

# Software Archaeology and Anthropology

17-313 Fall 2023

Foundations of Software Engineering

<https://cmu-313.github.io>

Andrew Begel and Rohan Padhye

# Administrivia

- Slack
  - Please add a profile picture.
  - Ask questions in #general or #technical-questions. Please use threads.
- Office hours can be found on the course home page: <http://cmu-313.github.io>
- For those of you who requested to swap recitations, stay tuned.
- COVID or other health issues? Please stay home.

# Smoking Section

- Last full row



# Homework

- Homework 1 is released.
  - Part (a) is due Friday Sept 1, 11:59 pm. **That's tomorrow!**
  - Part (b) is due Thursday, Sept 7, 11:59pm.
  - This is an individual assignment; we will compose groups next week.
  - Get started early, ask for help, and check the #technical-questions channel; chances are your questions have been asked by others!

# Team Formation Survey Due Friday

- Team formation survey is posted on Canvas and in Slack #announcements.
- Please fill in by TONIGHT!

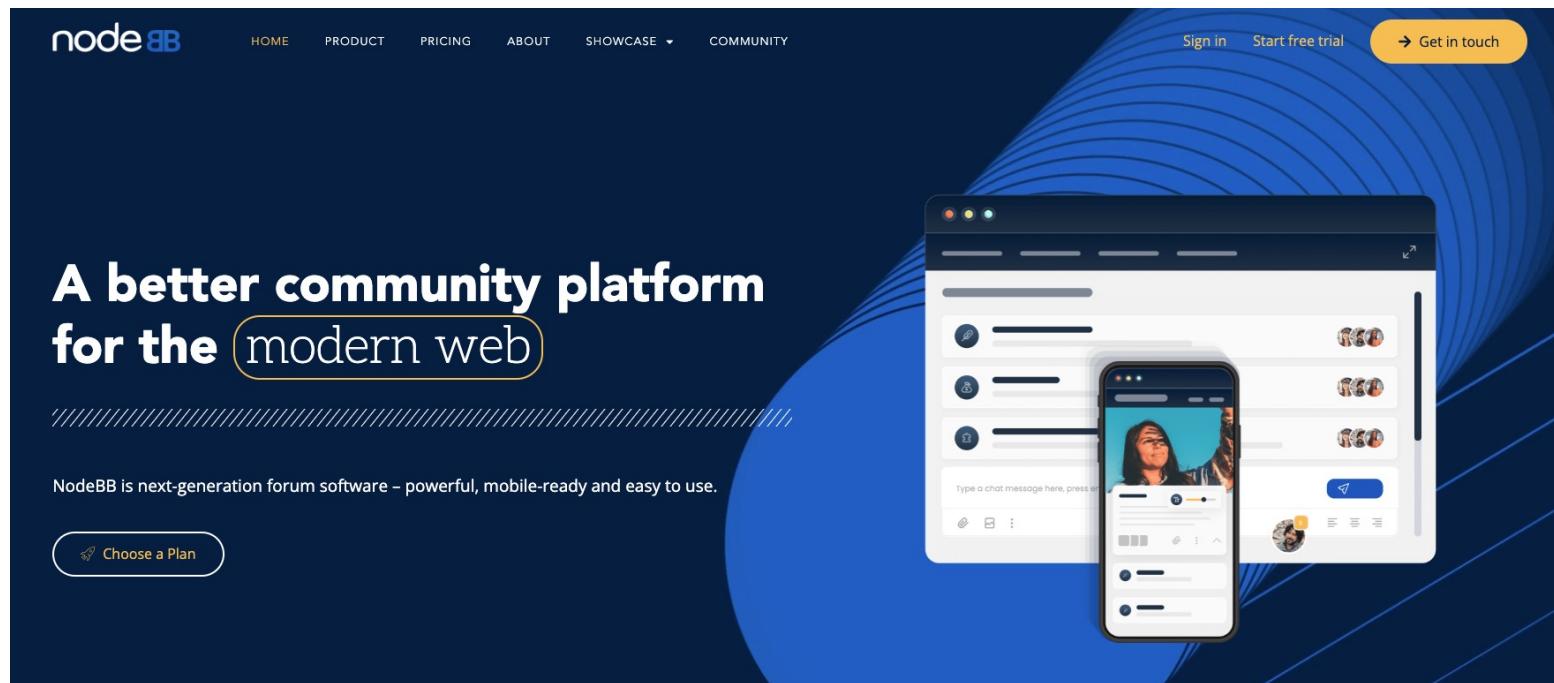


# Learning Goals

- Understand and scope the task of taking on and understanding a new and complex piece of existing software
- Appreciate the importance of configuring an effective IDE
- Contrast different types of code execution environments including local, remote, application, and libraries
- Enumerate both static and dynamic strategies for understanding and modifying a new codebase

