parse_the_cvitae: we check to see if there
are headers or footers and if so, we remove these. In
either case, we parse the cv's sections.
```

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 at the same level of abstraction.

```
cvitae = cv_importer(path)
sections_to_parse = cvitae.get_section_breaks()
output_string.append("Begin Parse\n")
```

- Each line 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 learn 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…
        TO…
    TO learn from the segments…

Observation: 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 (triadic) 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 in functions

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. One for `has_flag=True` and one for `False`.

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.

```
write_field(name);
assert_first_equals_last(first,last);
```

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

Functions Summary:

1. Functions should be small.
2. Do one thing. (use the TO statement)
3. One level of abstraction per function.
4. Code should be top-to-bottom readable.
5. Use descriptive names. *Functions are like variables*.
6. Minimize function arguments.
7. Avoid flag arguments—write two functions.
8. Verb your nouns.

Yes but I write “scripts”

Rules apply to scripts too!

Don’t let this happen to you:

How a study based on a typo made news everywhere — and the retraction didn’t

A paper arguing religious kids are less generous turned out to be the result of a typo.

By Kelsey Piper | Oct 3, 2019, 8:10am EDT

The glitch caused results of a common chemistry computation to vary depending on the operating system used, causing discrepancies among Mac, Windows, and Linux systems. The researchers published the revelation and a debugged version of the script, which amounts to roughly 1,000 lines of code, on Tuesday in the journal *Organic Letters*.

MOTHERBOARD
TECH BY VICE

A Code Glitch May Have Caused Errors In More Than 100 Published Studies

The answer? The entire result was literally the product of a typo. The researchers had collected data in many different countries — the US, Canada, China, Jordan, Turkey, and South Africa. When coding in their results, they used numbers to represent each country — 1 for the US, 2 for Canada, and so on. Then, they tried to control for the country in evaluating their results. But instead of controlling for country, Psychology Today reports, they “just **treated it as a single continuous variable** so that, for example “Canada” (coded as 2) was twice the “United States” (coded as 1).”

Bloomberg Businessweek

April 18, 2013, 4:31 AM MDT

FAQ: Reinhart, Rogoff, and the Excel Error That Changed History

Harvard University economists Carmen Reinhart and Kenneth Rogoff have acknowledged making a spreadsheet calculation mistake in a 2010 research paper, “Growth in a Time of Debt” (PDF), which has been widely cited to justify budget-cutting. But the

https://www.vox.com/future-perfect/2019/10/3/20895240/study-typo-religion-children-generosity-retraction
https://www.vice.com/en_us/article/zmjwda/a-code-glitch-may-have-caused-errors-in-more-than-100-published-studies
https://www.bloomberg.com/news/articles/2013-04-18/faq-reinhart-rogoff-and-the-excel-error-that-changed-history

Tips for writing clean code.

1. Organizing a project.
2. Naming variables & writing functions.
3. Test-driven development.

The logic of writing tests

Here's a function. It appears to take a string and return every other letter.

```
def get_every_other_letter(my_string):
    t = [ my_string[idx] for idx in range(0,2,len(my_string)) ]
    return ''.join(t)
```

A test function simply confronts the function with an input and its expected output.

```
def test_get_every_other_letter():
    input_1  = 'PandaRobotOctopus'
    output_1 = 'PnaooOtps'
    assert(get_every_other_letter(input_1) == output_1)
```

Our test fails! This tells us that we need to make changes.

```
def get_every_other_letter(my_string):
    t = [ my_string[idx] for idx in range(0,len(my_string),2) ]
    return ''.join(t)
```

The logic of writing tests

Writing tests is how we know that our functions work.

If we have a large set of tests, we can now easily roll out new features:

A new feature can be released/pushed/merged ONLY IF

1. The tests for the new feature pass.
2. All the previously existing tests also pass.
   [this ensures we didn't break something]

Reasonable question: how complicated should my test be?

Reasonable answer: how complicated is your function?

Tests provide strong incentives:

2. Functions should do one thing.
6. Minimize function arguments.

The logic of writing tests first

When we write a function, we have a task or goal. The function *does* something.

If we can articulate the goal, we can actually **start** by writing out a test!

