Software Archaeology

17-313 Spring 2025

Foundations of Software Engineering

https://cmu-313.github.io

Michael Hilton, Austin Henley, and Nadia Nahar

Administrivia

- Part (b) is due Thursday, Jan 23rd, 11:59pm.
- If you haven't: **PLEASE FILL OUT TEAMWORK SURVEY!**
- Get started early, ask for help, and check the #technical-support channel; chances are your questions have been asked by others!

Smoking Section

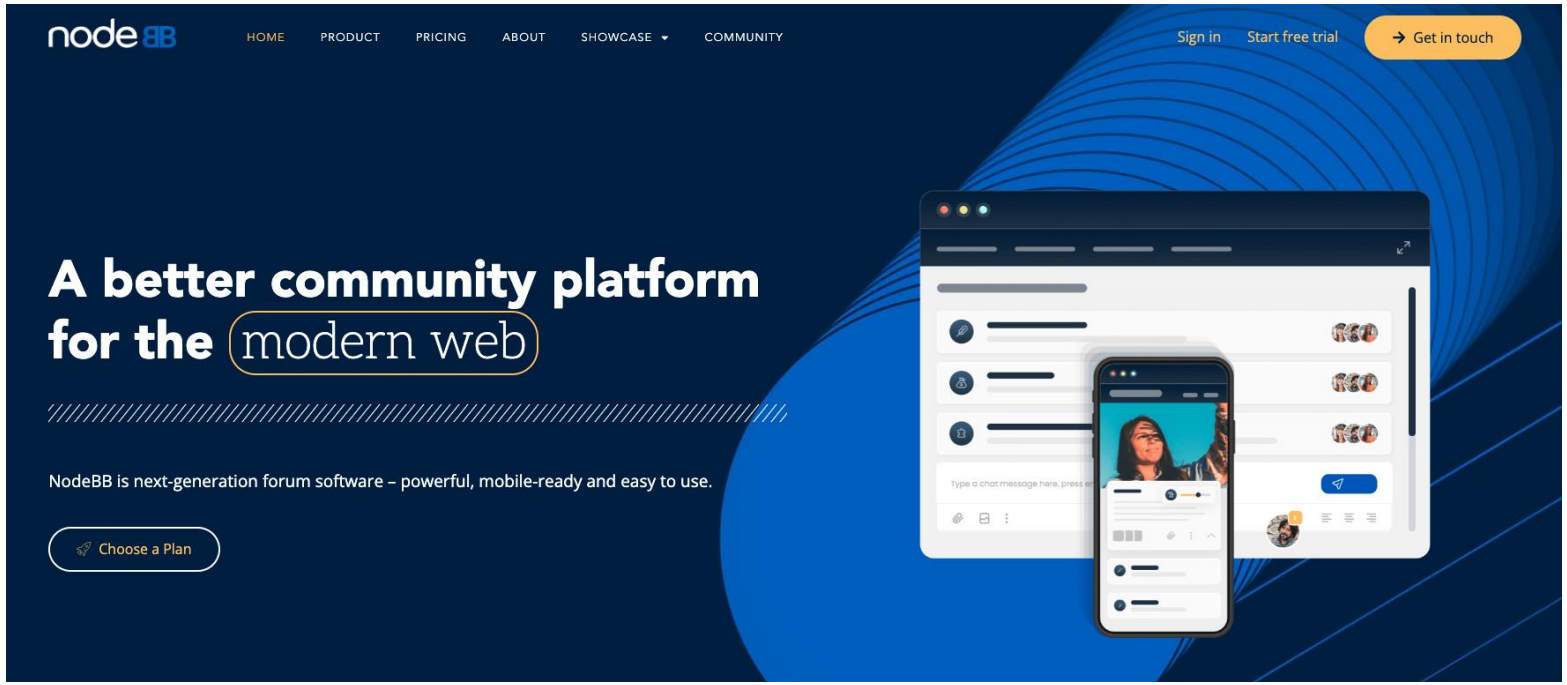
- Last full row

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!