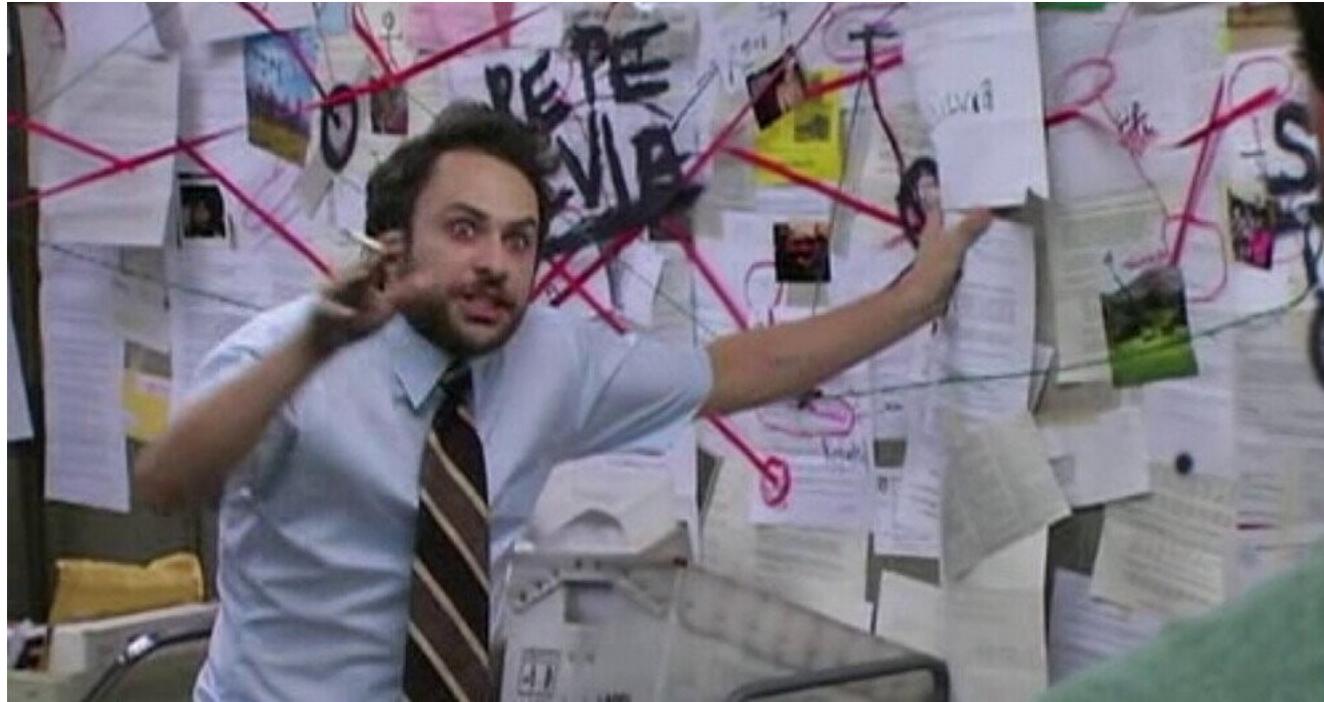
# Context: big ole pile of code

- ... do something with it!



You will never  
understand the  
entire system!

# Challenge: How do I tackle this codebase?



# Challenge: How do I tackle this codebase?

- Leverage your previous experiences (languages, technologies, patterns)
- Consult documentation, whitepapers
- Talk to experts, code owners
- Follow best practices to build a working model of the system

# Bad news: There are few helpful resources!

- **Working Effectively with Legacy Code.**

Michael C. Feathers. 2004.

- **Re-Engineering Legacy Software.**

Chris Birchall. 2016.

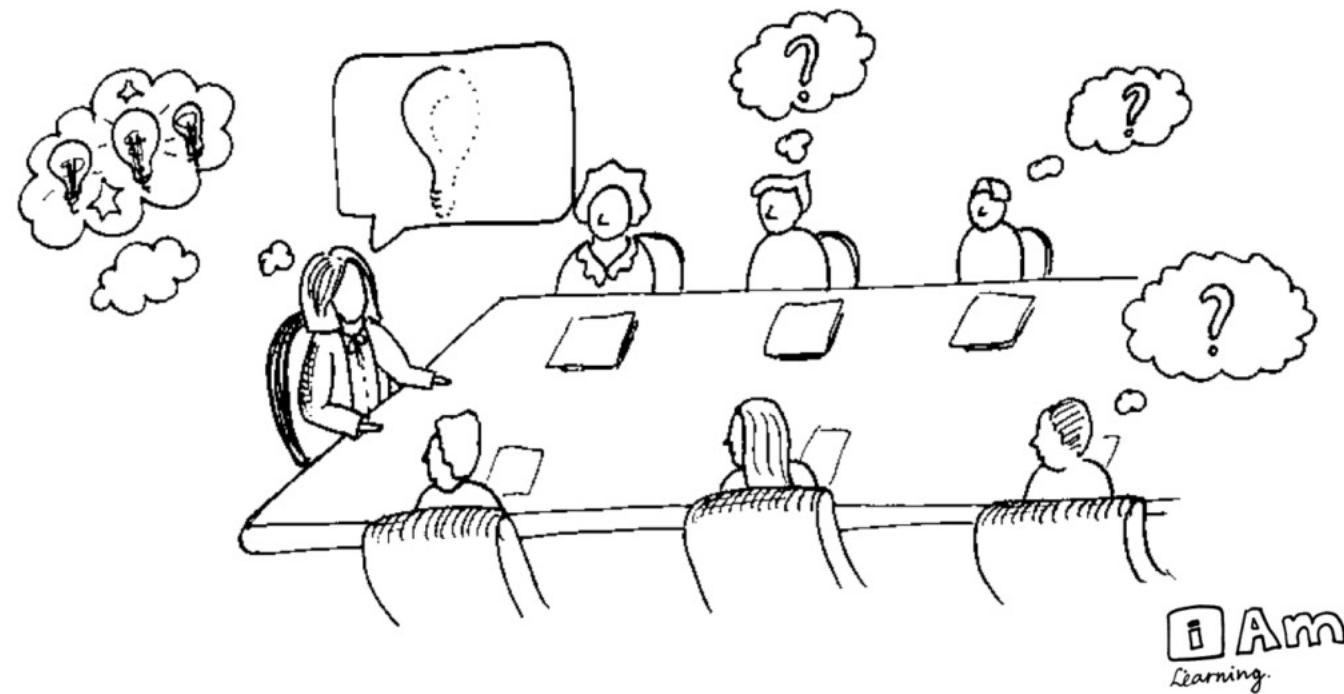
- **The Legacy Code Programmer's Toolbox.**

Jonathan Boccaro. 2019.



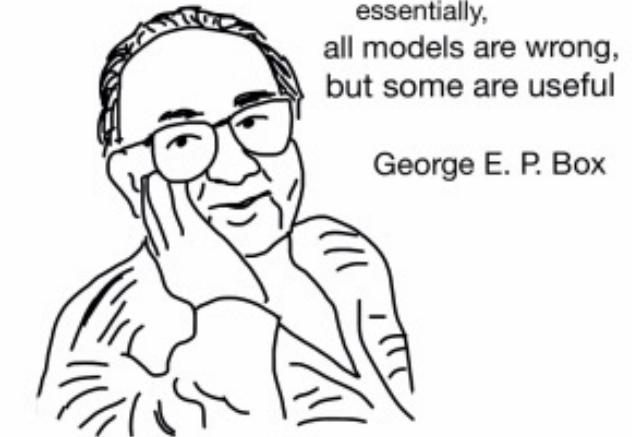
A screenshot of the Wealthsimple Magazine website. At the top right, there are navigation links for 'START INVESTING', 'Wealthsimple Magazine', and 'MENU'. Below the header, there is a section titled 'MONEY &amp; THE WORLD' with an article thumbnail for 'The Code That Controls Your Money'. The main content area discusses COBOL, describing it as a coding language older than Weird Al Yankovic, which underpins the entire financial system. To the right of the article, there is a sidebar with a section titled 'THE LEGACY CODE PROGRAMMER'S TOOLBOX' featuring a cartoon illustration of a toolbox and its contents.

