REU WORKSHOP - PA DEPARTMENT - MSU - SUMMER 2017

PROGRAMMING PRACTICES + SOURCE CODE MANAGEMENT

D. W. Silvia

WHY ARE WE HERE?

WHY ARE WE HERE?

- Scientific research is becoming increasingly dependent on computers
- Even outside of “computational” science, most of our work depends on writing and running code
- Poorly written, undocumented code without a version history is **not useful** and makes *reproducibility* **difficult**

HOW DO WE FIX THIS?

- Write quality code
    - Use good, consistent practices; include comments; design code to be modular and flexible; write documentation
- Use version control! (e.g. git, mercurial)

WRITING QUALITY CODE

BECAUSE YOU REALLY SHOULD.

"Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live."

–JOHN F. WOODS (C++ PROGRAMMER)

GOOD BASIC PRACTICES

- Don't write lines of code longer than 80 characters, while this has a history in punch cards, it increases code readability.
- Use spaces for indentation rather than tabs — tabs can lead to weird formatting across machines
- Be wary of overly nested code
- Avoid writing overly long functions, if a function gets long, look for ways to split it up.

GOOD BASIC PRACTICES (CONT'D)

- Use consistent, clear naming conventions
    - Examples:
        - For variables, use "lower with under" style and when you first define a variable, it's worth commenting the purpose of the variable

```
photon_count = 0 #track the number of photons collected
```

GOOD BASIC PRACTICES (CONT'D)

- Use consistent, clear naming conventions
  - Examples:
    - For symbolic constants (e.g. physical constants), define them separately and use all caps

```
BOLTZMANN_CONSTANT = 1.38e-23 #Boltzmann’s constant in Joules/Kelvin
```

GOOD BASIC PRACTICES (CONT'D)

- Use consistent, clear naming conventions
    - Examples:
        - For functions, also use the "lower with under" style and include comments to indicate the purpose of the function and the parameters it uses

```
def count_photons(pixel, start_time, end_time):
    “””
    Counts the number of photons that hit a given pixel

    Receives: an integer representing a pixel ID (pixel)
        as well as the start and end time for counting
        pixels as floating-point values
        (start_time, end_time)
    Returns: the number of photons that impact a given
       pixel over a specified time interval
    “””
```

GOOD BASIC PRACTICES (CONT'D)

- Use consistent, clear naming conventions
  - Examples:
    - For classes, use "CamelCase" to separate them from functions

```
class PhotonDetector():
    “ ” ”
    The PhotonDetector class contains all of
    machinery necessary for counting and
    analyzing photons.

    Parameters:
    …
    …
    “ ” ”
```

OTHER GOOD PRACTICES

- Write modular code
  - Split complex code into functions with unique purposes and comment/document those functions!
  - Break up code into separate scripts when the code base gets large —> this simplifies working with shared repositories