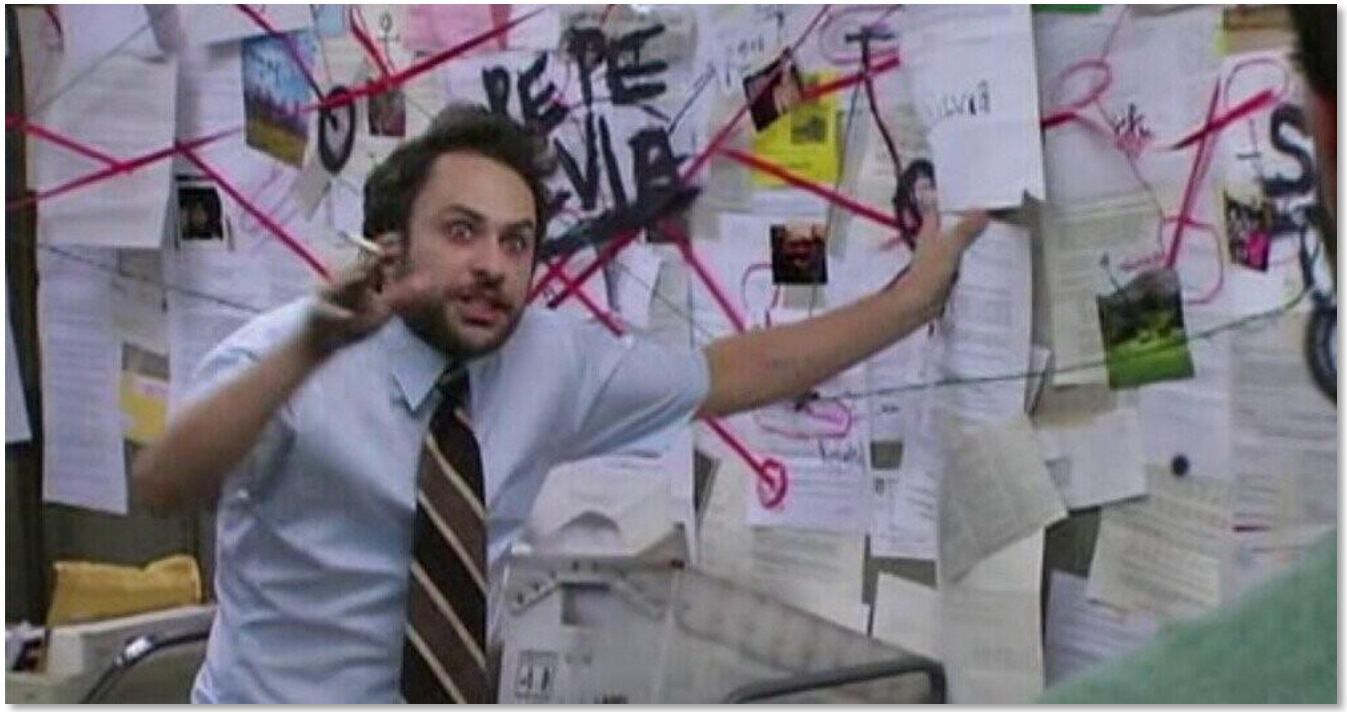


Participation Activity—Part 1

- Take out a piece of paper (or ask for one).
- Write down the **challenges you've faced trying to understand someone else's code**.
- Pair with your neighbor and discuss your answers. Do you agree?
- Share with the class!
- Write your own andrewID on the paper, leave it at the end of class.

You will never understand the entire system!

Challenge: How do I tackle this codebase?



Participation Activity—Part 2

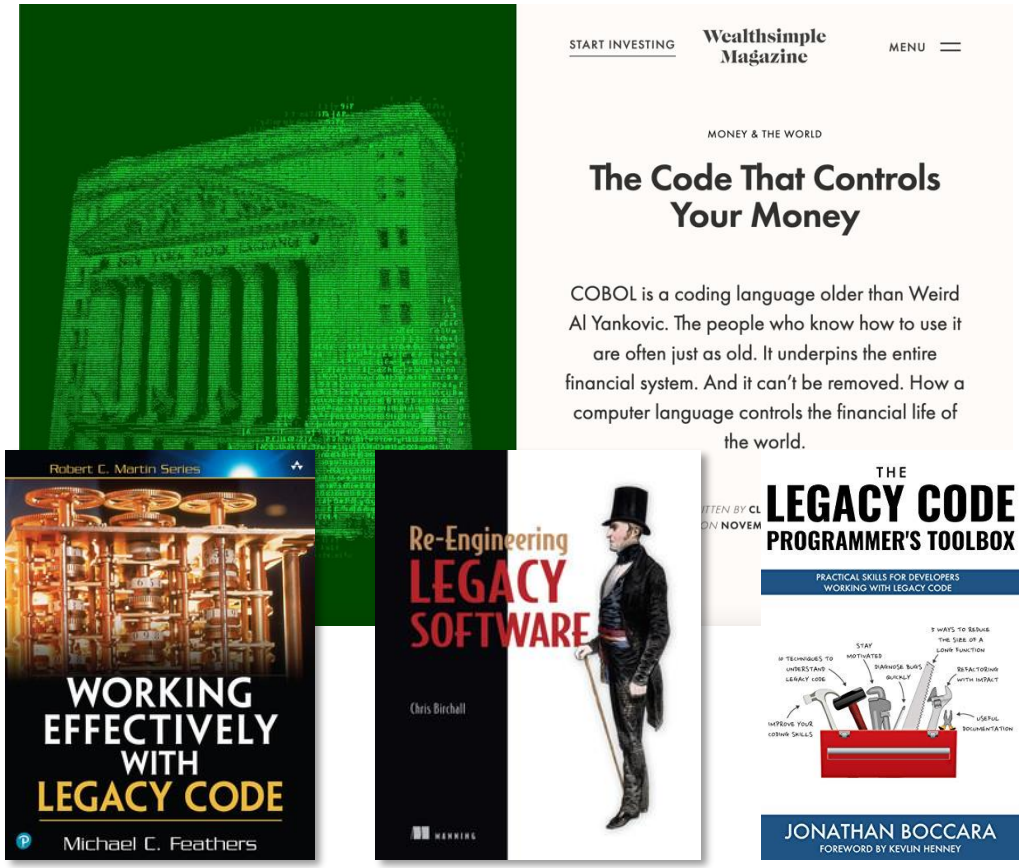
- Write down **strategies to understand a large codebase that is unfamiliar to you**.

Challenge: How do I tackle this codebase?

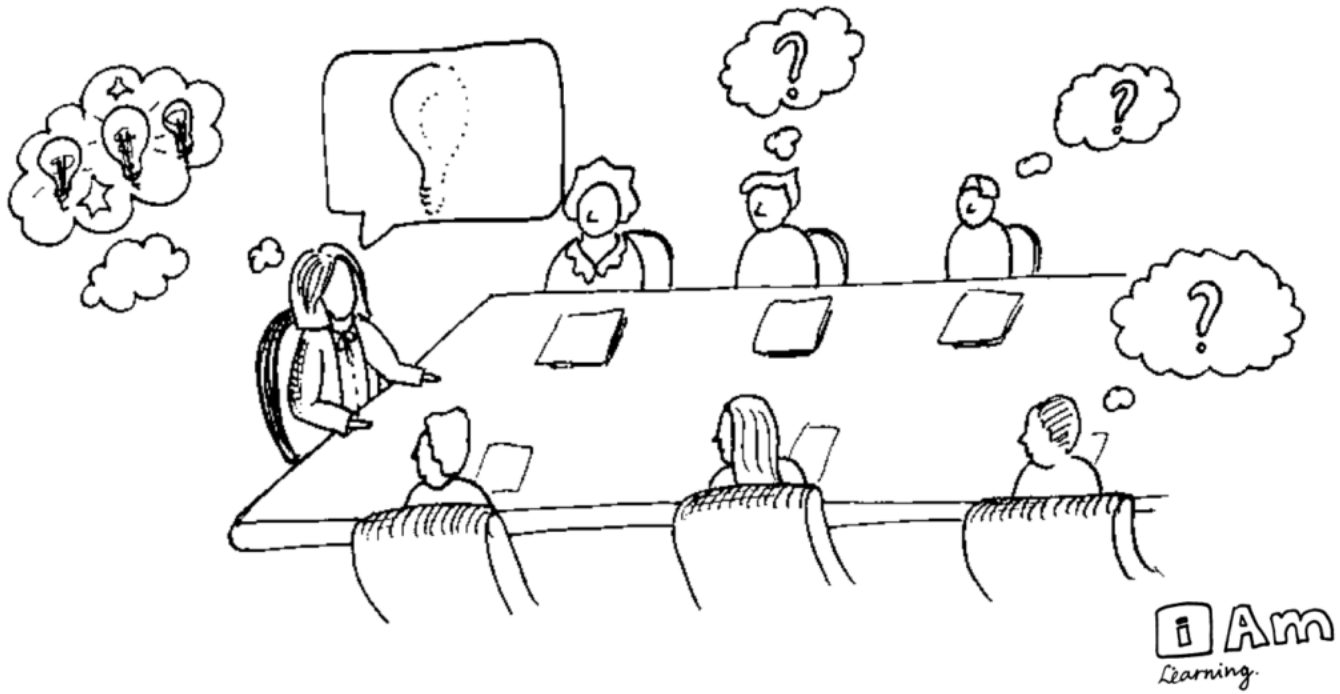
- Leverage your previous experiences (languages, technologies, patterns)
- Consult documentation, whitepapers
- Talk to experts, code owners
- Follow best practices to build a working model of the system