# Why? Because of Tacit Knowledge



# Today: How to tackle codebases

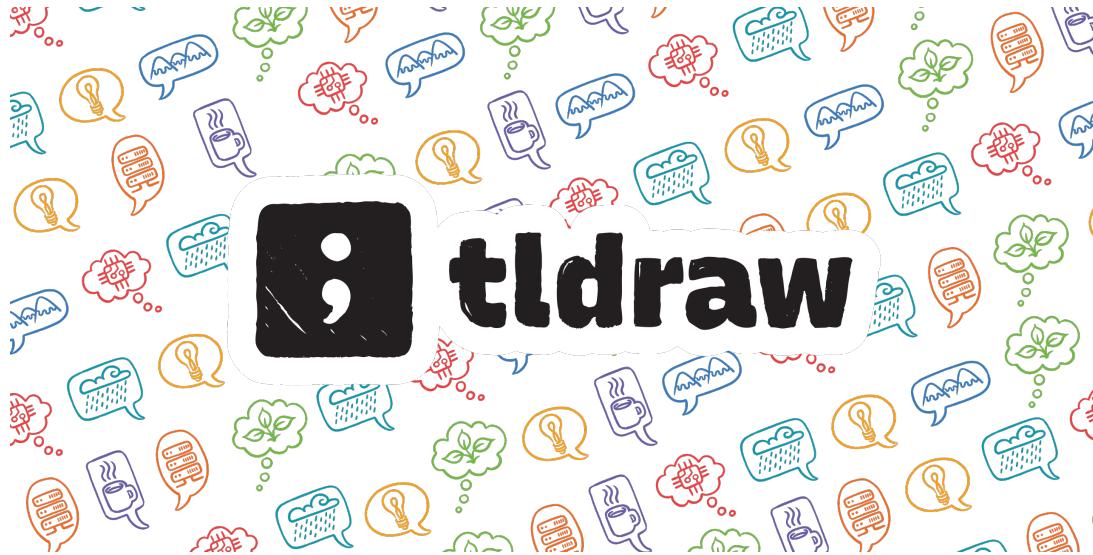
- Goal: develop and test a working model or set of working hypotheses about how (some part of) a system works
- Working model: an understanding of the pieces of the system (components), and the way they interact (connections)
- Focus: Observation, probes, and hypothesis testing
  - Helpful tools and techniques!



essentially,  
all models are wrong,  
but some are useful

George E. P. Box

# Live Demonstration: tldraw



<https://github.com/tldraw/tldraw>

# Steps to Understand a New Codebase

- Look at README.md
- Clone the repo.
- Build the codebase.
- Figure out how to make it run.
- What do you want to mess with?
  - Clone and own
- Traceability - Attach a debugger
  - View Source
  - Find the logs.
  - Search for constants (strings, colors, weird integers (#DEADBEEF))

# Participation Activities

- Pull out your phone.
- Download the Gradescope app.
- Log into Gradescope.
  - Use email login, not SSO.
- Click “+” button to add 17-313.
  - Entry Code: G24487



Download Gradescope  
for iPhone



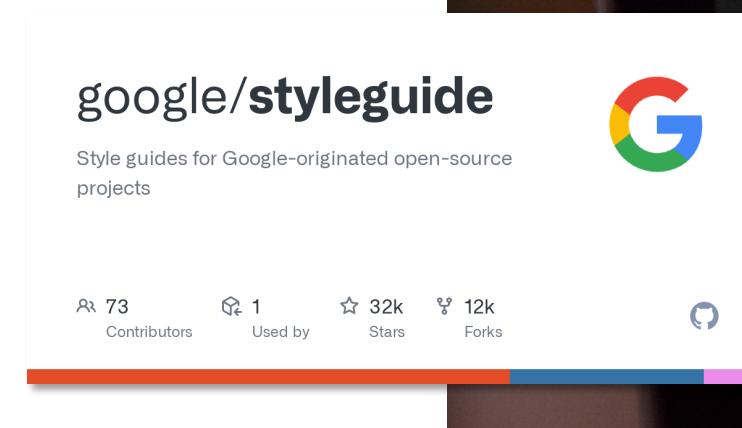
Download Gradescope  
for Android

# Participation Activity

- Take out a piece of paper.
- Write down one pro and one con about trying to understand a new codebase by compiling and building it vs. just reading the code.
- Pair with your neighbor and discuss your answers. Do you agree?
- Share with the class!
- Submit it on Gradescope by the end of class.
  - Under Not Submitted (Assignments), click on August 31 Activity.
  - Take a picture of your paper.
  - Assign the picture to Question 1.
  - Submit.

# Observation: Software is full of patterns

- File structure
- System architecture
- Code structure
- Names
- ...

