EXASCALE COMPUTING PROJECT

IDEAS productivity

better scientific software

Students@SC: Modern Software Design, Tools, and Practices

Elsa Gonsiorowski
Lawrence Livermore National Laboratory
gonsie@llnl.gov

Reed Milewicz
Sandia National Laboratories
rmilewi@sandia.gov

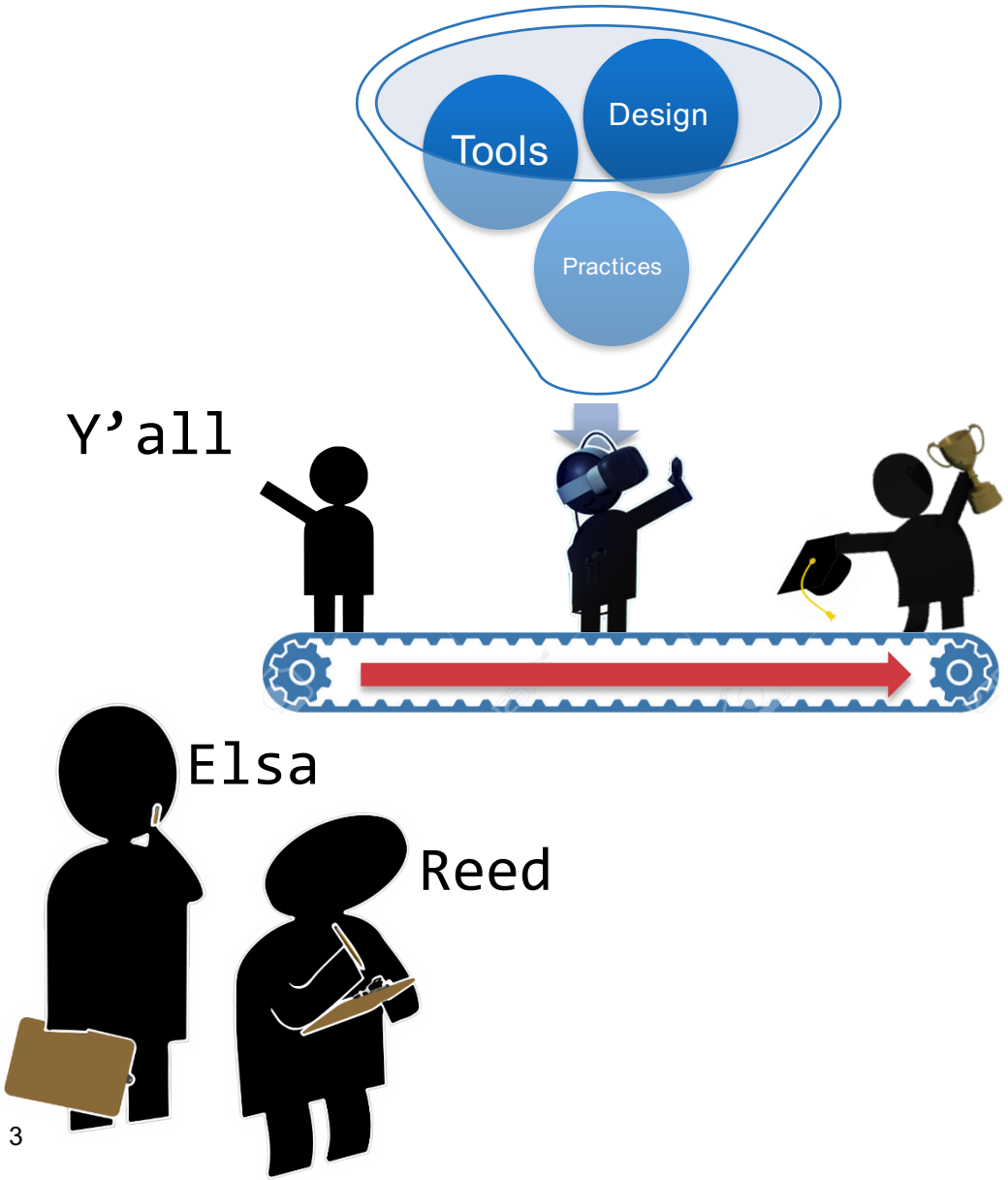
18 November 2019

SAND2019-14171 PE

Why We Are Here

- We are part of the US Exascale Computing Project (ECP)
  - Accelerating the delivery of next-generation computing ecosystems!
- Within the ECP, we are members of the Interoperable Design of Extreme-scale Application Software (IDEAS) team.
  - Improving the productivity and sustainability of ECP scientific software projects through better practices, processes, and tools!

Why We Are Here



- The future of science is equal parts theoretical, empirical, and computational.
  - Your career will involve creating, using, and interpreting software.
- The typical research software developer is a domain scientist or mathematician with little (if any) formal training in software development.
  - "It's crazy how much of my career has become devoted to making and maintaining software since I got here. I've had to learn a lot on the job."
- The Students@SC program gives people like Elsa and me the opportunity to help you realize your full potential.

Plan for Today's Lecture

Schedule	
2:30 PM – 3:30 PM	Reed's Talk on Software Design and Implementation
3:30 PM – 4:00 PM	Break
4:00 PM – 5:00 PM	Elsa's Talk on Tools and Practices

The aim is to provide you with tools and a mindset that you can apply to your own code.

How to Write Code You're Not Embarrassed to Share

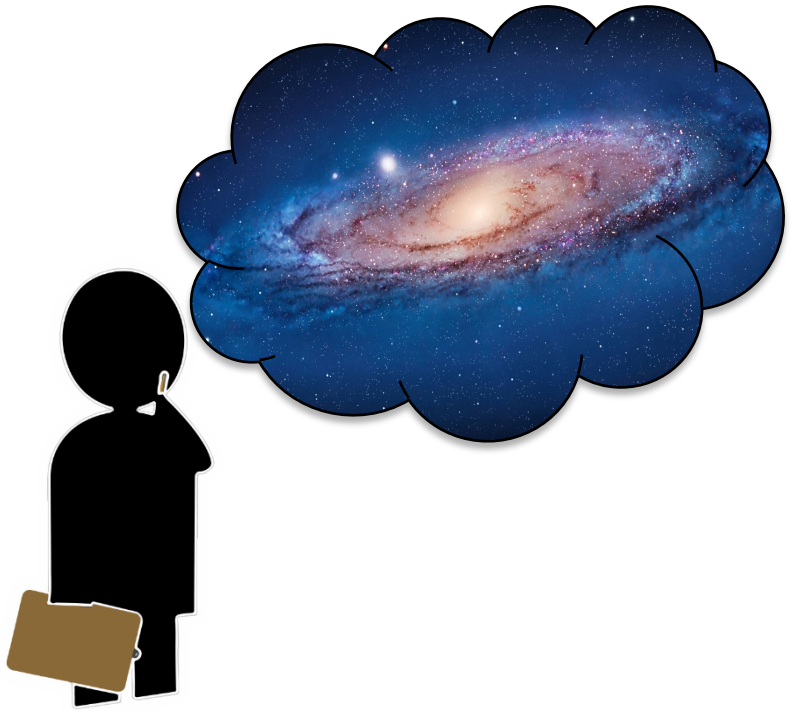
Reed Milewicz
Sandia National Laboratories
rmilewi@sandia.gov

18 November 2019

Who Am I?