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 Boccara. 2019.

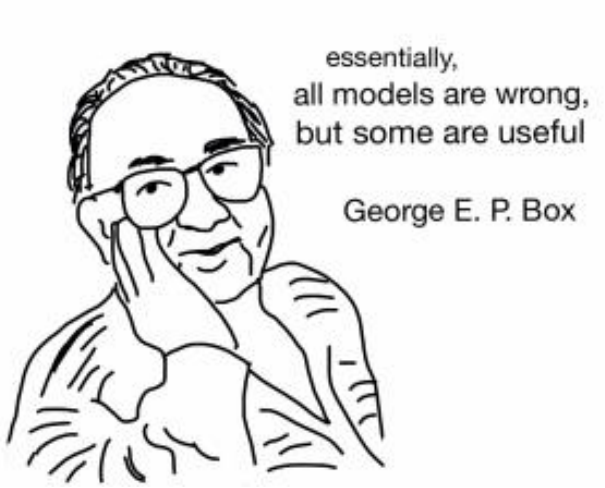


Why? Because of Tacit Knowledge

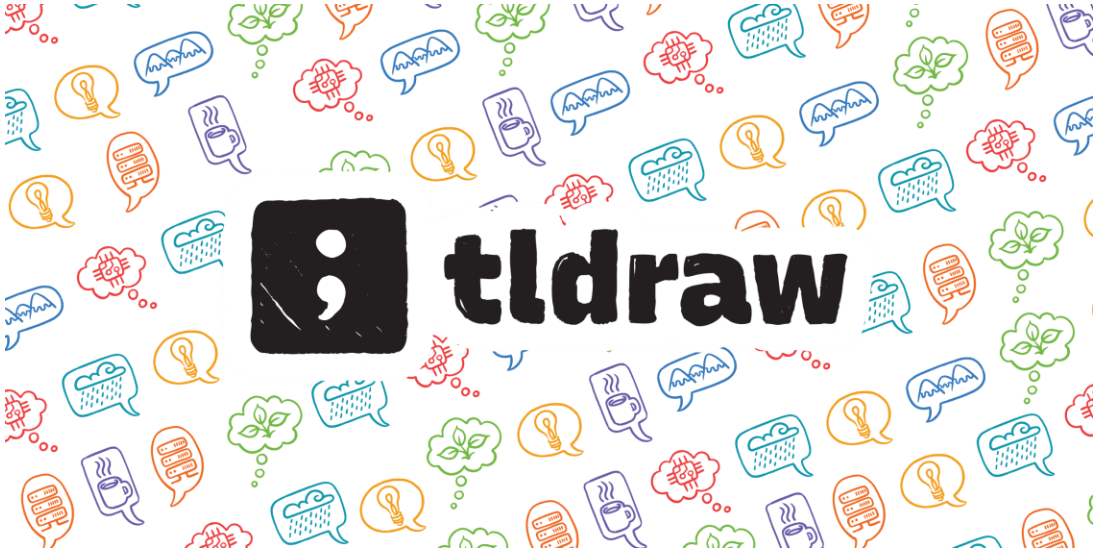


Today: How to tackle codebases

- 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)
- Observation, probes, and hypothesis testing
  - Helpful tools and techniques!



Live Demonstration: tldraw



https://github.com/tldraw/tldraw

Participation Activity—Part 3

- Write down **what you would do if you wanted to modify the Duplicate functionality**.

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?
- Traceability - Attach a debugger
  - View Source
  - Find the logs.
  - Search for constants (strings, colors, weird integers (#DEADBEEF))

My experiences (headaches) at companies

- Documentation was **ALWAYS** out of date—often the core devs didn't know
- Had to ask someone to ask someone to help me get the project building (i.e., sit beside me for hours)
- Better take notes… not unusual to break something and need to do it all again
- Often the authors are no longer there
- So many design decisions are never written down, or they are trapped in old Jira tickets, commit messages, and emails