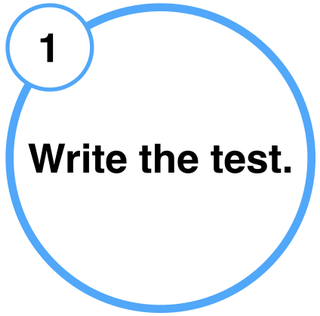
```
def test_get_every_other_letter():
    input_1  = 'PandaRobotOctopus'
    output_1 = 'PnaooOtps'
    assert(get_every_other_letter(input_1) == output_1)
```

Writing the test brings clarity, forcing us to be specific about the function's task.

Imagine: write a test, and *then* write code till the test passes…

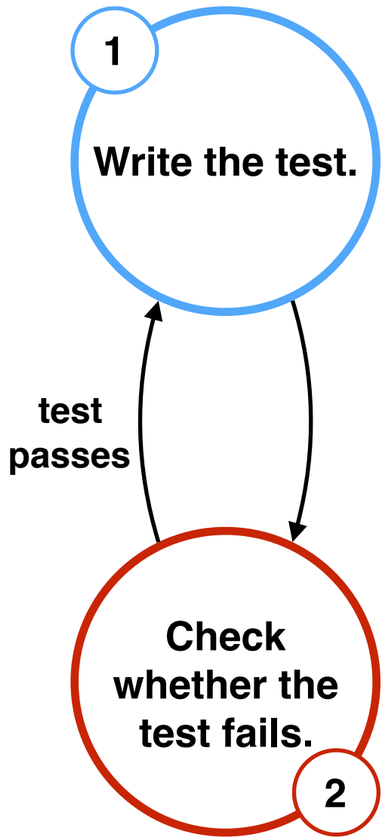
Test-Driven Development (bowling with bumpers)

1. **Write the test** for the new feature. This forces us to consider requirements *before* writing code.



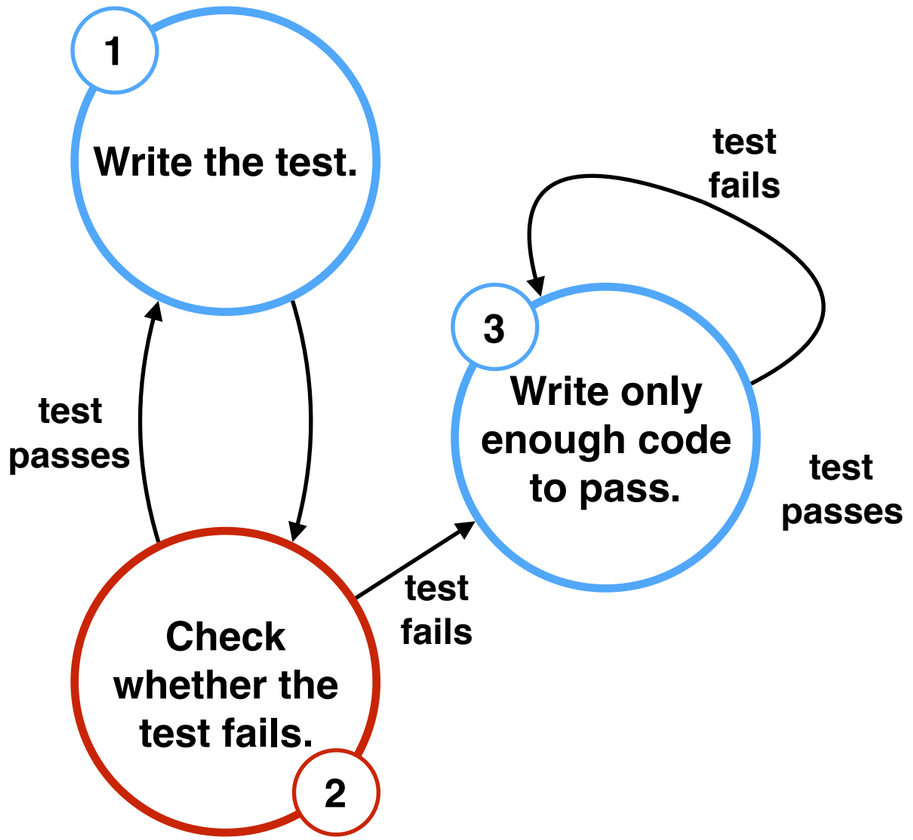
Test-Driven Development (bowling with bumpers)

1. **Write the test** for the new feature. This forces us to consider requirements *before* writing code.