OTHER GOOD PRACTICES

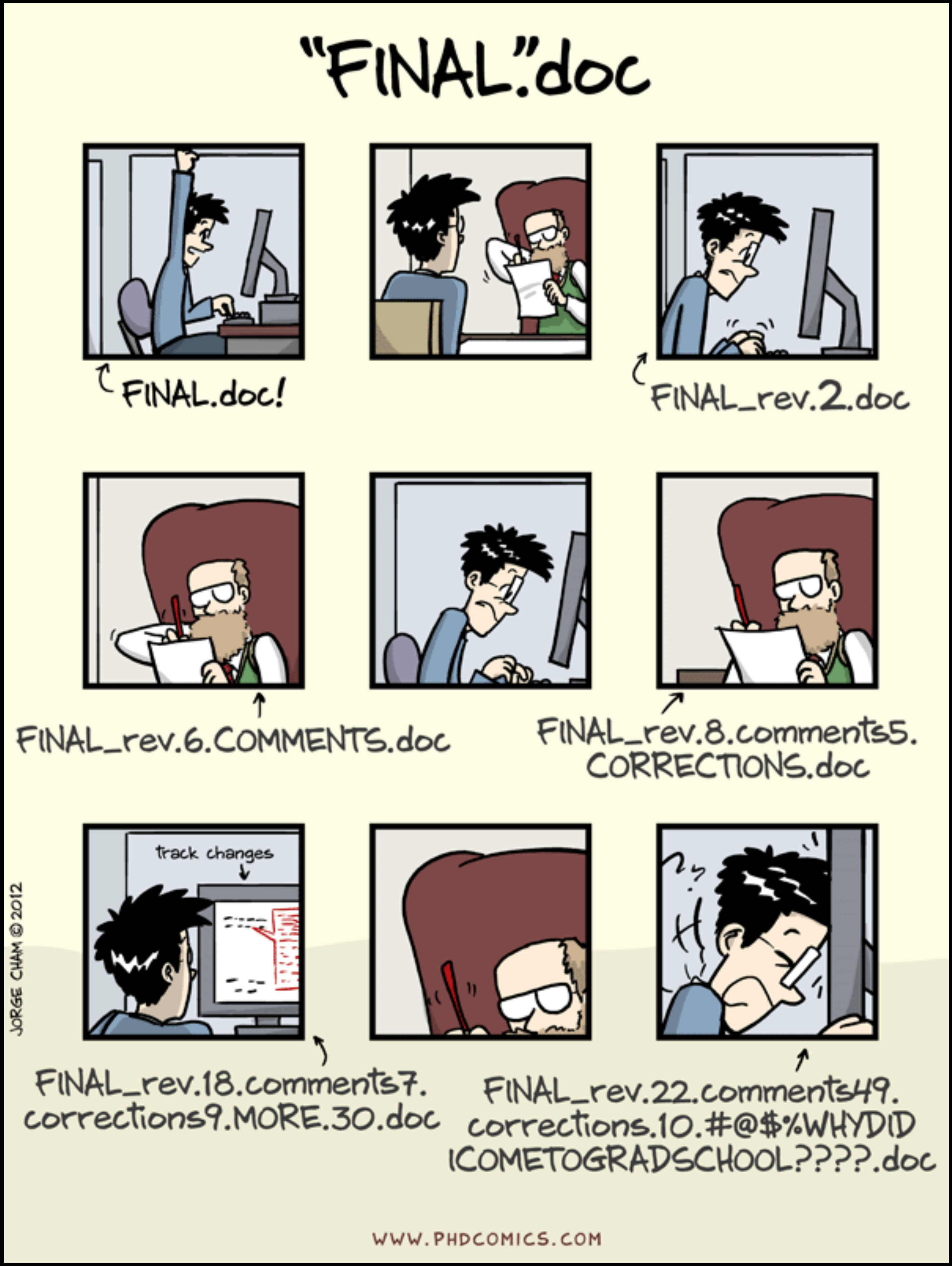
- Write documentation!
  - Include inline documentation that explains the purpose of functions, what variables are, how the code works, references to models, etc.
  - Produce documentation that allows others to use and understand your code.
    - Simplest: include a README file in your code's root directory
    - More complex: write detailed documentation that can be viewed online