Program comprehension strategies

Novice

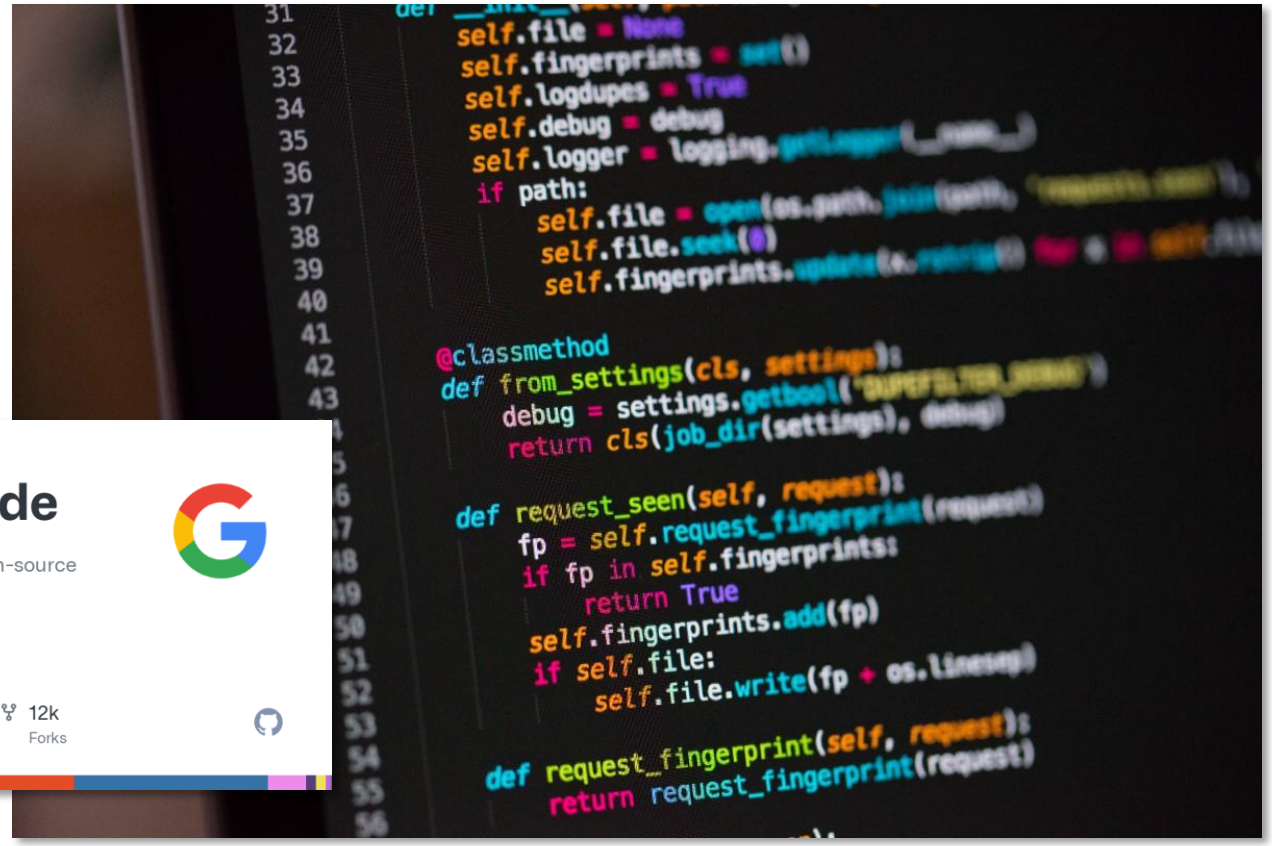
- Reads code line by line
- Revisits same code repeatedly
- Trial and error
- Only tests “happy path”

Expert

- “Top down”
- Recognizes patterns
- Forms hypotheses
- Checks up/downstream consequences

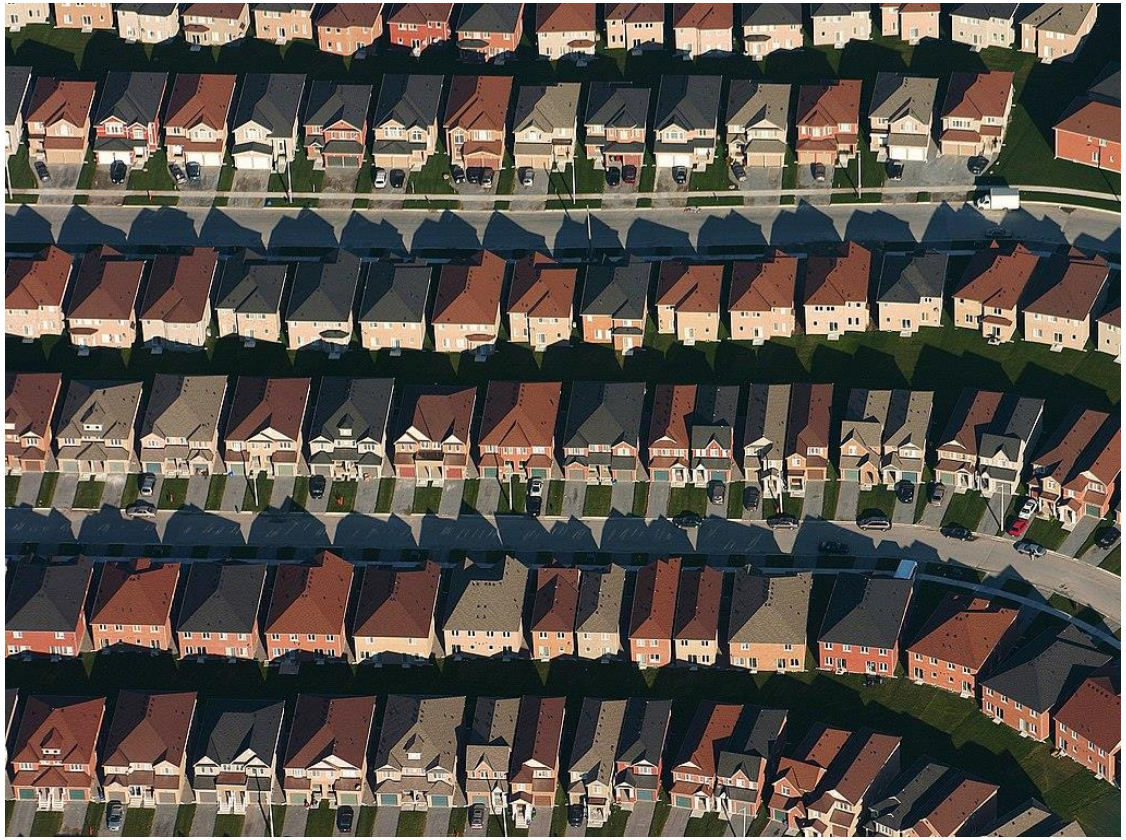
Observation: Software is full of patterns