2. Run all tests to **make sure that the new test fails** without the new feature.



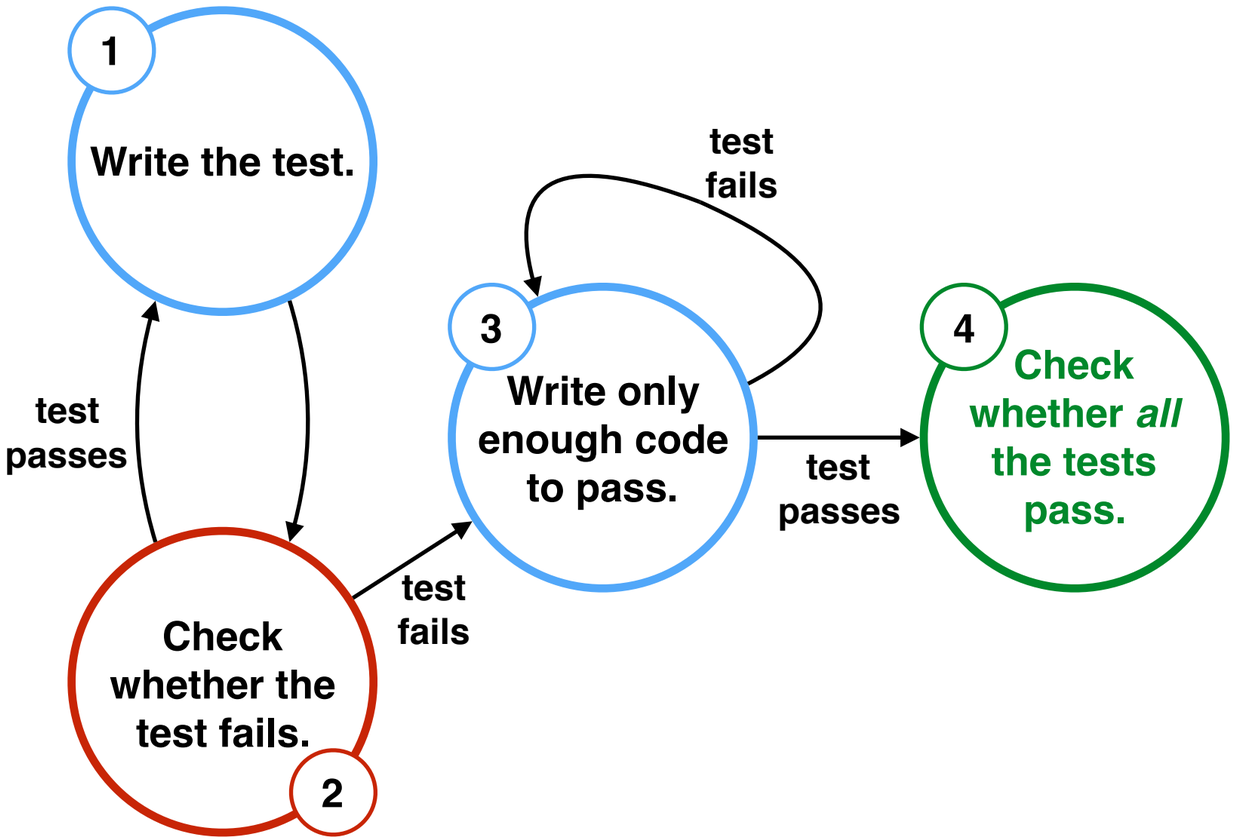
Test-Driven Development (bowling with bumpers)

1. **Write the test** for the new feature. This forces us to consider requirements *before* writing code.
2. Run all tests to **make sure that the new test fails** without the new feature.
3. Write **minimal code** to pass the test. Don't focus on writing perfect code. Focus on style and getting the job done. No extras! Pass the new test before continuing.