NEVER LOSE CODE AGAIN WITH…

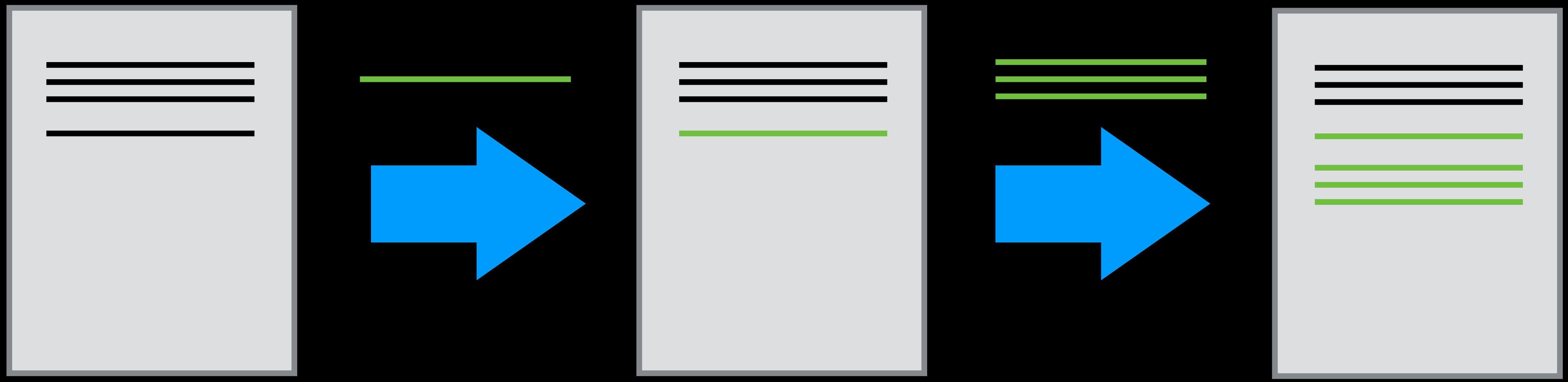
VERSION CONTROL

WHAT IS IT?

It's not this:



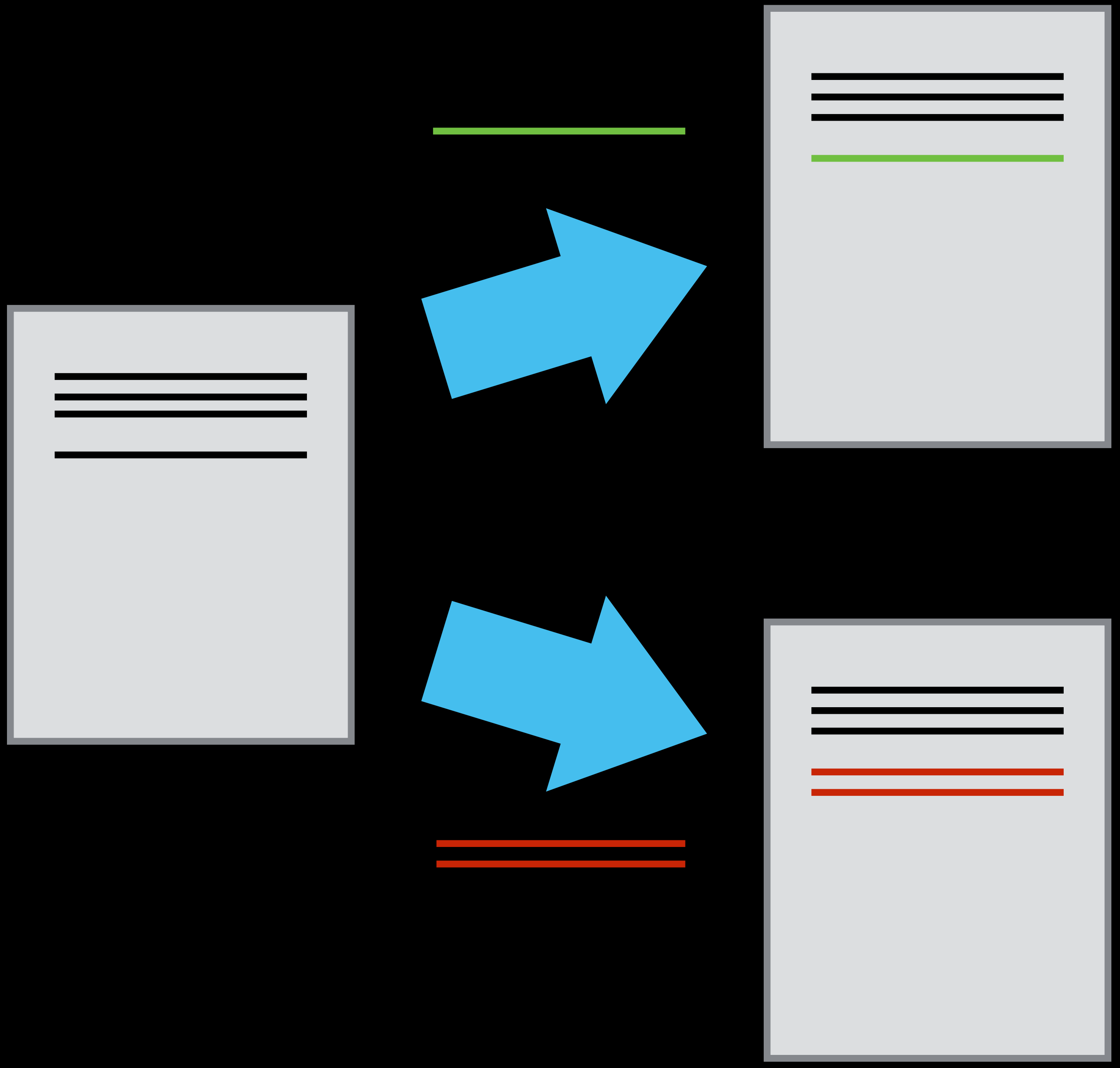
WHAT IS IT?