- File structure
- System architecture
- Code structure
- Names
- …

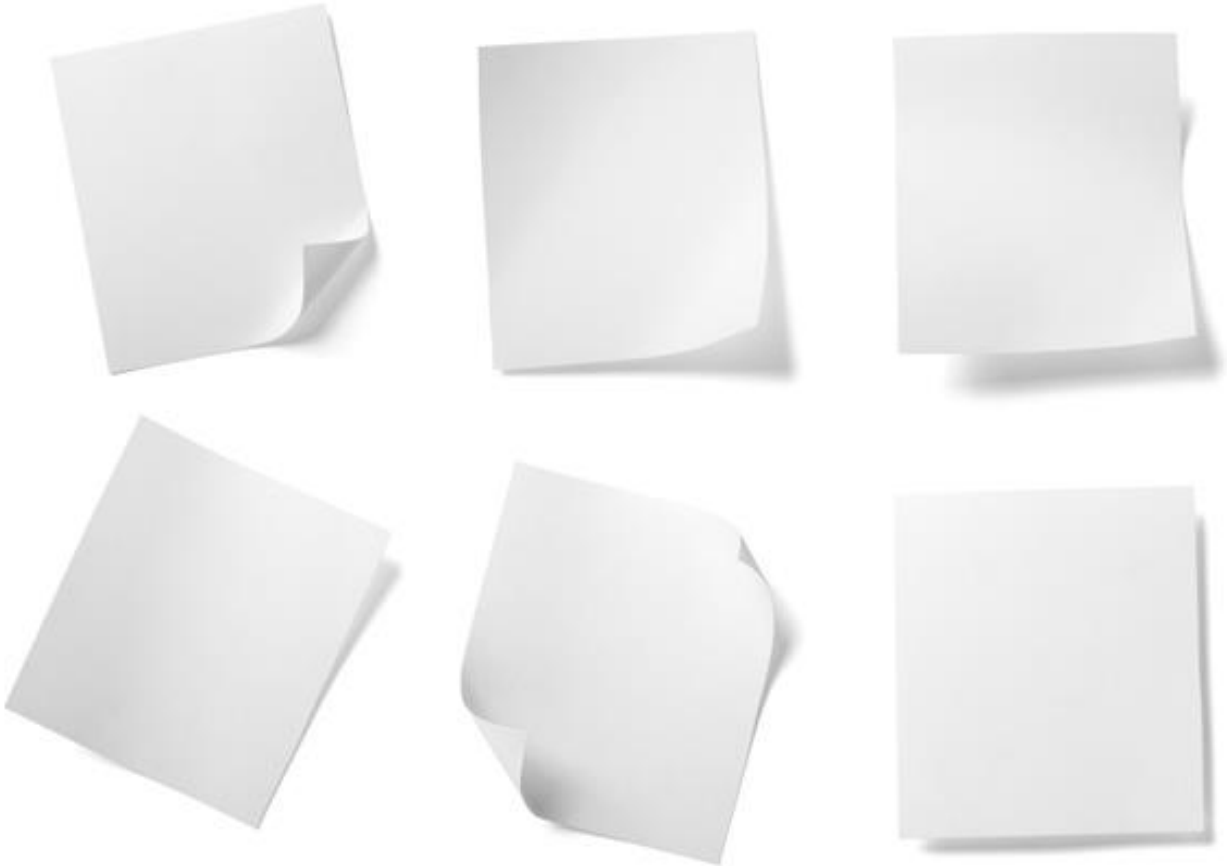


Observation: Software is massively redundant

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



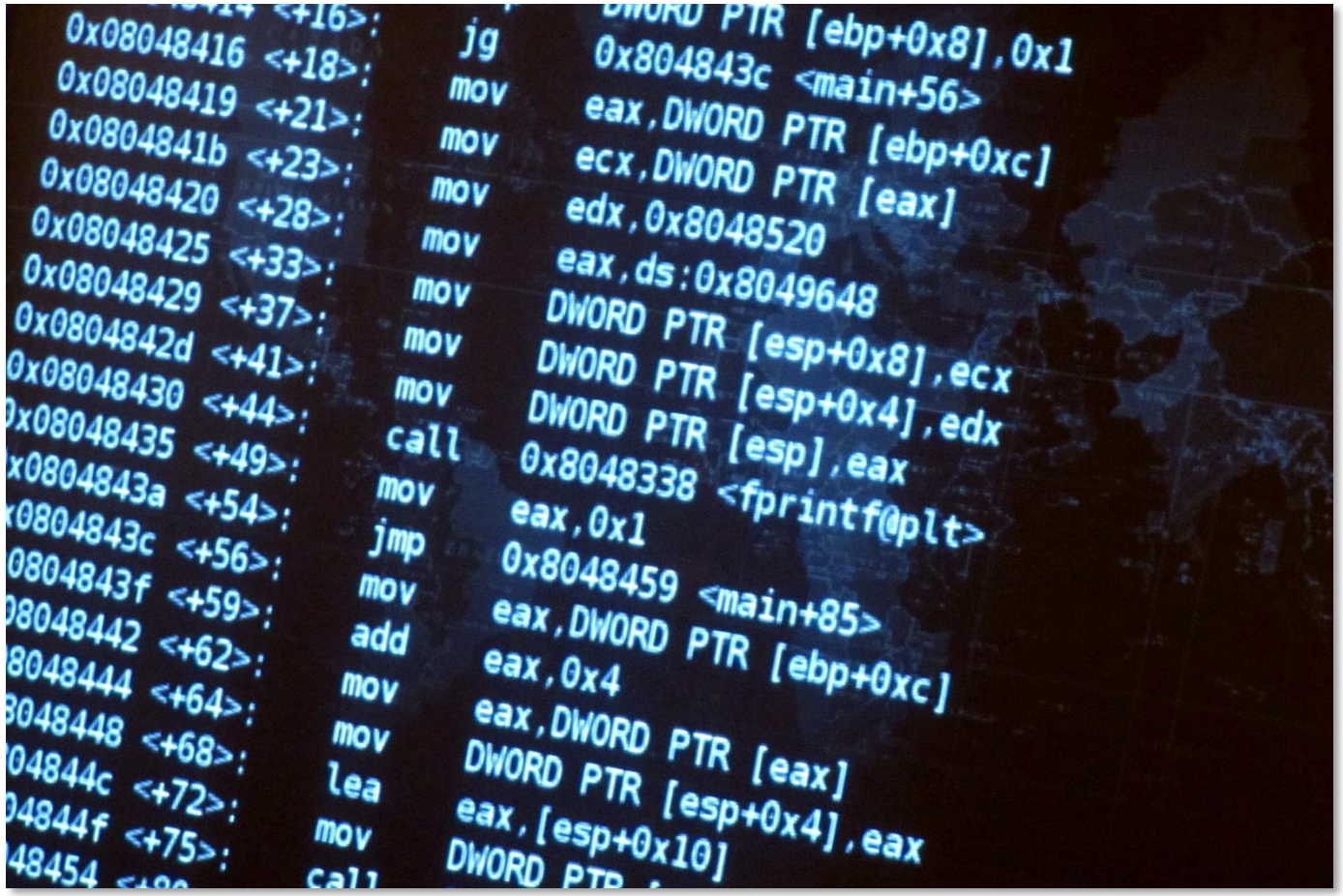
Observation: Code must run to do stuff!