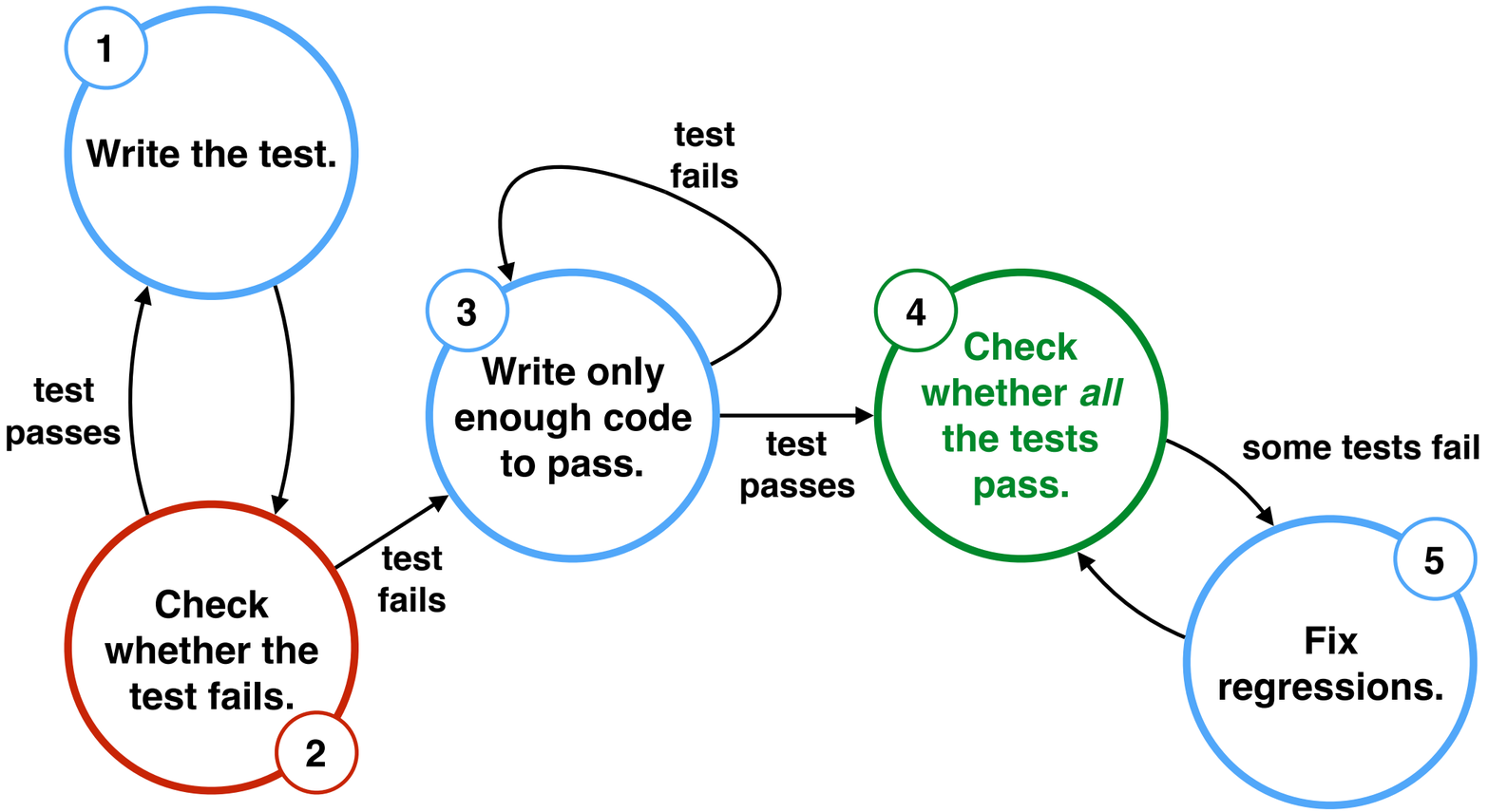
Test-Driven Development (bowling with bumpers)

1. **Write the test** for the new feature. This forces us to consider requirements *before* writing code.
2. Run all tests to **make sure that the new test fails** without the new feature.
3. Write **minimal code** to pass the test. Don't focus on writing perfect code. Focus on style and getting the job done. No extras! Pass the new test before continuing.