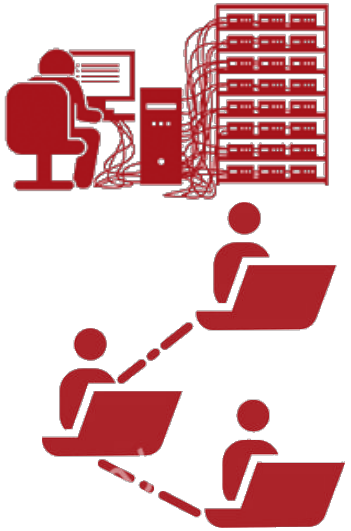
- I am a computer scientist who specializes in software engineering research.
- I try to **model, understand, and predict** the factors that lead to high-quality software systems.
- In turn, I aim to develop better **practices, processes, and tools** that drive those factors.
- Right now, much of my work targets the research software community.

Defining Research Software



It can range from tiny, personal scripts to multi-million-line libraries.

Research software is any software that helps answer a research question.



It can be a one-of code for research paper, or a project spanning many years.

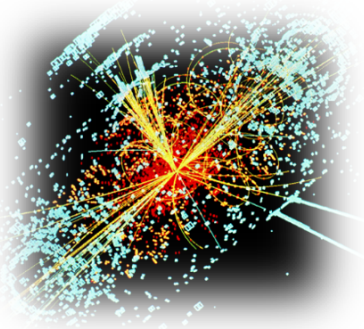
How to Write Code You're Not Embarrassed to Share

Reed Milewicz
Sandia National Laboratories
rmilewi@sandia.gov

Why?

18 November 2019

Why Should You Share Your Software?



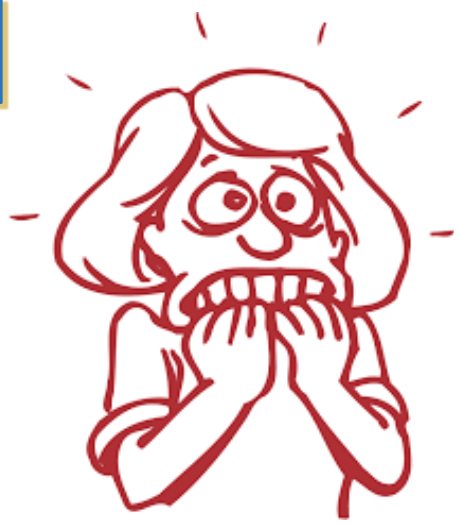
Software is revolutionary for research. It's an instrument for producing insight that is infinitely copiable and distributable, an engine for rapid innovation.

Sharing software likewise enables greater reproducibility of results, which is fundamental to the scientific method.

But here's what I really want to get across: the software you share is generally more useful **to you** than the software you don't.

Why Should You NOT Share Your Software?

1



"I'm afraid that there's a flaw in my code, and people will discover it if I release my software!"

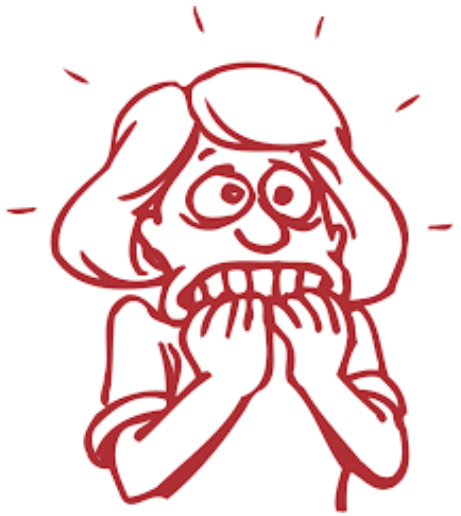
2

"My software is a competitive advantage. Other researchers might beat me to a publication using my code!"

3

"Releasing software implies that you will support that software. I'm not ready for that kind of commitment!"

Why Should You NOT Share Your Software?



"I'm afraid that there's a flaw in my code, and people will discover it if I release my software!"

Your peers aren't half as judgmental as you think.

Being paralyzed by fear that your code is bad becomes a self-fulfilling prophecy.

Why Should You NOT Share Your Software?

"My software is a competitive advantage. Other researchers might beat me to a publication using my code!"

This is often the first point in disguise.

This is reasoning is also self-sabotaging.
You only lose out on credit and influence.

Why Should You NOT Share Your Software?

"Releasing software implies that you will support that software. I'm not ready for that kind of commitment!"

That's not… true.
Regardless, if the software were written in a maintainable way, this wouldn't be a big concern.

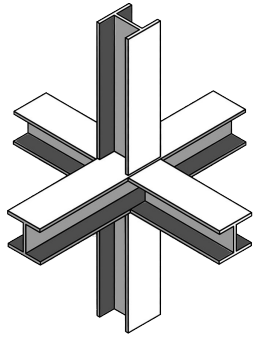
Why Should You NOT Share Your Software?

In my experience, underneath all these claims are fear and uncertainty regarding the **quality** of software.

Well, What Is “Quality” Anyway?

- Two kinds of quality:

Functional quality: Can the software do what it needs to (e.g. provide correct results given the right resources)? In other words, can it fulfill the functional requirements?

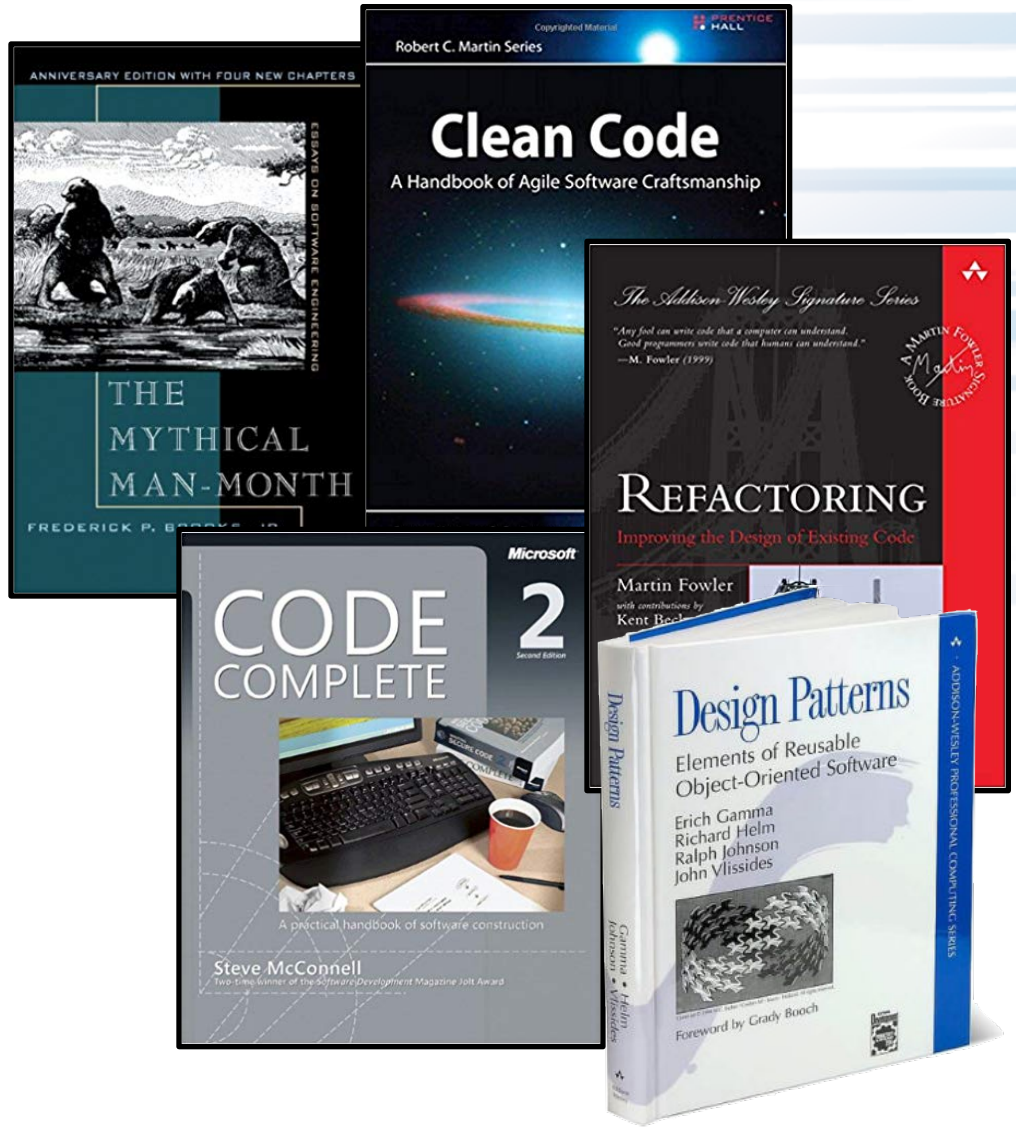


Structural quality: Is it written well? Can it fulfill the non-functional requirements?