Ask me about the 11,000-line file

Observation: If code runs, it must have a beginning…

Observation: If code runs, it must exist…



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!)

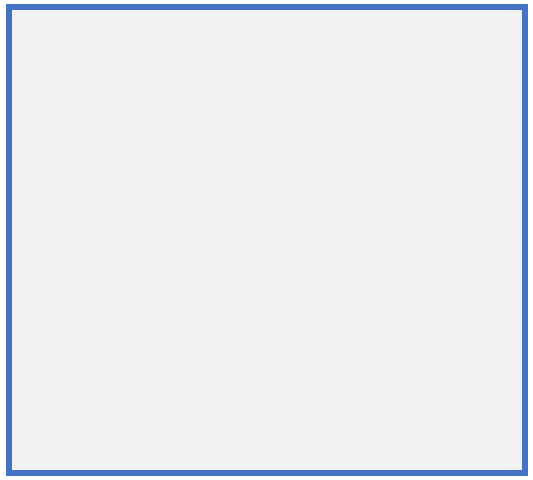
Can running code be Probed/Understood/Edited?

White-box

Source code built locally

(P+U+E)

Grey-box



Binaries running locally

Open source (P+U)

Closed source (P)

Black-box

Server-side apps running remotely

Open source (U)

Closed source (Talk to NSA)