Automated version tracking.

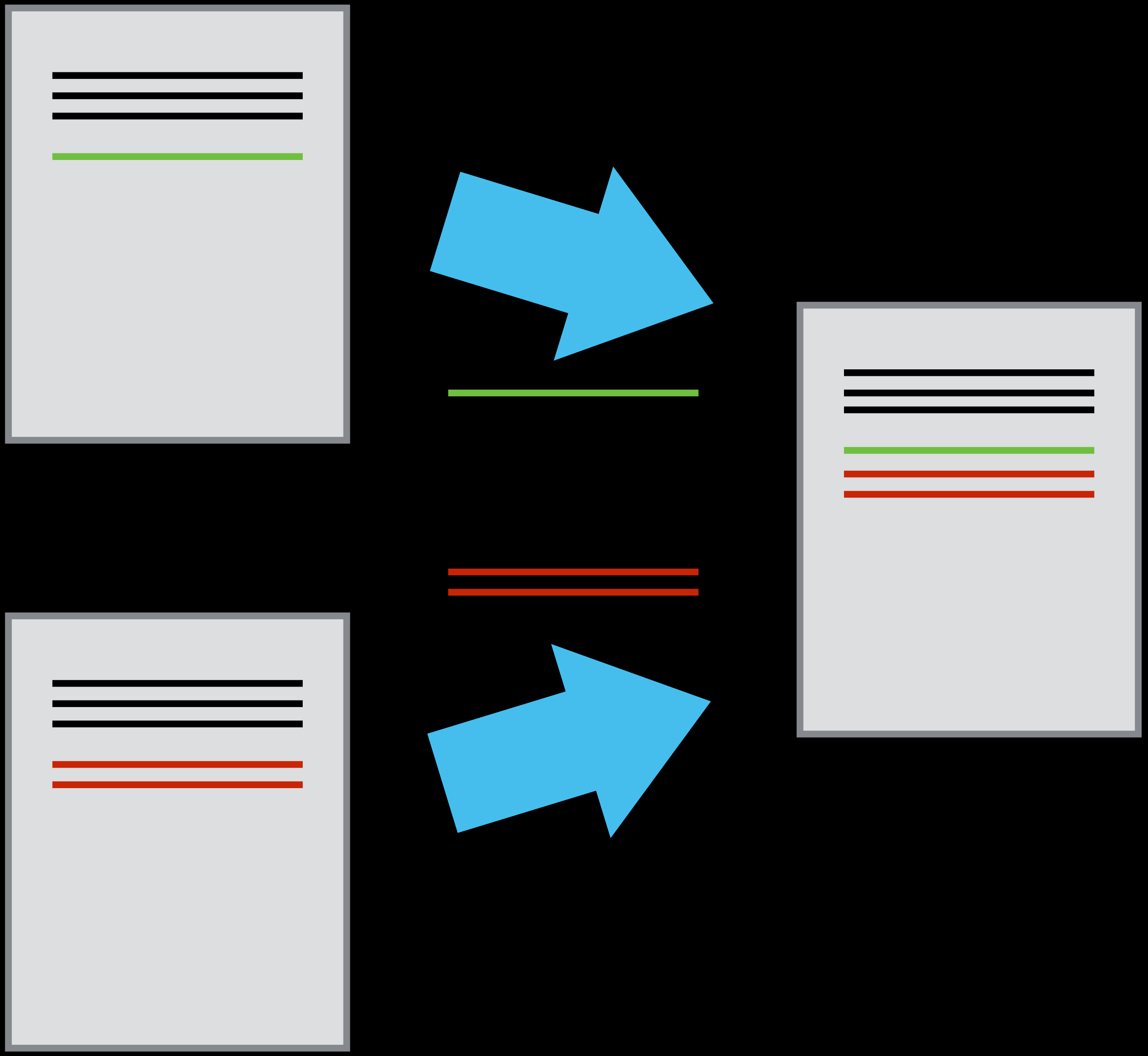
Every change is a new commit.