- Most quality is internal, and – all things being equal – that’s where most of the **fear** is.

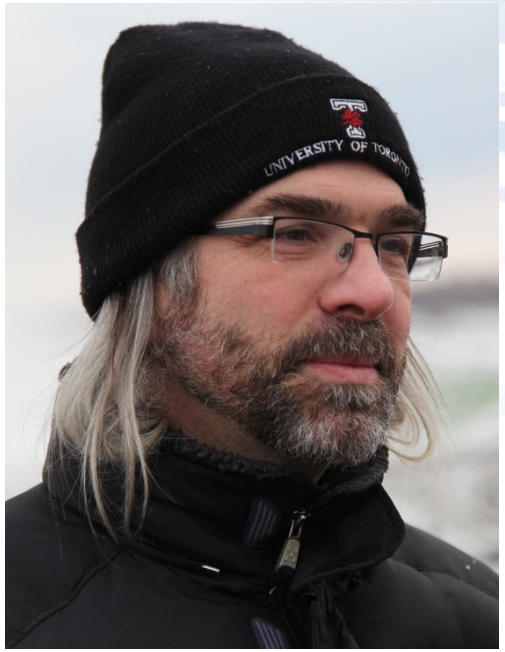
How Do We Achieve Quality?

- At the time of this presentation, 50 years of study and experience have gone into answering that question.
- Software engineering is like medicine.
  - We have cutting edge research developing innovative therapies (better languages, tools, methods), and these have helped tremendously.
  - But, much like a doctor, most of the advice that I have for you is about hygiene.
- Today I will try to answer two questions:
  - How to write code cleanly and effectively at the level of lines and routines.
  - How to think about and execute on software design more generally.



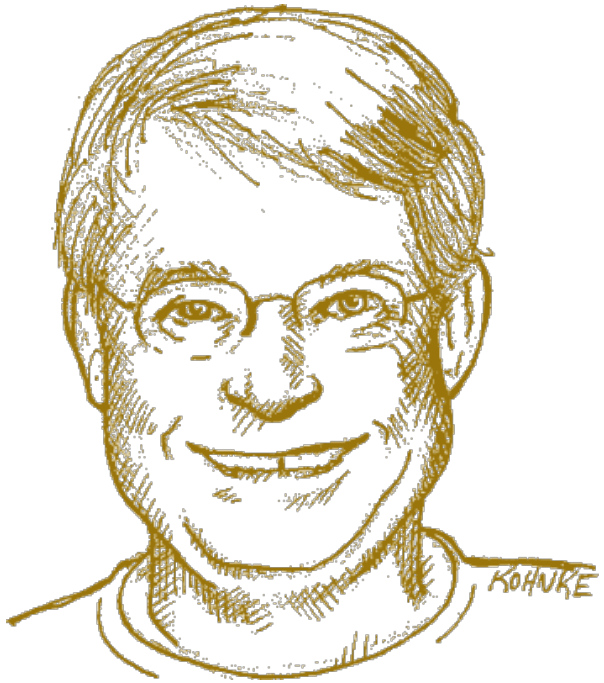
Write Programs for People, Not Computers

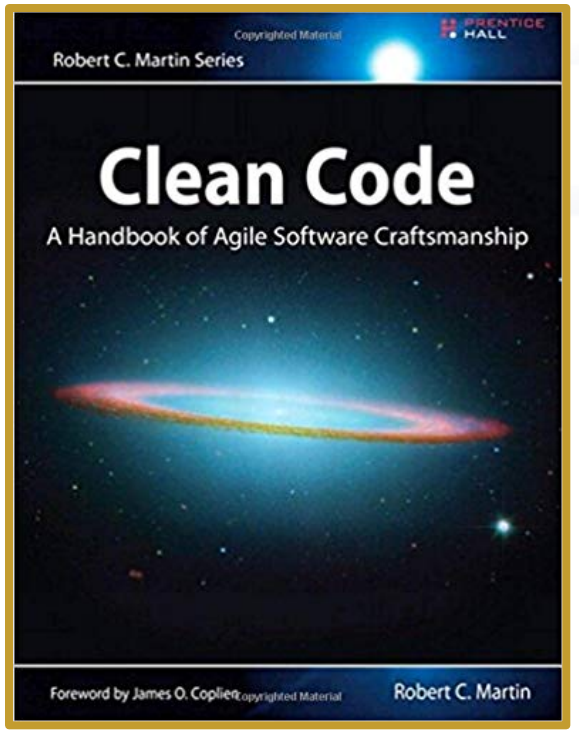
- Scientists writing software need to write code that both…
  - Executes **correctly**
  - Can be easily **read and understood** by other programmers (especially the author's future self).
- If software cannot be easily read and understood, it is much more difficult to know that it is actually doing what it is intended to do.
  - Human working memory is **limited**, human pattern matching abilities are **finely tuned**, and human attention span is **short.**
  - A program should not require its readers to hold more than a handful of facts in memory at once.



Wilson, Greg, et al. "Best practices for scientific computing." *PLoS biology* 12.1 (2014): e1001745.

Clean Code





Robert Martin, author of the book *Clean Code*, asked several famous/influential developers what they thought the phrase "clean code" meant to them.

What Makes Code "Clean"?

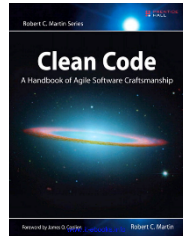
"Clean code is simple and direct. Clean code **reads like well-written prose.** Clean code never obscures the designer's intent but rather is full of crisp abstractions and straightforward lines of control."

Grady Booch

"You know you are working on clean code when each routine you read turns out to be **pretty much what you expected**."

Ward Cunningham

Robert Martin, author of the book *Clean Code*, asked several famous/influential developers what they thought the phrase "clean code" meant to them.