4. **Run all existing tests.**



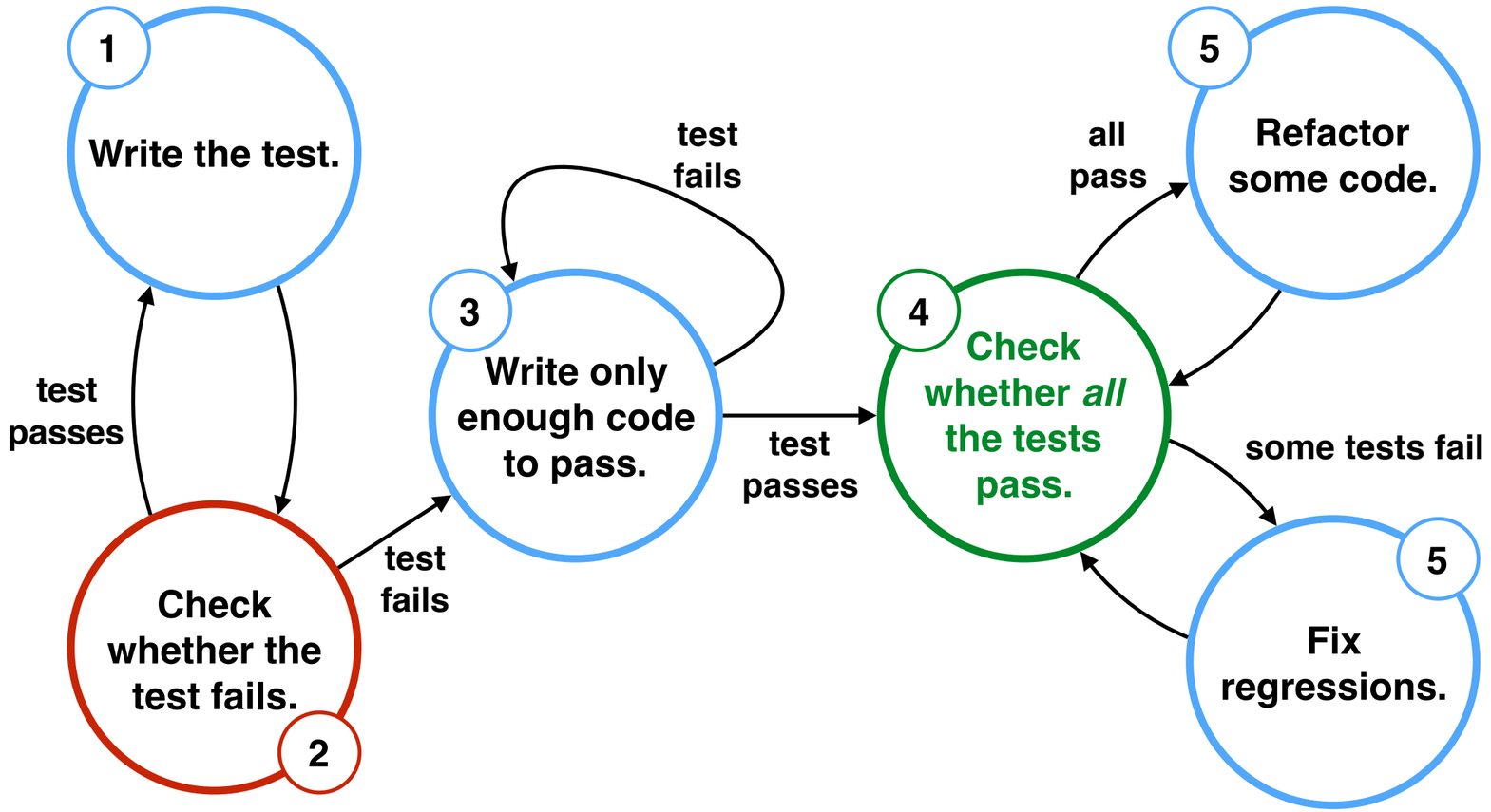
Test-Driven Development (bowling with bumpers)

1. **Write the test** for the new feature. This forces us to consider requirements *before* writing code.
2. Run all tests to **make sure that the new test fails** without the new feature.
3. Write **minimal code** to pass the test. Don't focus on writing perfect code. Focus on style and getting the job done. No extras! Pass the new test before continuing.