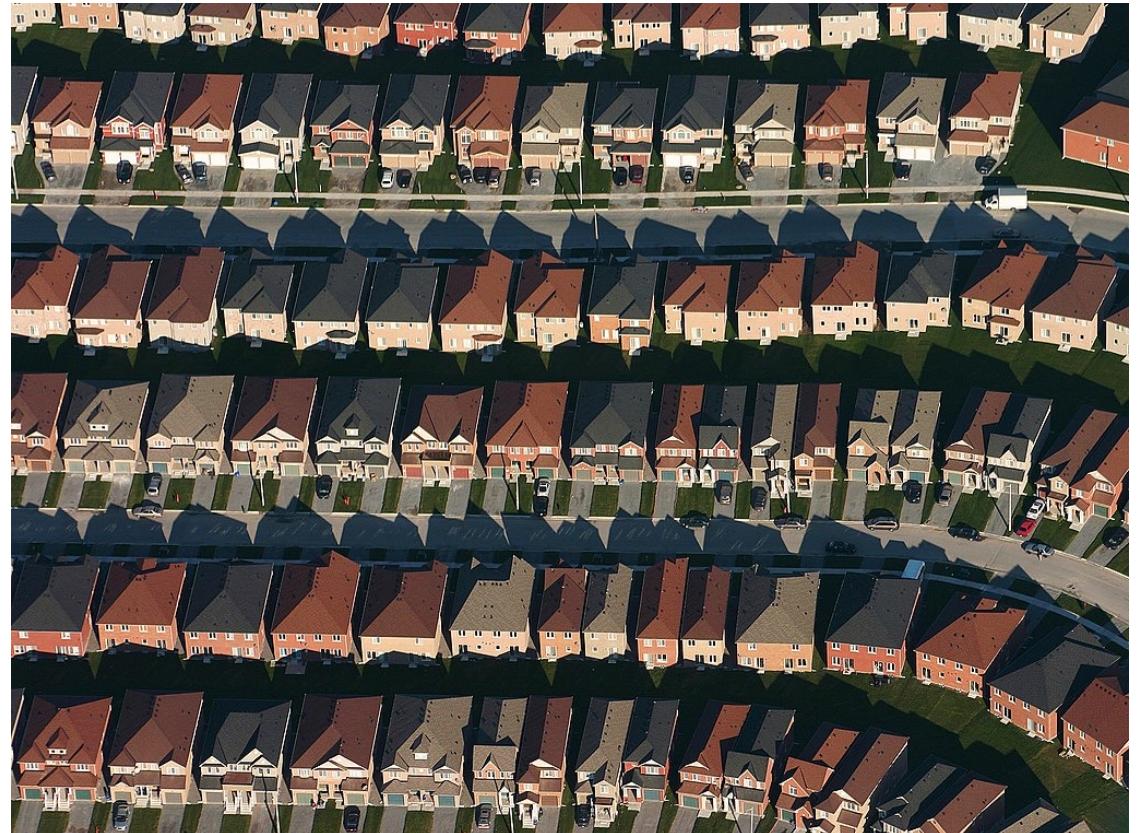


A dark-themed code editor window is shown, displaying a snippet of Python code. The code defines a class 'RequestFingerprinter' with several methods: \_\_init\_\_, from\_settings, request\_seen, and request\_fingerprint. The code uses various Python features like classmethods, file I/O, and logging.

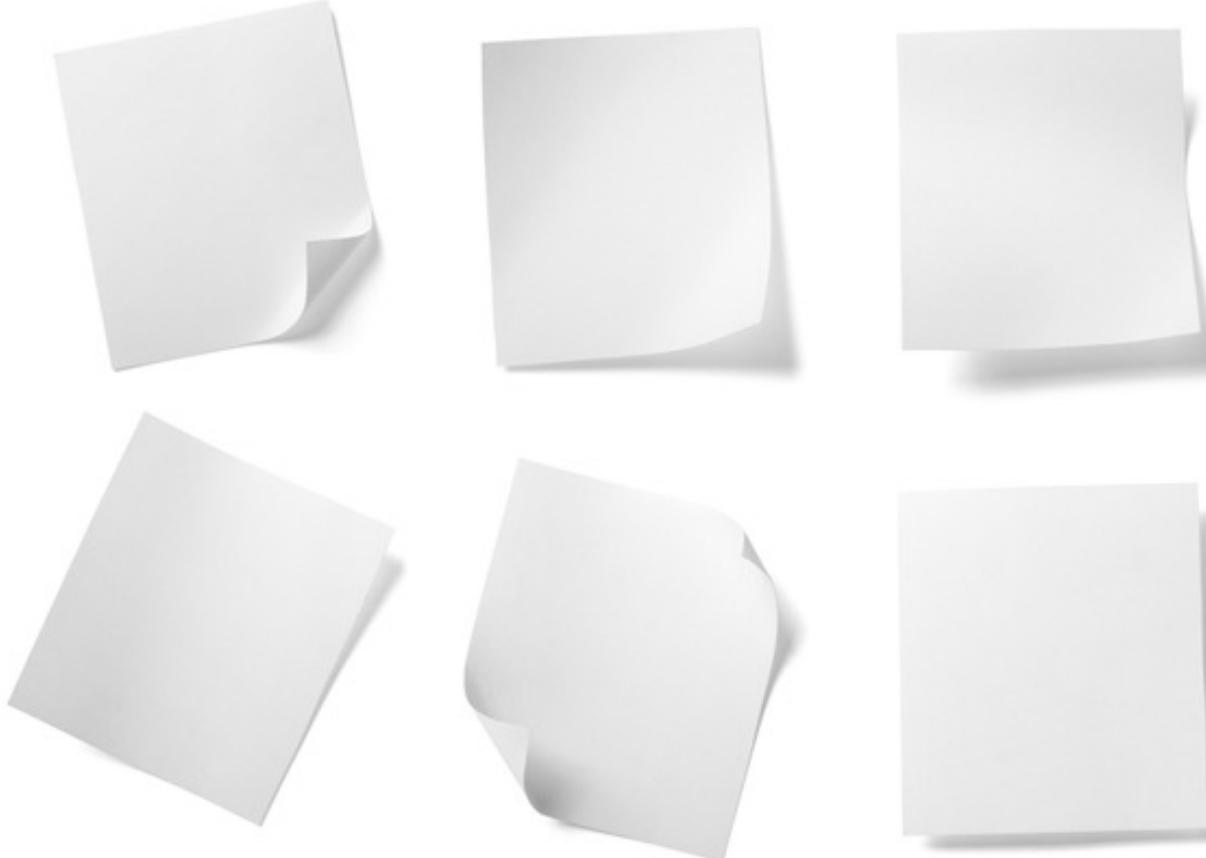
```
31     self.file = None
32     self.fingerprints = set()
33     self.logdups = True
34     self.debug = debug
35     self.logger = logging.getLogger(__name__)
36     if path:
37         self.file = open(os.path.join(path, 'request_log'))
38         self.file.seek(0)
39         self.fingerprints.update([l.strip() for l in self.file])
40
41
42     @classmethod
43     def from_settings(cls, settings):
44         debug = settings.getbool('DEBUG')
45         return cls(job_dir(settings), debug)
46
47     def request_seen(self, request):
48         fp = self.request_fingerprint(request)
49         if fp in self.fingerprints:
50             return True
51         self.fingerprints.add(fp)
52         if self.file:
53             self.file.write(fp + os.linesep)
54
55     def request_fingerprint(self, request):
56         return request_fingerprint(request)
```

# Observation: Software is massively redundant

- There's always something to copy/use as a starting point!