WHAT IS IT?



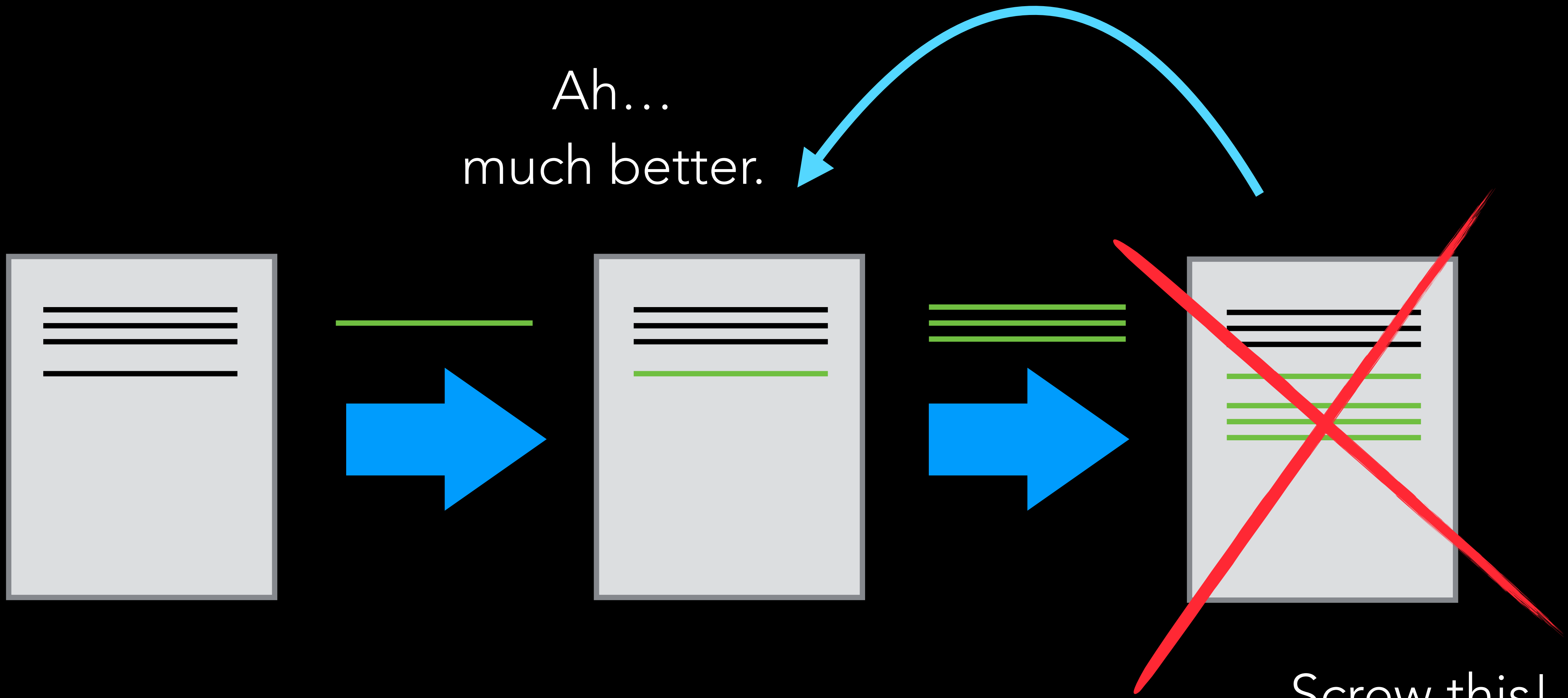
Different people can have different versions.

WHAT IS IT?



Assuming there are no direct conflicts, you can merge changes.

WHAT IS IT?



And if something goes wrong?
Jump back to a previous version.

LET'S GIVE IT A GO

THINGS ARE ABOUT TO GET INTERACTIVE…

NOW, TAKE SOME OF YOUR OWN CODE AND SET UP A REPOSITORY