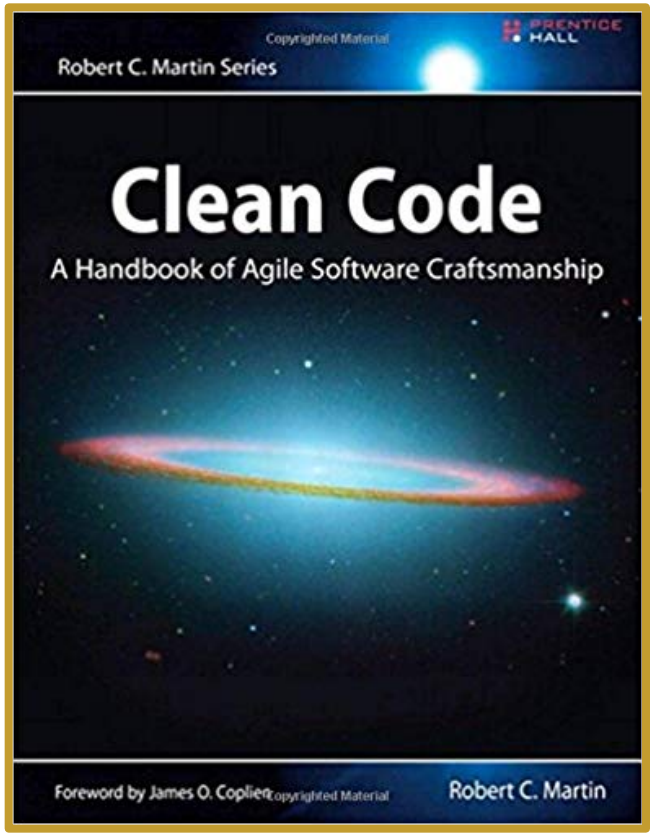


Why Does “Clean” Matter?

- Most software development consists of reading and using other people’s code, or code you yourself wrote in the past.
  - Scientific software needs to be accessible and interpretable just as much as, say, a research paper.
- Debugging is twice as hard as writing a program in the first place. If you are as clever as you can be when you write it, how will you ever debug it?[1]
- Research code is an idea in motion. It should be written in a way that allows you to comfortably express new ideas as they emerge in the research process.

[1] https://v4.software-carpentry.org/softeng/principles.html

Clean Code: Some Key Concepts



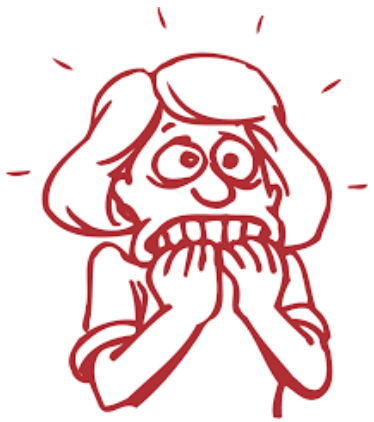
1. Names
2. Documentation
3. Functions

Clean Code: On Names

- Naming is perhaps the most powerful cognitive tool that we humans possess.
- Names are everywhere in software. We name our variables, our functions, our arguments, classes, and packages. We name our source files and the directories that contain them.
- But what's in a name?
- Choose names that…
  - are meaningful
  - convey your intentions
  - are pronounceable and searchable

On Names: Use Meaningful Names

```
fxd = dom.xd(i) + dom.xdd(i) * dt
```



”Wait, what’s wrong with this?! If you just follow along with the research paper…”

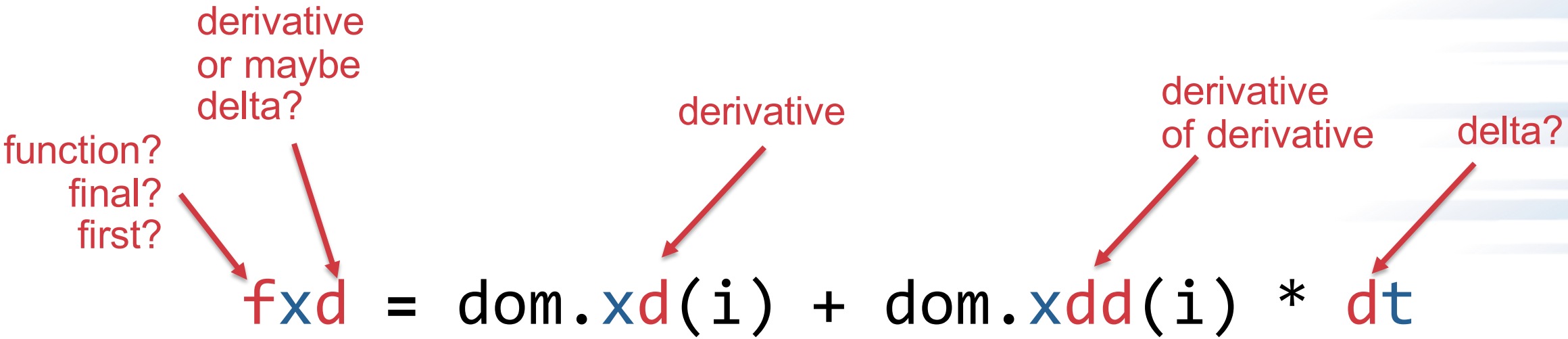
The code and the paper are separate, co-equal entities. Each should be able to explain itself on its own.

On Names: Use Meaningful Names

```
fxd = dom.xd(i) + dom.xdd(i) * dt
```

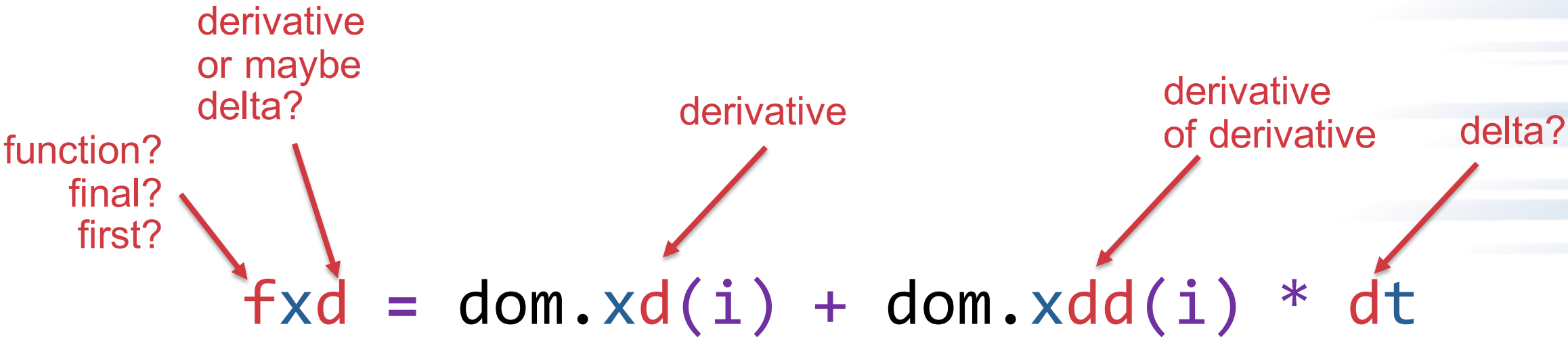
There's a variable called X involved.
Also a 'T'.

On Names: Use Meaningful Names



There's a variable called X involved.
Also a 'T'.

On Names: Use Meaningful Names



There's a variable called X involved.
Also a 'T'.

"the final velocity of an object is equal to its initial velocity added to its acceleration multiplied by time of travel."

Compute the final velocity on the X dimension at index i and store it in fxd.

On Names: Use Meaningful Names

The velocity and acceleration data are held by the domain object and can only be reached through accessor methods.

There's a variable called X involved.
Also a 'T'.

"the final velocity of an object is equal to its initial velocity added to its acceleration multiplied by time of travel."

Compute the final velocity on the X dimension at index i and store it in fxd.

I’m having to explain all this to you because the line itself, in isolation, doesn’t give you enough to go on.

Whether consciously or not, the reader is having to do these mental gymnastics for every line of code we write.

On Names: Use Meaningful Names

```
fxd = dom.xd(i) + dom.xdd(i) * dt
```

VS.

```
finalVelocityX = domain.getVelocityXAt(i) +
  domain.getAccelerationXAt(i) * deltaTime
```

"But this will make dense mathematics code hard to read!"