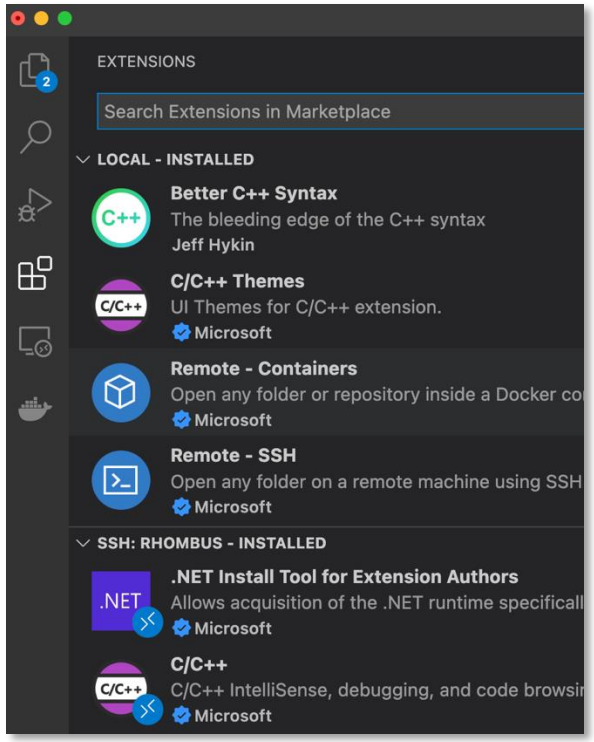
Creating a model of unfamiliar code

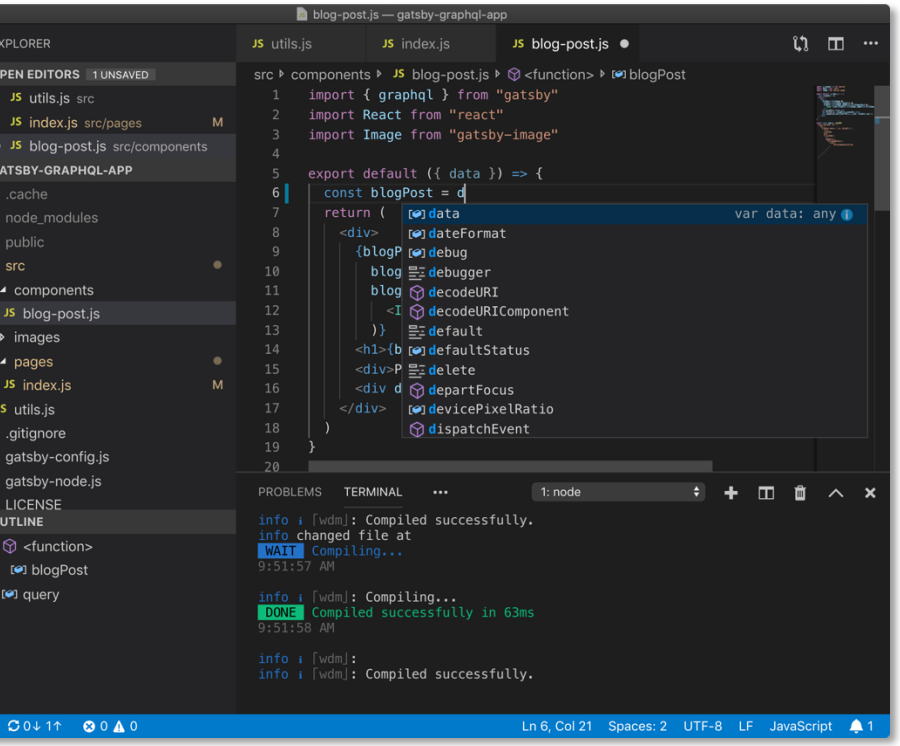
Source code built locally

Information Gathering

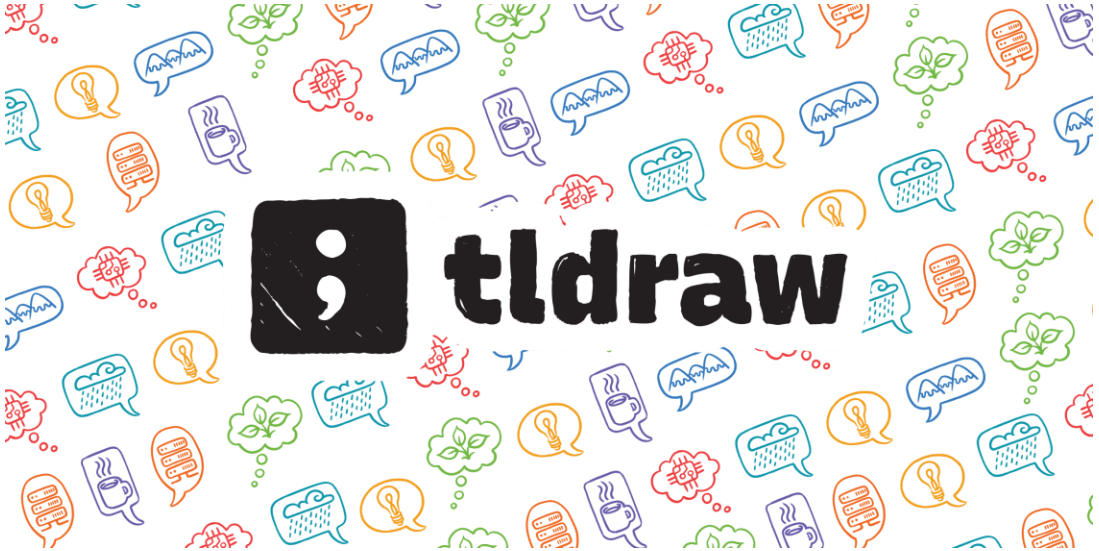
- Basic needs:
  - Code/file search and navigation
  - Code editing (probes)
  - Execution of code, tests
  - Observation of output (observation)
- Many choices here on tools! Depends on circumstance.
  - grep/find/etc. Knowing Unix tools is invaluable
  - A decent IDE
  - Debugger
  - Test frameworks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

Static Information Gathering: Use an IDE!
Real software is too complex to keep in your head