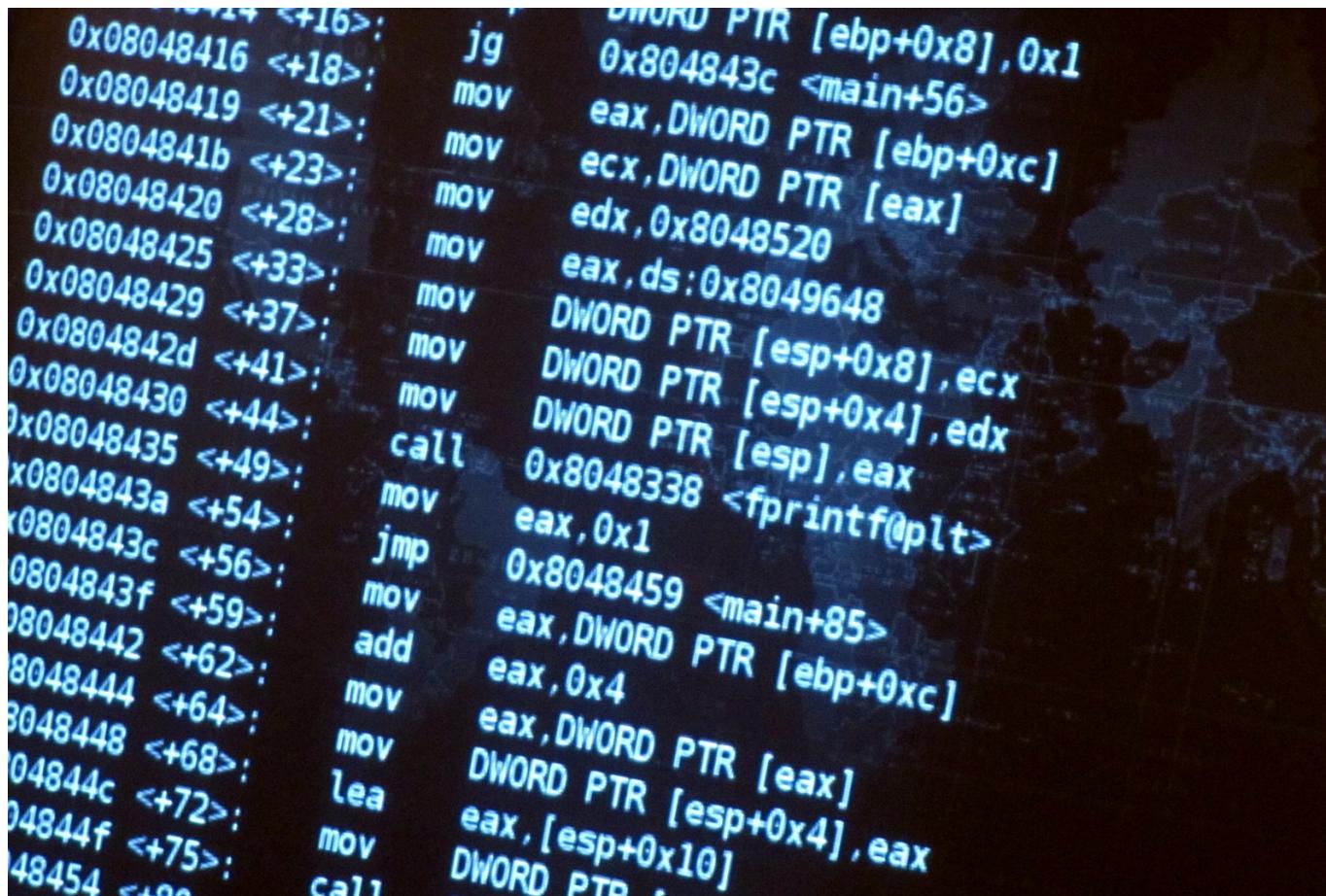
# Observation: Code must run to do stuff!



# Observation: If code runs, it must have a beginning...



# Observation: If code runs, it must exist...



```
0x08048414 <+16>:    jg    DWORD PTR [ebp+0x8],0x1
0x08048416 <+18>:    mov   0x804843c <main+56>
0x08048419 <+21>:    mov   eax,DWORD PTR [ebp+0xc]
0x0804841b <+23>:    mov   ecx,DWORD PTR [eax]
0x08048420 <+28>:    mov   edx,0x8048520
0x08048425 <+33>:    mov   eax,ds:0x8049648
0x08048429 <+37>:    mov   DWORD PTR [esp+0x8],ecx
0x0804842d <+41>:    mov   DWORD PTR [esp+0x4],edx
0x08048430 <+44>:    mov   DWORD PTR [esp],eax
0x08048435 <+49>:    call  0x8048338 <fprintf@plt>
0x0804843a <+54>:    mov   eax,0x1
0x0804843c <+56>:    jmp   0x8048459 <main+85>
0x0804843f <+59>:    mov   eax,DWORD PTR [ebp+0xc]
0x08048442 <+62>:    add   eax,0x4
0x08048444 <+64>:    mov   eax,DWORD PTR [eax]
0x08048448 <+68>:    mov   DWORD PTR [esp+0x4],eax
0x0804844c <+72>:    lea   eax,[esp+0x10]
0x0804844f <+75>:    mov   DWORD PTR .
0x08048454 <+76>:
```