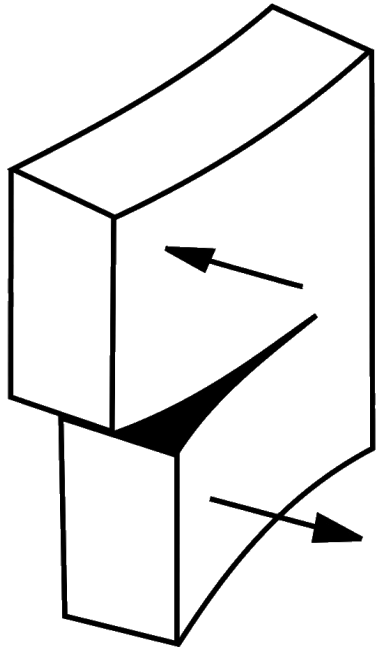
- Vary lengths of names according to their scope.
- Use spaces and line breaks.
- Decompose functions further.

On Names: Convey Your Intent

Is this variable name meaningful?

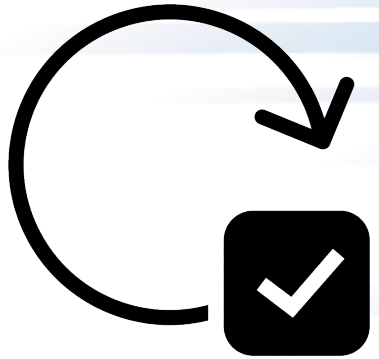
`shearupdate`

Yes!



`shear`

+

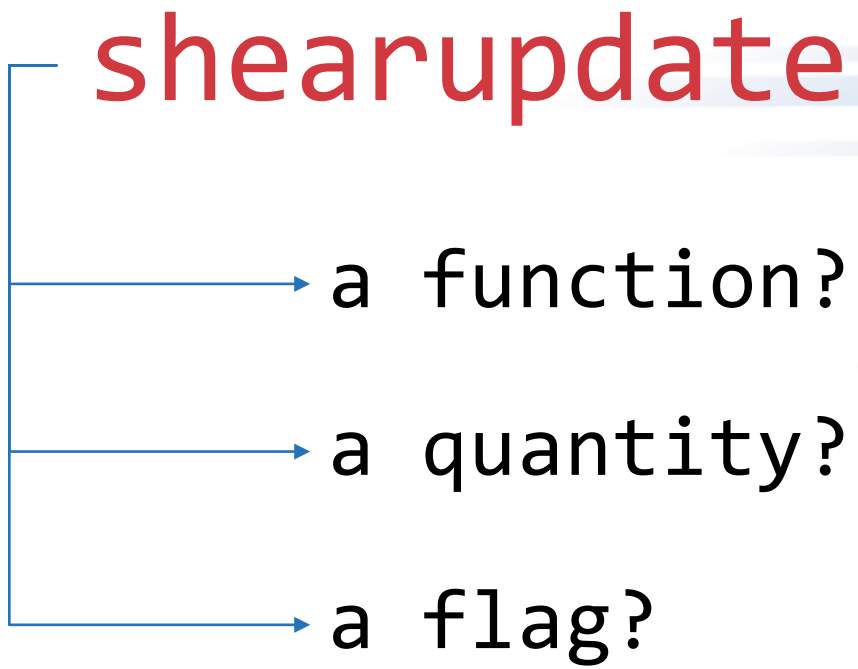


`update`

On Names: Convey Your Intent

But what about the **intent**? The name of a variable, function, or class, should answer all the big questions. It should tell you…

- why it exists
- what it does
- how it is used



On Names: Convey Your Intent

```
int shearupdate = 1;
if (update->setupflag)
     shearupdate = 0;
<…>
if (shearupdate) {
     shear[0] += vtr1*dt;
     shear[1] += vtr2*dt;
     shear[2] += vtr3*dt;
}
```

shearupdate

shouldUpdateShear

~~shearUpdateAmount~~

~~updateShear~~

On Names: Use Names That Are Pronounceable And Searchable

`m_qqc_monoq`

/em kksi: mɒnəʊkʌ/

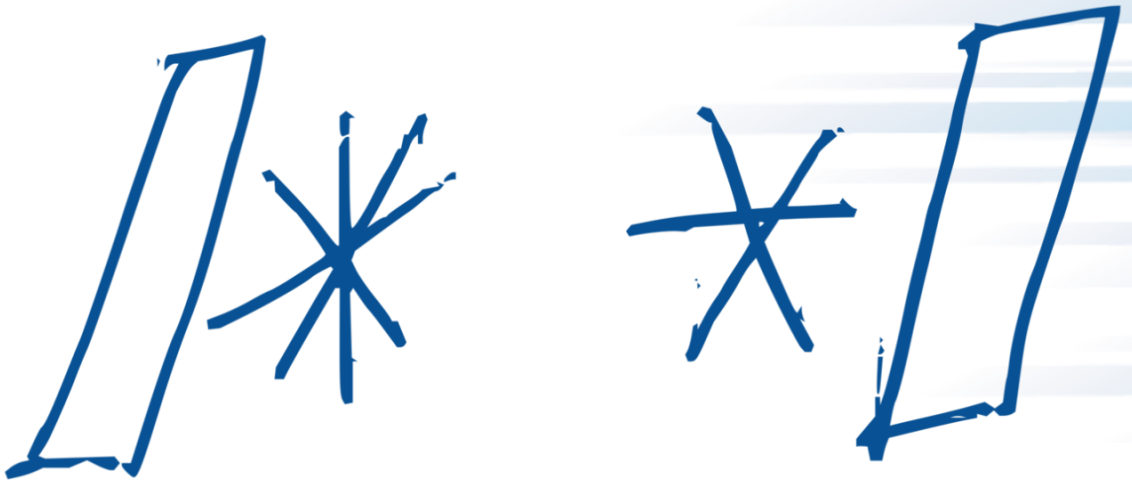
“Em k-k-see mono-cue”

The Quadratic term coefficient for ‘q’, which is the measure of viscosity of the material.

- If you can’t pronounce it, you can’t discuss it without sounding like an idiot. This matters because programming is a social activity.
- Single-letter names and numeric constants have a particular problem in that they are not easy to locate across a body of text.

Clean Code: On Documentation

- Documentation is expensive and you're unlikely to do it. Don't lie to yourself.
- Explain yourself in code.
  - Don't comment bad code—rewrite it!
- But comments can be helpful if used judiciously.



Clean Code: On Functions

- Functions do things. They enclose a set of instructions that we want to perform.
- Another way of looking at functions is that they're a way of making code **intellectually manageable**.
- There are many ways to organize a program into different functions.
- Some good rules of thumb:
  - Functions should be small.
  - They should do only one thing.
  - Don't repeat yourself.

On Functions: 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.
- Uncle Bob claims that 20 lines per function is a good target to aim and, ideally, they should be even smaller.

~20 Lines

On Functions: Functions Should Do One Thing

- Functions should do one thing.
  - They should do it well.
  - They should do it only.
- This is a good heuristic for keeping functions small.
- Let's use a real-world example...

```
void PairGranHookeHistory::compute(int
eflag, int vflag)
{
        150 Lines
}
```

On Functions: Functions Should Do One Thing

- Update rigid body info for owned & ghost atoms if using FixRigid masses.
- Loop over neighbors of my atoms.
- Unset non-touching neighbors.
- Compute relative translational velocity.
- Compute normal component.
- Compute tangential component.
- Relative rotational velocity.
- Compute MEFF = effective mass of pair of particles
- Compute normal forces = Hookian contact + normal velocity damping
- Compute shear history effects
- Rotate shear displacements
- Tangential forces = shear + tangential velocity damping
- Rescale frictional displacements and forces if needed.
- Compute forces and torques.