4. **Run all existing tests.**
5. If the new code breaks an existing test, **fix "regressions."** Note that your old tests prevent you from breaking something old with your new feature. *This is key.*



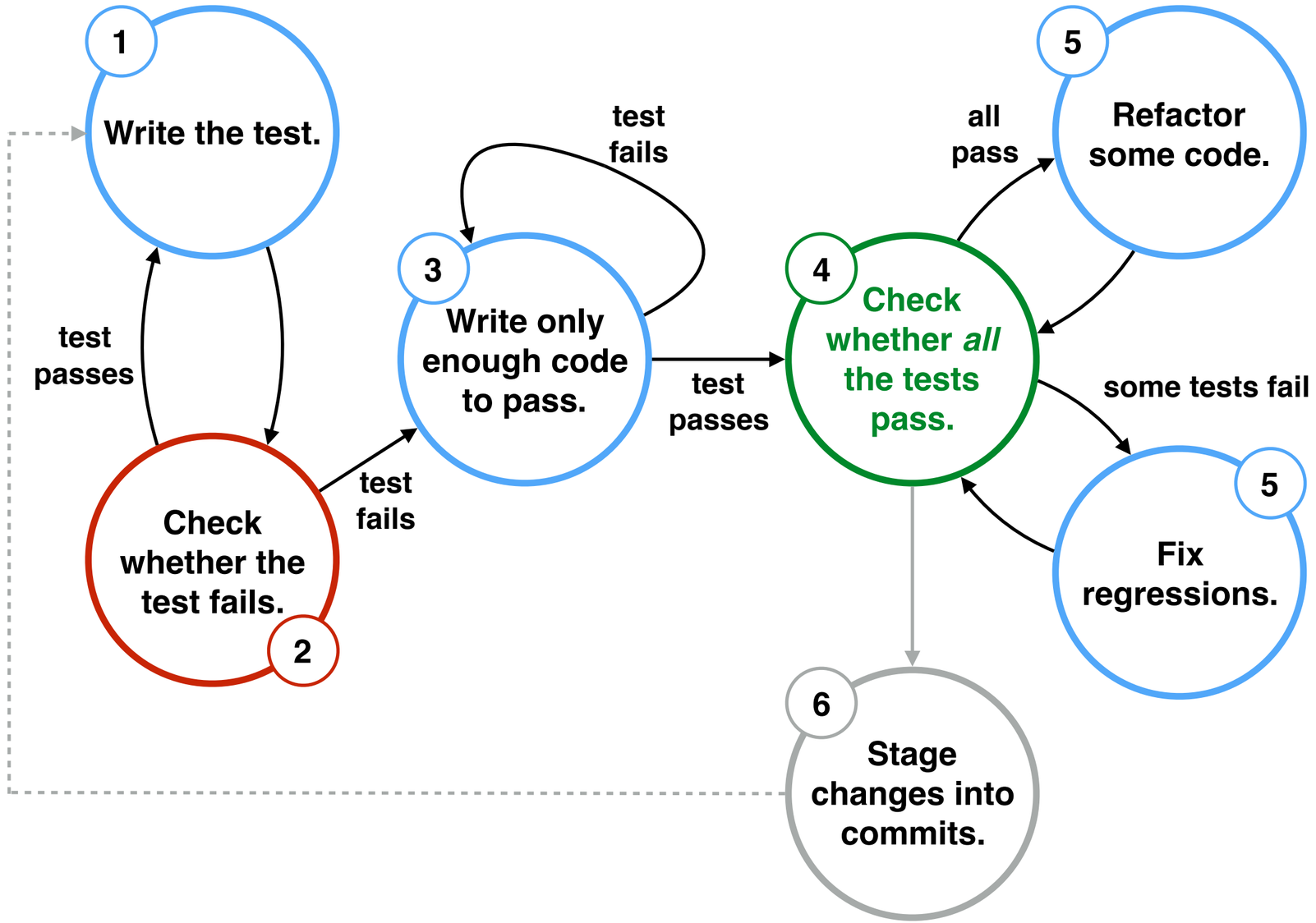
Test-Driven Development (bowling with bumpers)

1. **Write the test** for the new feature. This forces us to consider requirements *before* writing code.
2. Run all tests to **make sure that the new test fails** without the new feature.
3. Write **minimal code** to pass the test. Don't focus on writing perfect code. Focus on style and getting the job done. No extras! Pass the new test before continuing.
4. **Run all existing tests.**
5. If the new code breaks an existing test, **fix "regressions."** Note that your old tests prevent you from breaking something old with your new feature. *This is key*.
5. Only if the tests are passing, then go back and **refactor (rewrite)** ugly passable code and make it shine.



Test-Driven Development (bowling with bumpers)

1. **Write the test** for the new feature. This forces us to consider requirements *before* writing code.