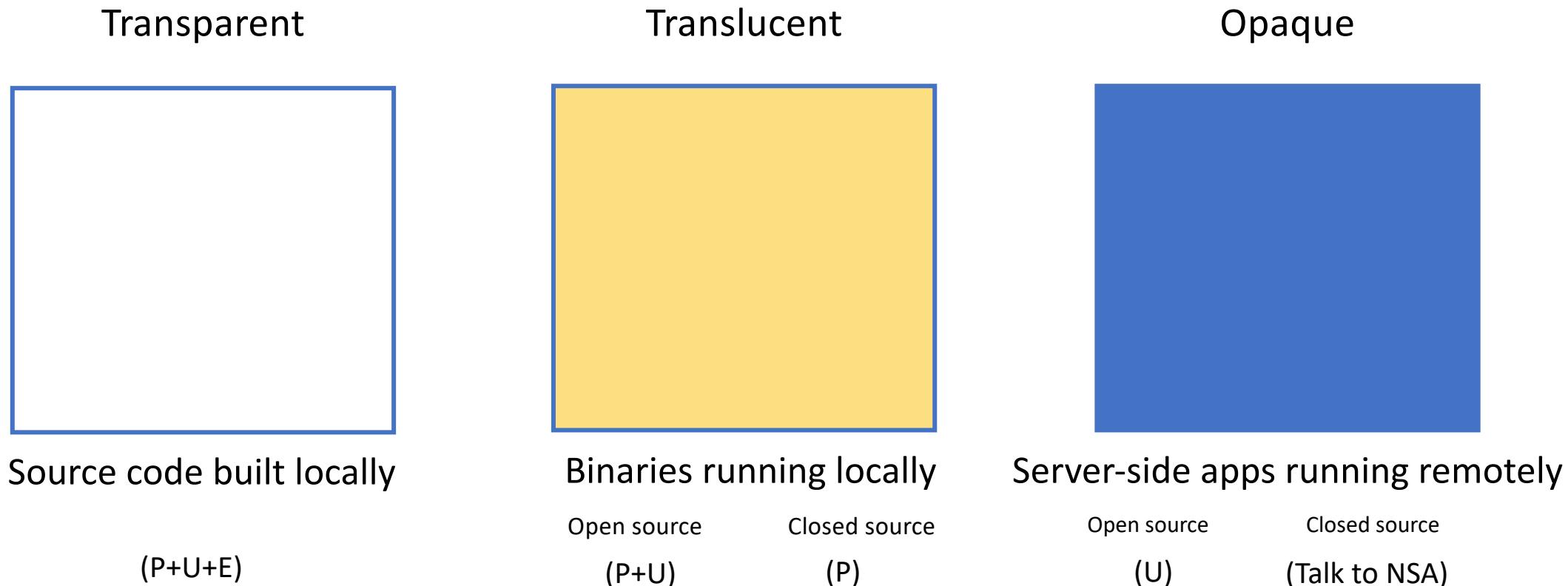
# The Beginning: Entry Points

- Locally installed programs: run cmd, OS launch, I/O events, etc.
- Local applications in dev: build + run, test, deploy (e.g., docker)
- Web apps server-side: Browser sends HTTP request (GET/POST)
- Web apps client-side: Browser runs JavaScript, event handlers

# Code must exist. But where?

- Locally installed programs: run cmd, OS launch, I/O events, etc.
  - Binaries (machine code) on your computer
- Local applications in dev: build + run, test, deploy (e.g., docker)
  - Source code in repository (+ dependencies)
- Web apps server-side: Browser sends HTTP request (e.g., GET, POST)
  - Code runs remotely (you can only observe outputs)
- Web apps client-side: Browser runs JavaScript, event handlers
  - Source code is downloaded and run locally (see: browser dev tools!)

# Can running code be Probed/Understood/Edited?



# Creating a model of unfamiliar code

Source code built  
locally

# Information Gathering

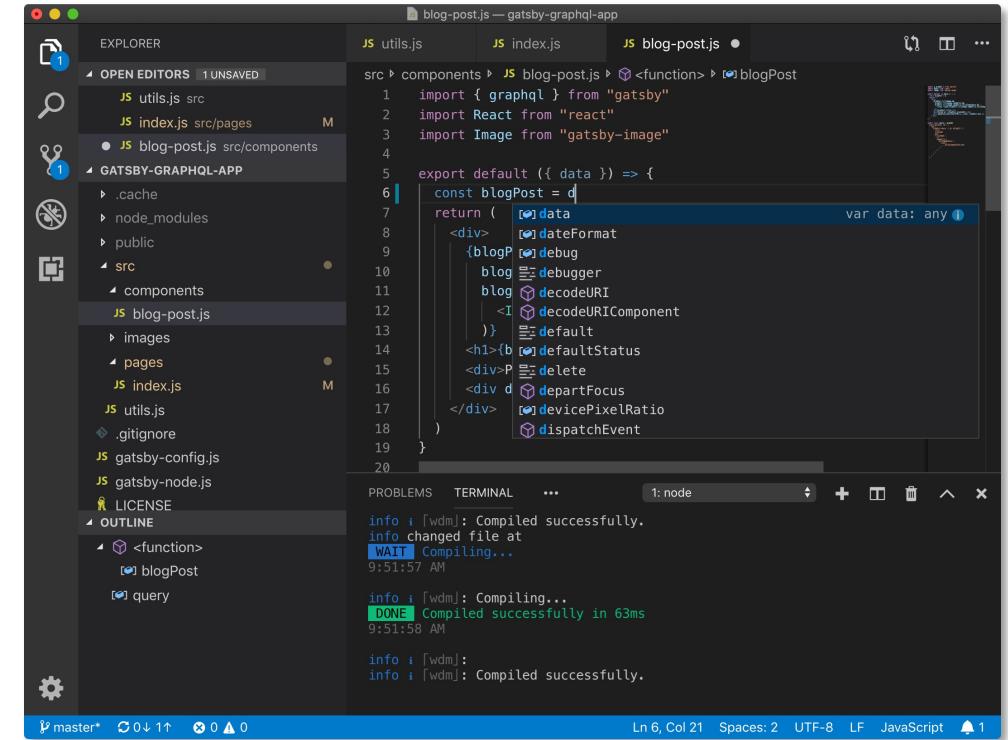
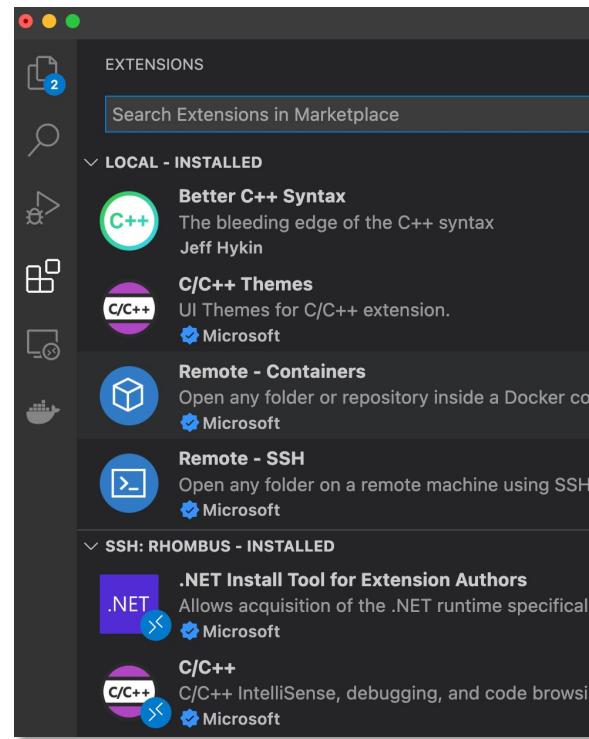
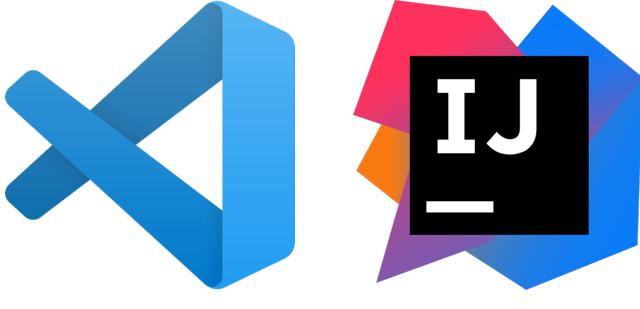
- Basic needs:
  - Code/file search and navigation
  - Code editing (probes)
  - Execution of code, tests
  - Observation of output (observation)
- Many choices here on tools! Depends on circumstance.
  - grep/find/etc. Knowing Unix tools is invaluable
  - A decent IDE
  - Debugger
  - Test frameworks + coverage reports
  - Google (or your favorite web search engine)
  - ChatGPT or LaMA