On Functions: Functions Should Do One Thing

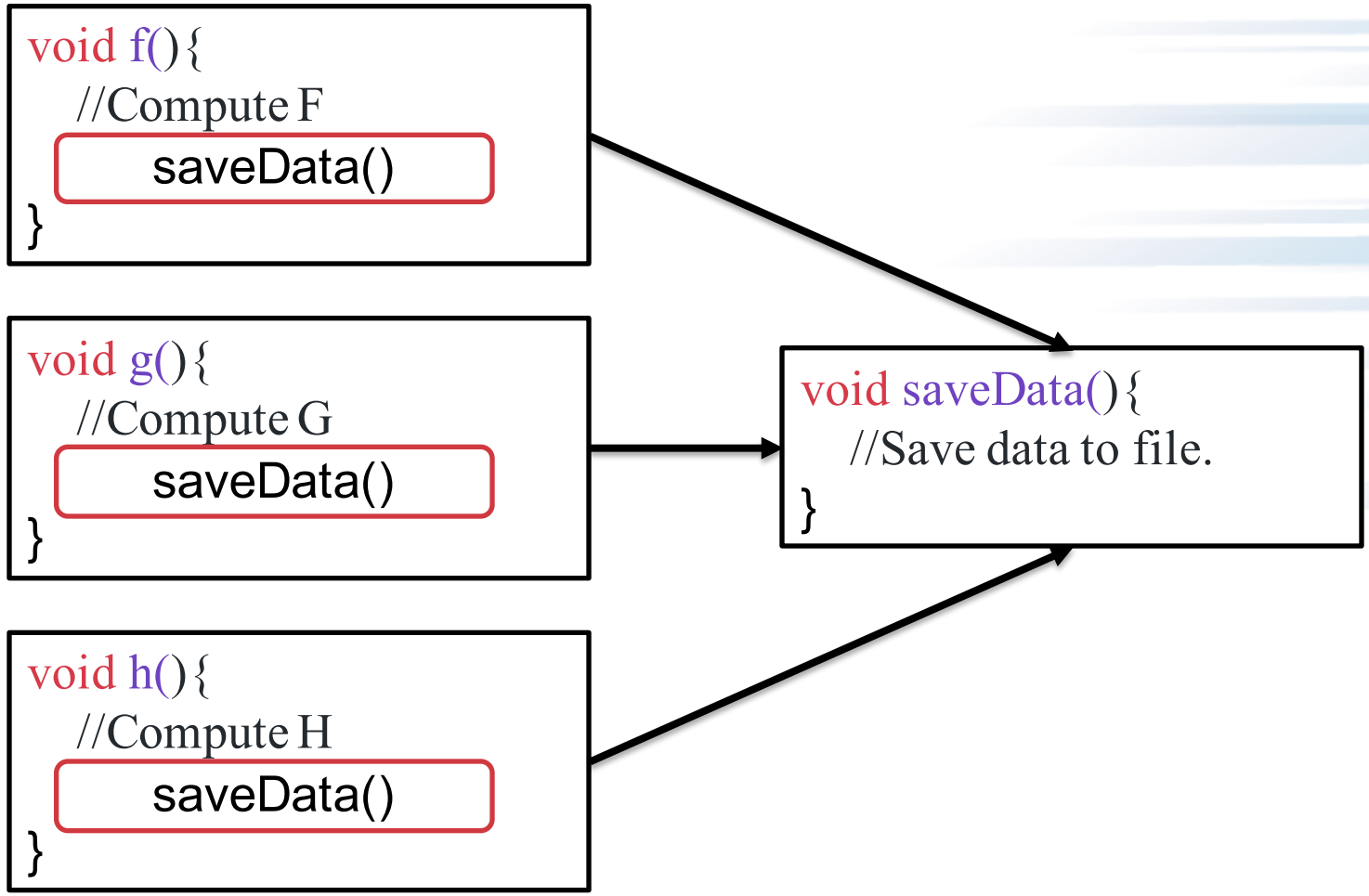
- Update rigid body info for owned & ghost atoms if using FixRigid masses. 33 lines
- Loop over neighbors of my atoms. ~30 lines
- Unset non-touching neighbors. 9 lines
- Relative translational velocity. 3 lines
- Compute normal component. 4 lines
- Compute tangential component. 3 lines
- Relative rotational velocity. 3 lines
- Compute MEFF = effective mass of pair of particles 10 lines
- Compute normal forces = Hookian contact + normal velocity damping 2 lines
- Compute shear history effects 5 lines
- Rotate shear displacements 9 lines
- Tangential forces = shear + tangential velocity damping 8 lines
- Rescale frictional displacements and forces if needed. 16 lines
- Compute forces and torques 19 lines

"All this code is related! Besides, function calls are expensive!"

- Related? Sure, but they do different things!
- Function call overheads are generally quite small.
- In any case, avoid pre-mature optimization! Write the code in a clean way first, *then* optimize it.

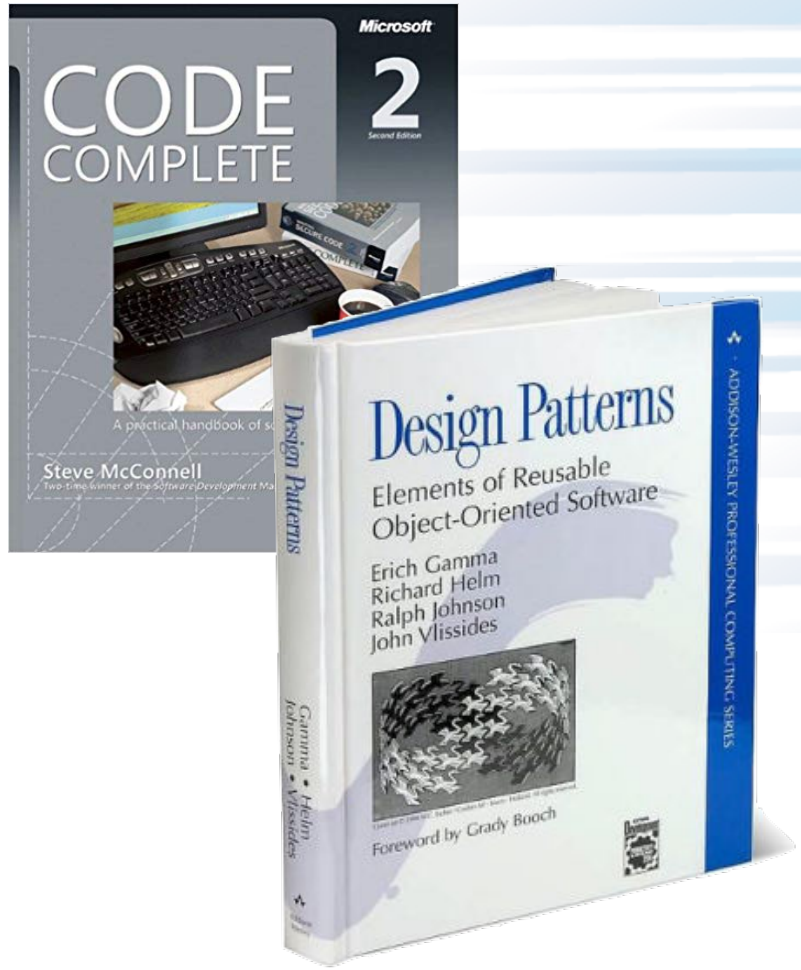
On Functions: Don't Repeat Yourself

- Duplication may be the root of all evil in software.
- Many innovations in software development have been an ongoing attempt to eliminate duplication from our source code.
- By splitting off frequently repeated code into functions, you make the code easier to test and change.