2. Run all tests to **make sure that the new test fails** without the new feature.
3. Write **minimal code** to pass the test. Don't focus on writing perfect code. Focus on style and getting the job done. No extras! Pass the new test before continuing.
4. **Run all existing tests.**
5. If the new code breaks an existing test, **fix "regressions."** Note that your old tests prevent you from breaking something old with your new feature. *This is key.*
5. Only if the tests are passing, then go back and **refactor (rewrite)** ugly passable code and make it shine.
6. Stage your changes into **commits**. Then **go back to 1** and add another new test.



Workshop #5: **Clean Code.**

Variables

1. Pick a style and stick with it—use a linter.
2. Booleans answer questions.
3. Variable names reveal intentions.
4. Avoid disinformation.
5. Beware small variations.
6. Just don't.
7. Make names pronounceable.
8. Make names searchable.
9. Classes are nouns. Methods are verbs.
10. Don't change a variable's type.

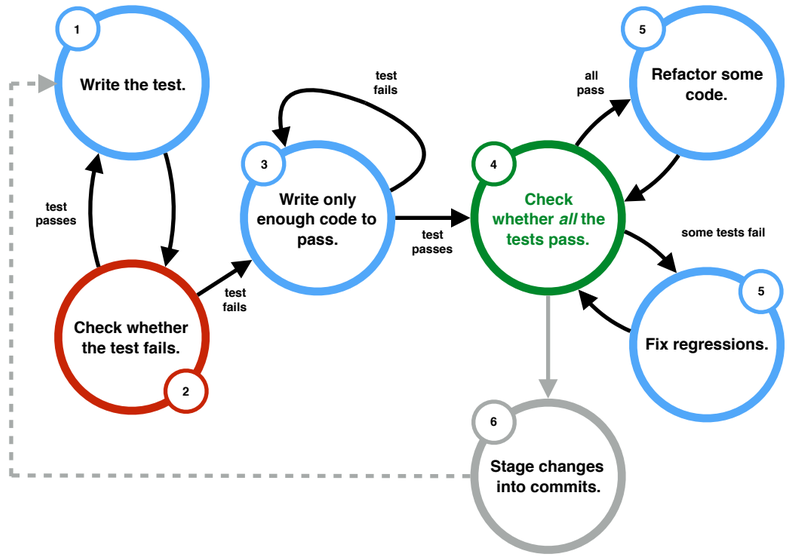
Functions

1. Functions should be small.
2. Do one thing. (use the TO statement)
3. One level of abstraction per function.
4. Code should be top-to-bottom readable.
5. Use descriptive names. *Functions are like variables*.
6. Minimize function arguments.
7. Avoid flag arguments—write two functions.
8. Verb your nouns.

Project-Level Planning

1. Organize your project.
2. Use github.
3. Use the README.md
4. Prioritize clarity and stability. Don't fear the rewrite.

Write tests. Even better, use Test-Driven Development

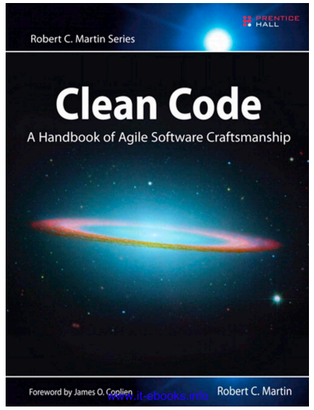


Workshop #5: **Clean Code.**

Dan Larremore
Assistant Professor
University of Colorado Boulder

Workshop #5: **Clean Code.**

The **vast majority** of the ideas, concepts, and revisions to this talk come from others:



Robert Martin
Agile Manifesto
SOLID principles

Ryan Layer
Assistant Professor
Computer Science
& BioFrontiers
CU Boulder

Sam Zhang
Pivotal Labs, OpenCounter
Now: PhD Student
Applied Math
CU Boulder

Dan Larremore

Assistant Professor
University of Colorado Boulder

Dan Larremore

Assistant Professor
University of Colorado Boulder

More from this series in PDF at LarremoreLab.github.io

Workshop #3: **Data Visualization.**

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

Workshop #5: **Clean Code.**

Workshop #6: **Peer Review.**