At the command line: **grep** and **find!**  
(Google for tutorials)

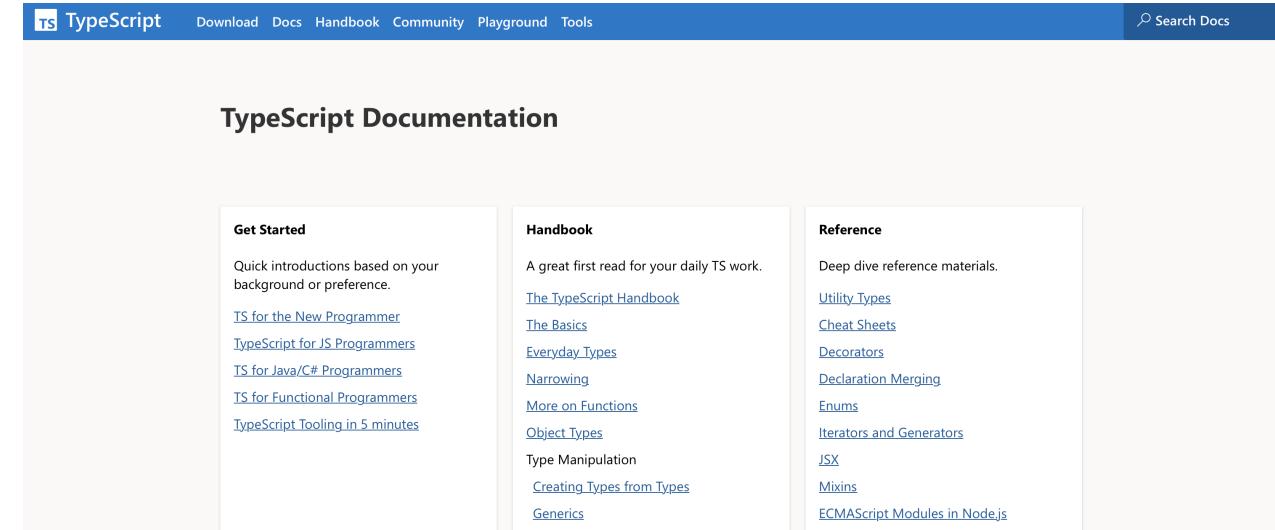
# Static Information Gathering: Use an IDE!

## Real software is too complex to keep in your head



# Consider documentation and tutorials judiciously

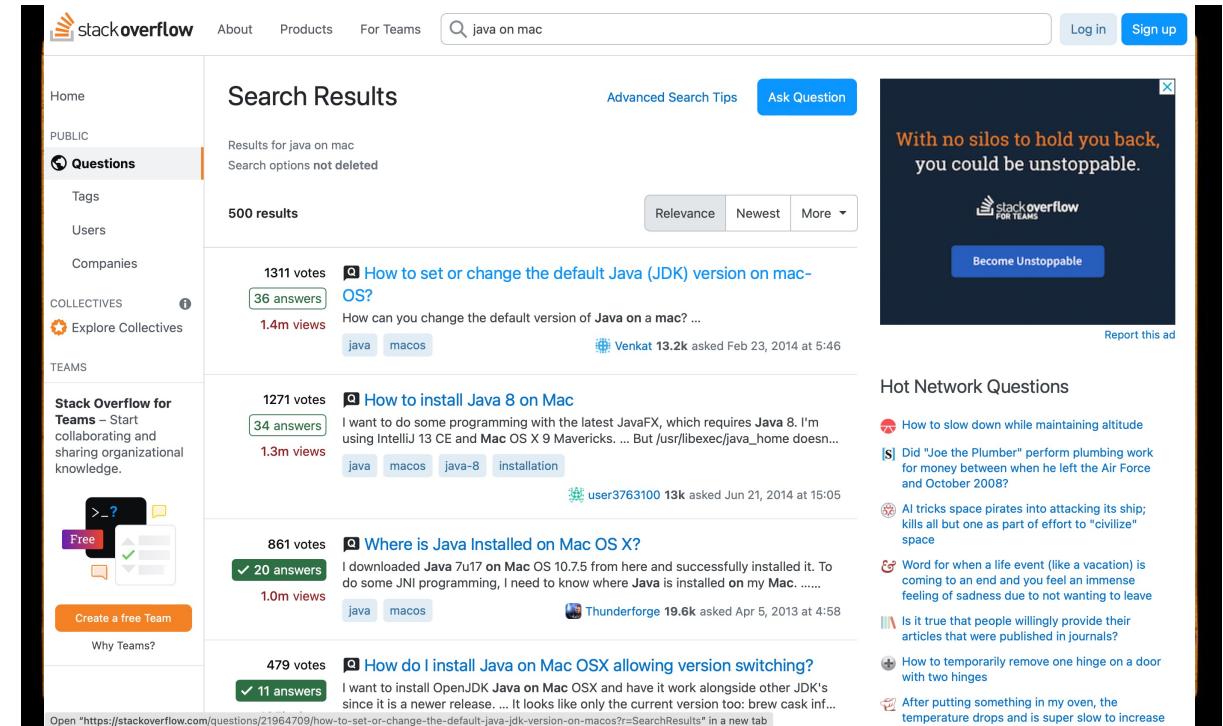
- Great for discovering entry points!
- Can teach you about general structure, architecture (more on this later in the semester)
- Often out of date.
- As you gain experience, you will recognize more of these, and you will immediately know something about how the program works
- Also: discussion boards; issue trackers



The screenshot shows the official TypeScript Documentation website. At the top, there is a blue header bar with the "TypeScript" logo and navigation links for "Download", "Docs", "Handbook", "Community", "Playground", and "Tools". On the far right of the header is a search bar labeled "Search Docs". Below the header, the main title "TypeScript Documentation" is centered. The page is divided into three main sections: "Get Started", "Handbook", and "Reference". The "Get Started" section contains links to "TS for the New Programmer", "TypeScript for JS Programmers", "TS for Java/C# Programmers", "TS for Functional Programmers", and "TypeScript Tooling in 5 minutes". The "Handbook" section contains links to "The TypeScript Handbook", "The Basics", "Everyday Types", "Narrowing", "More on Functions", "Object Types", "Type Manipulation", "Creating Types from Types", and "Generics". The "Reference" section contains links to "Utility Types", "Cheat Sheets", "Decorators", "Declaration Merging", "Enums", "Iterators and Generators", "JSX", "Mixins", and "ECMAScript Modules in Node.js".