Clean Code: Recap

- Choose names that…
  - Are meaningful
  - Convey your intentions
  - Are pronounceable and searchable
- Explain yourself in code.
  - Don’t comment bad code, rewrite it.
- Write functions that…
  - That are small.
  - That only one thing.
  - That don’t force you to repeat yourself.

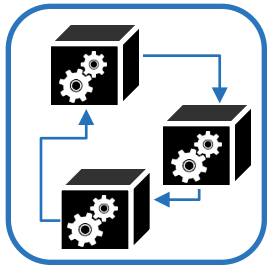


Next: On Design

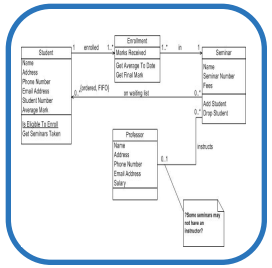
What is Design?

- Many researchers that I've interviewed picked up programming on their own, and they became adept at crafting scripts to solve their problems.
  - However, they don't always have a clear concept of design as something separate from the act of writing code.
- Software design is about conceptualizing and framing the way that a software system is organized.
- We talked about steps #4 and #5 in the last section. I want to focus on the first three now.

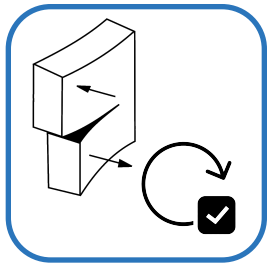
1. Software System



2. Division into subsystems/subpackages



3. Division into classes/submodules.



4. Division into data and routines.

```
for each…
do X…
then call
f…
```

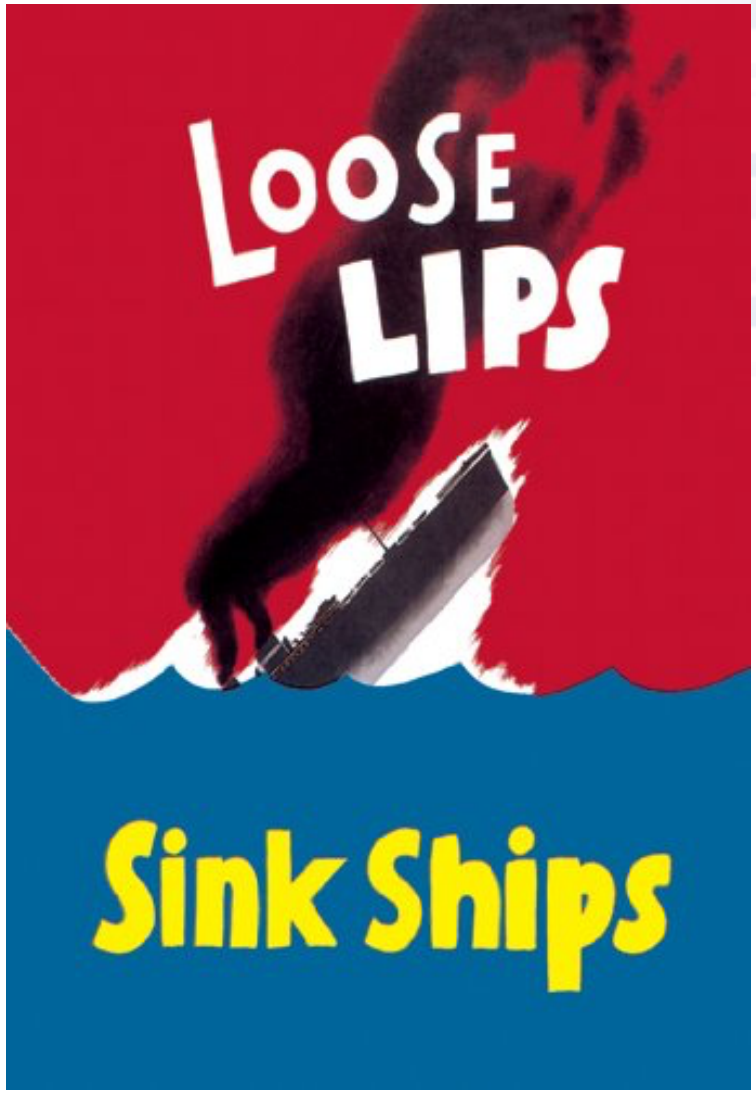
5. Internal Routine Design

What is Design About?

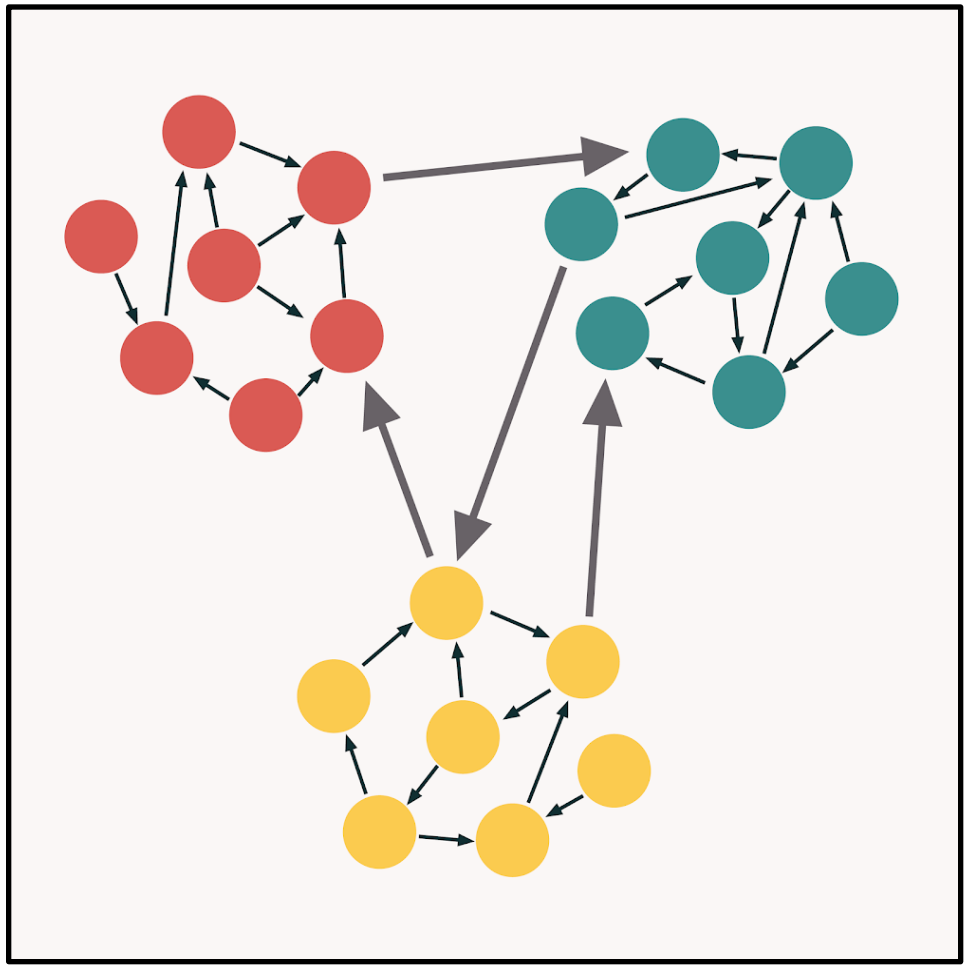
- Quality isn’t an accident, it’s the result of deliberate strategy.
- Design is about making mistakes, that’s the point. It’s cheaper to make mistakes and correct design than to write every line of code and then have to fix it all afterwards.
- Design is about trade-offs. No software can be perfect (see the Fallacy of Maximizing All).

