hack by plasmapy

Writing Clean Scientific Software

Nick Murphy

Center for Astrophysics | Harvard & Smithsonian

Many of these suggestions are from: *Clean Code* & *Clean Architecture* by R. C. Martin, *Best Practices for Scientific Computing* by Wilson et al., *Code Complete* by S. McConnell, *Design Patterns* by Gamma et al, *Software Engineering for Science* edited by Carver et al., and the *Copyright Guide for Scientific Software* by Albert et al.

We acknowledge support from NSF

Where I’m coming from...

- These suggestions do not come from:
  - Years of experience writing clean code
- Rather, these suggestions come from:
  - Years of experience writing messy code
  - And then living with the consequences… 

Common pain points with scientific software

hack by plasmapy

- Lack of user-friendliness
- Difficult installation
- Inadequate documentation
- Unreadable code
- Cryptic error messages
- Missing tests
- Often not openly available

Why do these pain points exist?

hack by plasmapy

- Programming **not covered in science courses**
- Scientists tend to be **self-taught** programmers
- Worth often measured by **number of publications**
- Code is often **written in a rush**
- **Time pressure** prevents us from taking time to learn
- Software **not valued** as a research product

How do we address these pain points?

- Make our software open source
- Use a high-level language
- Prioritize documentation
- Create automated test suites
- Develop code as a community
- **Write readable, reusable, & maintainable code**

My definition of clean code

- Readable and modifiable
- Easy to change
- Communicates intent
- Well-tested
- Well-documented
- Succinct
- Lets us understand the big picture
- Makes research fun!

“Code is communication!”

Which is more readable?

hack by plasmapy

```
>>> omega_ce=1.76e7*B
>>> electron_gyrofrequency = e * B / m_e
```

How do we choose good variable names?

hack by plasmapy

- **Reveal intention and meaning**
- **Choose clarity over brevity**
  - Longer names are better than unclear abbreviations
- **Avoid ambiguity**
  - Is `electron_gyrofrequency` an *angular* frequency?
  - Is `volume` in cm^3 or in barn-megaparsecs?
- **Be consistent**
  - Use one word for each concept
- **Use searchable names**

Change numbers to named constants

- In this expression:

```
velocity = -9.81 * time
```

  - Where does `-9.81` come from?
  - Are we sure it's correct?
  - What if we go to a different planet?

- Clarify intent by using *named constants* instead:

```
velocity = gravitational_acceleration * time
```

Decompose large programs into functions

- Huge chunks of code are hard to:
  - Read
  - Test
  - Keep track of in our mind
- Breaking code into functions helps us:
  - Reuse code
  - Improve readability
  - Isolate bugs

Don’t repeat yourself (DRY)

- Copying and pasting code is risky!
  - Bugs would need to be fixed *for every copy*
- Create functions instead of copying code
  - Simplifies fixing bugs
  - Reduces code duplication
- To change *one thing* in the code, we should only need to change it in *one place*

How do we write clean functions?

- Functions should:
  - Be **short**
  - Do **one thing**
  - Have **no side effects**
- Write explanatory note at top of function
- Avoid having too many required arguments
  - Use keywords or optional arguments
  - Define classes or data structures

High-level vs. low-level code

- High-level code:
  - Describes the big picture
  - “Abstracts away” implementation details
- Low-level code:
  - Describes implementation details
  - Contains concrete instructions for a computer

High-level vs. low-level cooking instructions

- High-level: describe goal of recipe
  - `Bake a cake`
- Low-level: a line in a recipe
  - `Add 1 barn-Mpc of baking powder to flour`

Avoid mixing low-level & high-level code

- Mixing low-level & high-level code makes it harder to:
  - Understand what the program is doing
  - Change the implementation
- Separate high-level, big picture code from low-level implementation details

Write code as a top-down narrative

To **perform a numerical simulation**, we:

1. Read in the inputs
2. Set initial conditions
3. Perform the time advances
4. Output the results

[1] This is called the “Stepdown Rule” in *Clean Code* by R. Martin.

Write code as a top-down narrative

hack by plasmapy

To **perform a numerical simulation**, we:

1. To **read in the inputs**, we:
    1.1. Open the input file
    1.2. Read in each individual parameter
    1.3. Close the input file
2. Set initial conditions
3. Perform the time advances
4. Output the results

Write code as a top-down narrative

hack by plasmapy

To **perform a numerical simulation**, we:

1. To **read in the inputs**, we:
   - 1.1. Open the input file
   - 1.2. To **read in each individual parameter**, we:
     - 1.2.1. Read in a line of text
     - 1.2.2. Parse the text
     - 1.2.3. Store the variable
   - 1.3. Close the input file
2. Set initial conditions
3. Perform the time advances
4. Output the results

The “extract method” refactoring pattern


```
def reduce_ccd_observation(raw_image):
    # Subtract bias
    (~20 lines of code)
    # Remove dark current
    (~20 lines of code)
    # Delete cosmic ray spikes
    (~30 lines of code)
```

- This function does more than one thing! 🙀
- What if we want to do only one of these steps?
- How do we test each individual step?