# Discussion Boards and Issue Trackers

- Software is written by people.
- How can we talk to them?
- Fortunately, they probably aren't dead.
- So, you can report problems on GitHub.
- Or, ask them questions on StackOverflow.



# Dynamic Information Gathering

## Change helps to inform and refine mental models

- Build it.
- Run it.
- Change it.
- Run it again.
- How did the behavior change?



# Probes: Observe, control or “lightly” manipulate execution

- `print("this code is running!")`
- Structured logging
- Debuggers
  - Breakpoint, eval, step through / step over
  - (Some tools even support remote debugging)
- Delete debugging
- Chrome Developer Tools

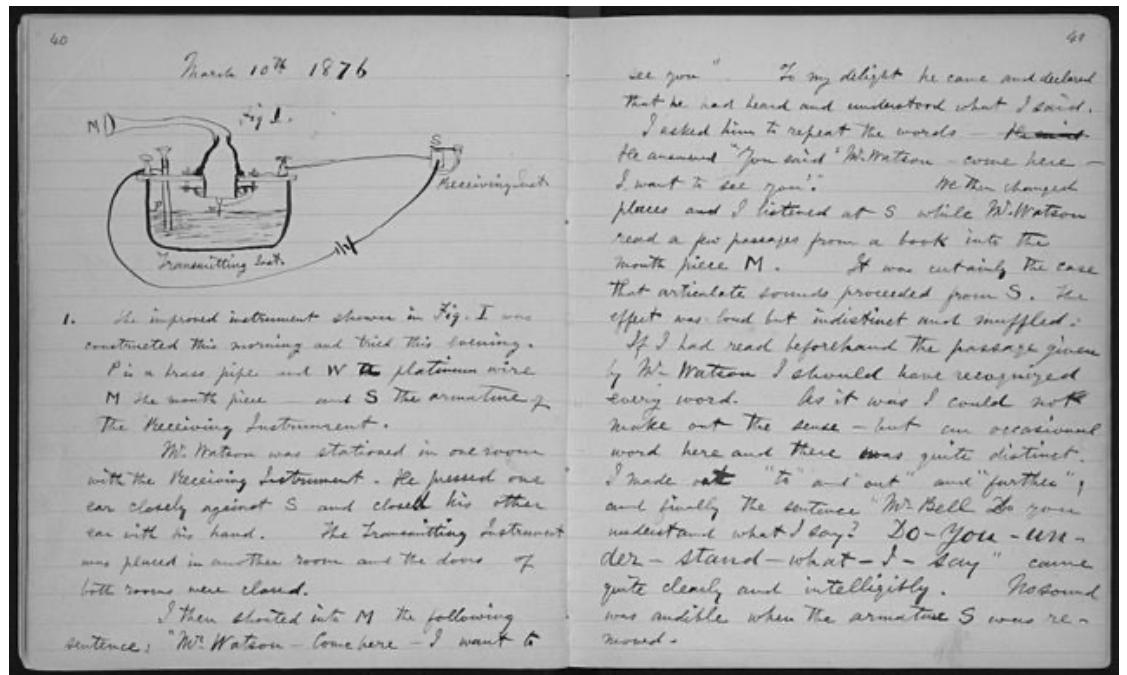
The screenshot shows the Visual Studio Code interface during a Java debugging session. The code editor displays `Owner.java` with several methods for managing owner information. The `VARIABLES` panel on the left shows a local variable `this` of type `Owner`. A context menu is open over the `telephone` field, with the option `Break When Value Changes` highlighted by a yellow arrow. The `CALL STACK` panel shows multiple threads running, with one thread for `Owner.setTelephone(String)` currently paused at line 85:1. The `BREAKPOINTS` panel at the bottom shows two breakpoints set: one for the `Owner.telephone` field and another for the `OwnerController.java` file. Yellow arrows point from the text labels in the previous slide to these specific elements in the debugger interface.

# Step 0: Sanity check basic model + hypotheses

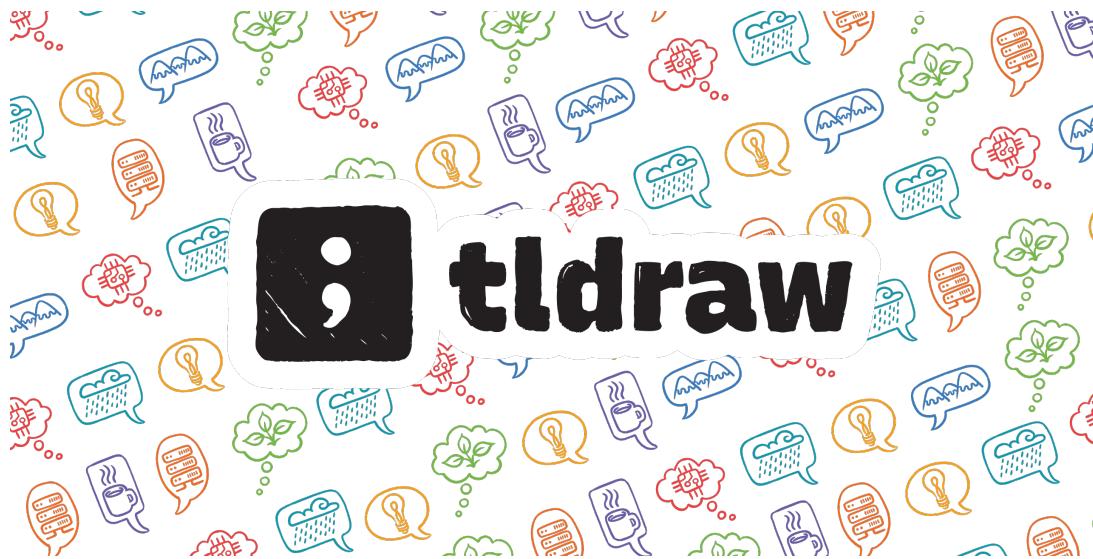
- Confirm that you can build and run the code.
  - Ideally both using the tests provided, and by hand.
- Confirm that the code you are running is the code you built
- Confirm that you can make an externally visible change
- How? Where? Starting points:
  - Run an existing test, change it
  - Write a new test
  - Change the code, write or rerun a test that should notice the change
- Ask someone for help

# Document and share your findings!

- Update README and docs
    - Or better: use a Developer Wiki
    - Use [Mermaid](#) for diagrams
  - Screencast on Twitch
  - Collaborate with others
  - Include negative results, too!



# Let's try some of these techniques again...



<https://github.com/tldraw/tldraw>