Design Principles: Some Key Terms

Information Hiding: the principle of segregation of the *design decisions* in a computer program that are most likely to change, thus protecting other parts of the program from extensive modification if the design decision is changed.



Design Principles: Some Key Terms



Coupling: the degree of interdependence between software modules.

Cohesion: the degree to which the elements inside a module belong together.

Design Principles: Some Key Terms

The Principle of Least Knowledge (The Law of Demeter):

- Each unit should have only limited knowledge about other units: only units "closely" related to the current unit.
- Don't talk to strangers.
- Only talk to your immediate friends.

On Design: A Real-World Example

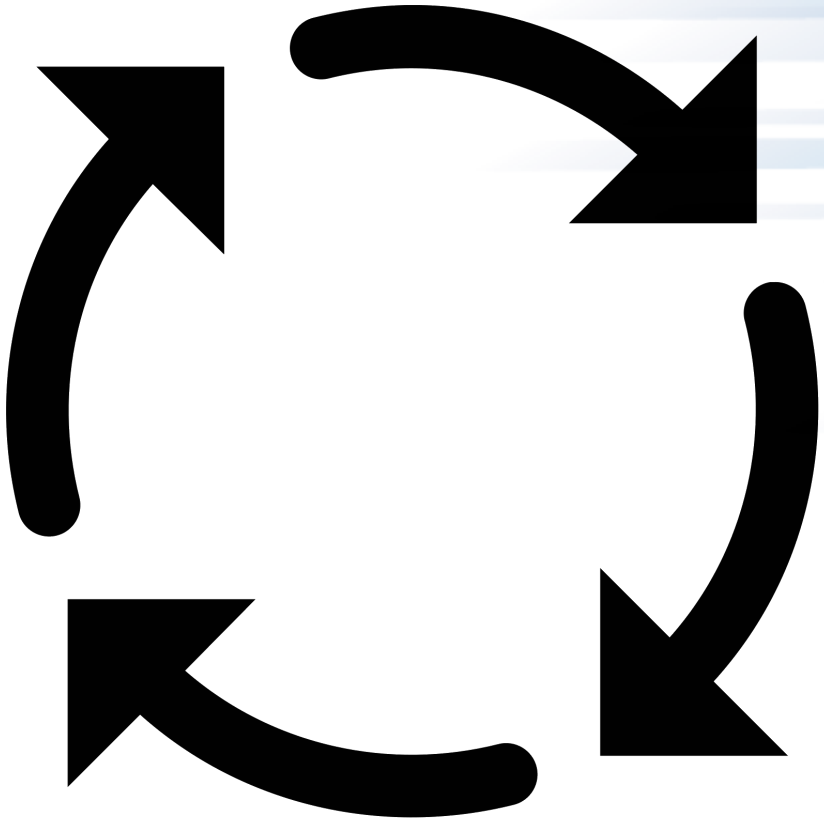
```
CARBON_SCALING_SLOPE = input("Enter the 13C scaling factor SLOPE:")

<...>

for carbon in conformation[CARBON_CS]:
    carbon.append(abs((CARBON_INTERCEPT-float(carbon[ISOTROPIC_VALUE]))
    CARBON_SCALING_SLOPE))
```

The code that handles the user's input interacts directly with the physics code through shared global variables.

Changing the I/O code could affect the operation of the physics code, even though these things are conceptually unrelated. They are tightly coupled.

Anticipating Change

- Information hiding is about anticipating what is most likely to change, and isolating those components. But what can change?
  - Domain math/science
  - Hardware dependencies
  - Input and output
  - Nonstandard language features
  - Difficult design/construction areas
  - Status variables
  - Data-size constraints



How To Isolate Code

- Keep conceptually dissimilar code in separate files.
- Program against function interfaces, make no assumptions about the internal workings of functions.
- In the case of object-oriented languages, use objects to encapsulate data and routines to minimize necessary knowledge.

A Sliding Scale of Coupling

Simple-data-parameter coupling

$$physics() \rightarrow analyze(500, True)$$

Simple-object coupling

$$physics() \rightarrow analyze(PhysicsModel)$$

Object-parameter coupling

$$physics() \rightarrow analyze(Model) \rightarrow visualize(Model)$$

Examples of Semantic Coupling

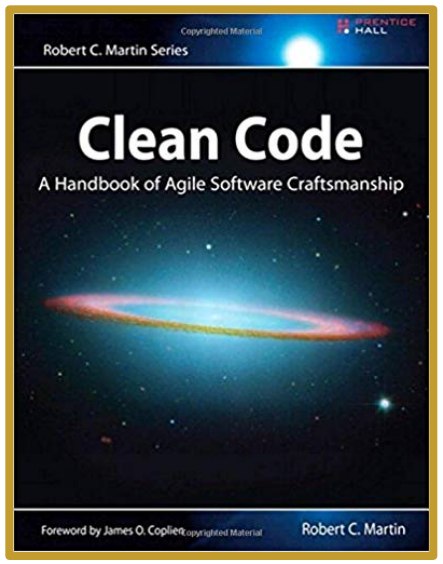
- The analysis module states that a caller must call `initializeAnalysis` before `analyze`. The physics module assumes that this has already been called, and only calls `analyze`.
- Physics modifies a global variable that Analysis depends upon.
- Visualization talks to the UI, and it can modify and pass back the physics model to the Physics module.

Semantic coupling is dangerous because changing code in one module can break code in another module in ways that are completely undetectable by the compiler.

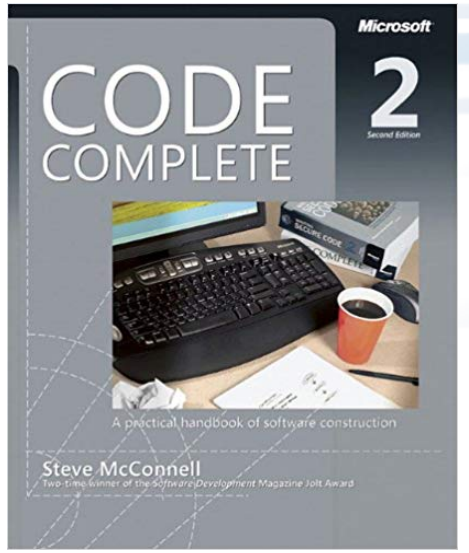
Conclusion

- The software you share is generally more useful to you than the software you don’t.
- All the fear and uncertainty around software can be managed through deliberate commitments to good practices.
- Scientific software needs to be accessible and interpretable just as much as, say, a research paper. Better yet, it needs to be even **more** accessible and interpretable than that!
- Write programs for people, not computers.

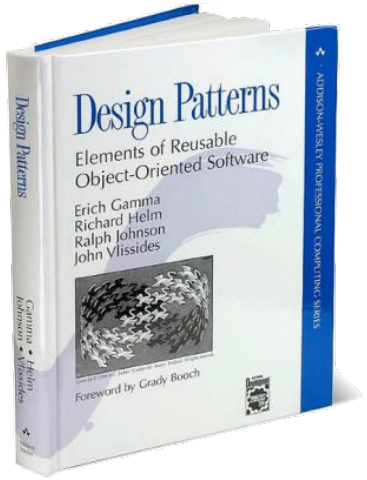
Passages Adapted From These Highly Recommended Texts



Martin, Robert C. *Clean Code: A Handbook of Agile Software Craftsmanship*. Pearson Education, 2009.



Gamma, Erich. *Design Patterns: Elements of Reusable Object-oriented Software*. Pearson Education, 1995.



McConnell, Steve. *Code Complete*. Pearson Education, 2004.