The “extract method” refactoring pattern

hack by plasmapy

Convert each section of code into its own function:

```
def subtract_bias(image): ...

def remove_dark_current(image): ...

def delete_cosmic_rays(image): ...

def reduce_ccd_observation(raw_image):
    image_level1 = subtract_bias(raw_image)
    image_level2 = remove_dark_current(image_level1)
    image_level3 = delete_cosmic_rays(image_level2)
    return image_level3
```

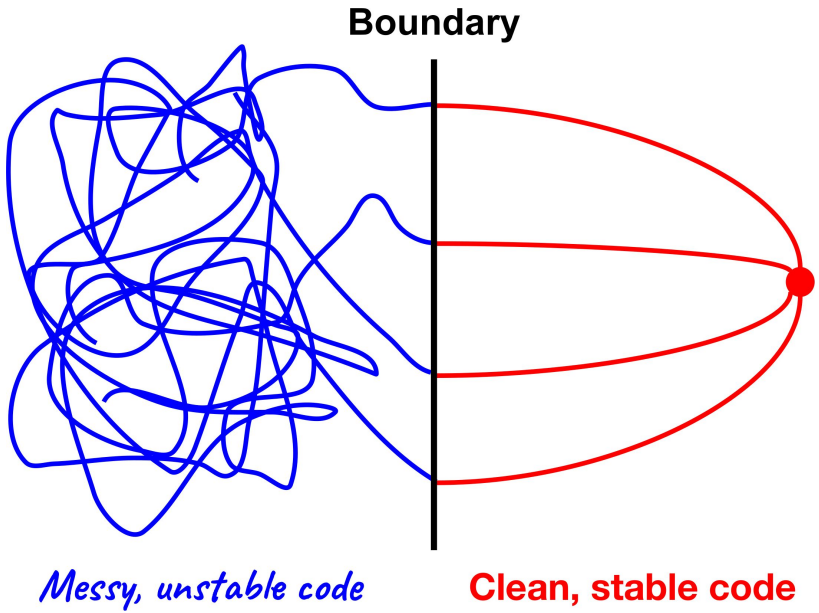
“Program to an interface, not an implementation”

- Suppose our program uses atomic data
- We’re using the Chianti database, but want to use AtomDB
- If our high-level code repeatedly calls Chianti, then…
  - Switching to AtomDB will be a pain!
- If our high-level code calls *functions that call Chianti*…
  - We need only make these *interface functions* call AtomDB instead
  - The high-level code can remain unchanged!

Quote from *Design Patterns*, Ch. 1

These interface functions represent a boundary

- Put a **boundary** between stable & unstable code
- The **clean, stable code** depends directly on the **boundary**, not the *messy unstable code*
- The **boundary** should be stable



Strive for high cohesion & low coupling

- **Cohesion** is the degree to which the contents of a module belong together
- **Coupling** is the degree to which the contents of a module depend on other modules
- Code elements that change together at the same time for the same reasons belong together
- Separate code elements that do not change with each other

Comments are not inherently good!

- As code evolves, comments often:
  - Become out-of-date
  - Contain misleading information
  - Get displaced from the corresponding code
- “A comment is a lie waiting to happen” 

Not so helpful comments

- Commented out code
  - Quickly becomes irrelevant
  - Use version control instead
- Definitions of variables
  - Encode definitions in variables names instead
- Redundant comments

```
i = i + 1  # increment i
```

- Description of the implementation (usually)
  - Becomes obsolete quickly
  - Communicate the implementation *in the code itself*

Helpful commenting practices

- Explain the *intent* and *interface*
  - Refactor code instead of explaining how it works
- Amplify important points
- Explain why an approach was *not* used
- Provide context and references
- Explain concepts unfamiliar to readers
- Update comments when updating code

When should we write clean code?

hack by plasmapy

- Some clean coding habits save time quickly
  - Writing short functions that do one thing
  - Writing tests
- Interactive exploration of a data set does not necessitate particularly clean code
- Investing extra time is worthwhile if:
  - You'll re-use the code
  - The code will be shared with others
- Avoid perfectionism
  - Better to *mostly* (but not completely) follow this advice

Summary

hack by plasmapy

- **Code is communication!**
- Choose names wisely
- Break up complicated code into manageable chunks
  - Write short functions that do one thing
  - Separate big picture code from implementation details
- Refactor code rather than explaining how it works
  - Communicate the implementation in the code itself