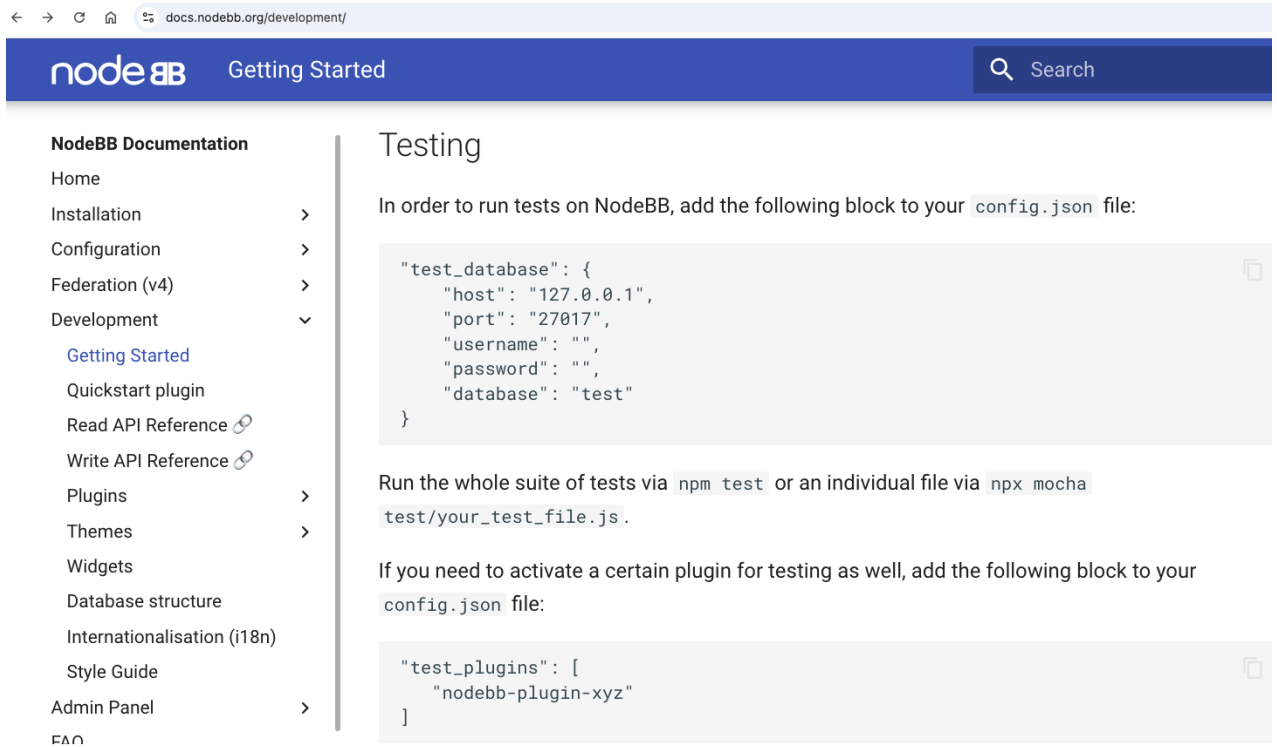
Live Demonstration: tldraw



https://github.com/tldraw/tldraw

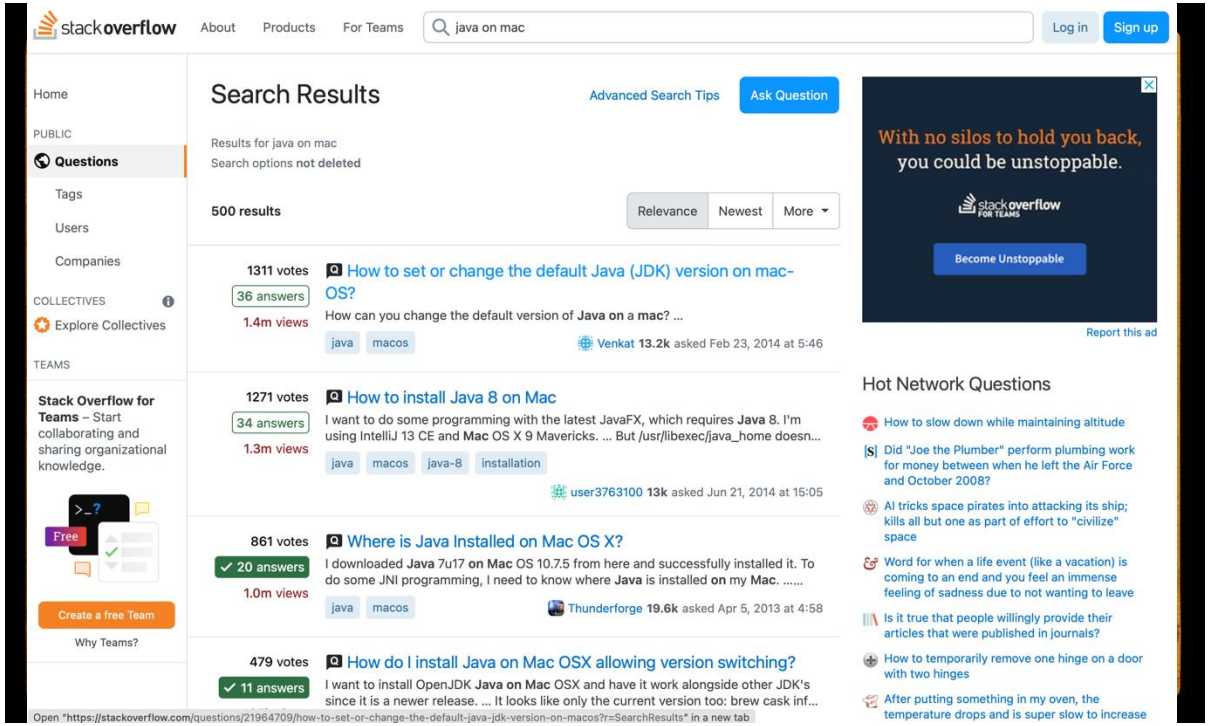
Consider documentation and tutorials judiciously

- 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



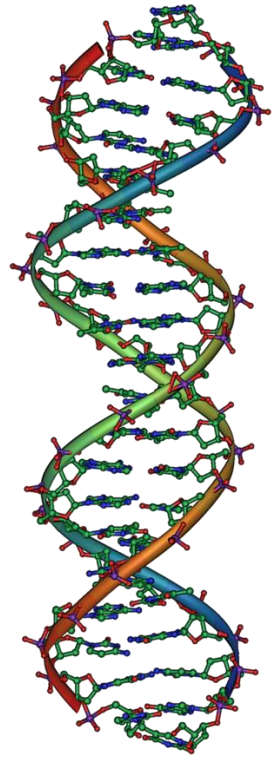
Discussion Boards and Issue Trackers

- 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.



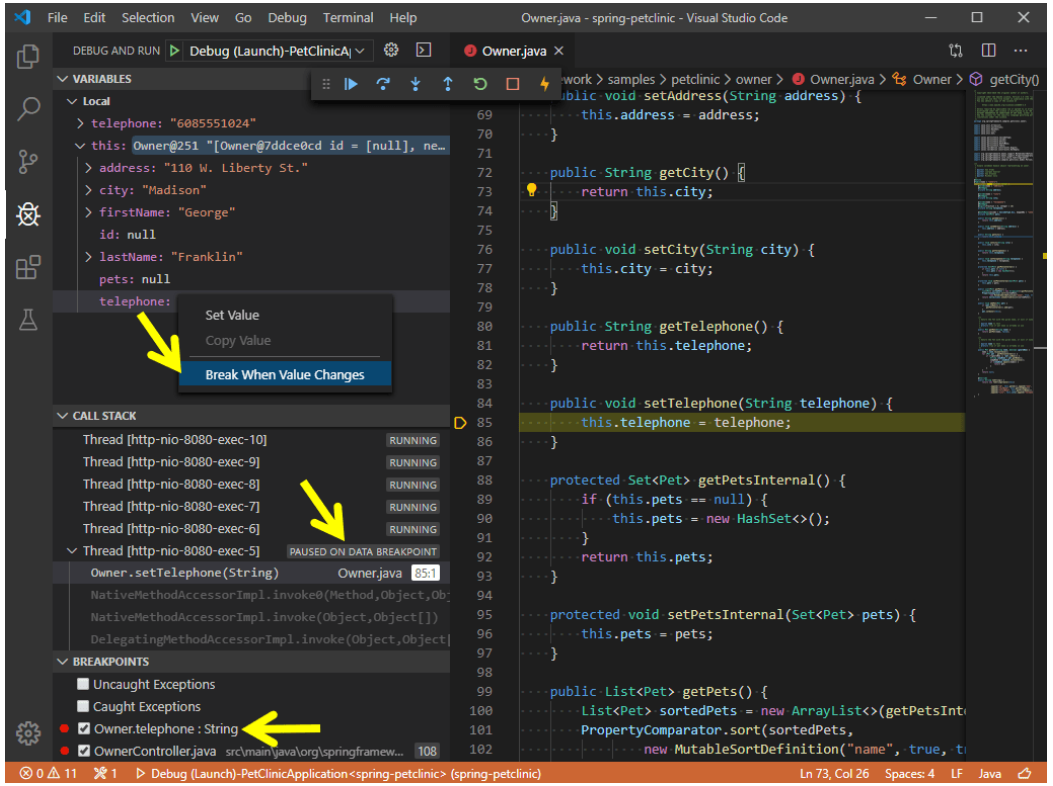
Dynamic Information Gathering
Change helps to inform and refine mental models

- 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

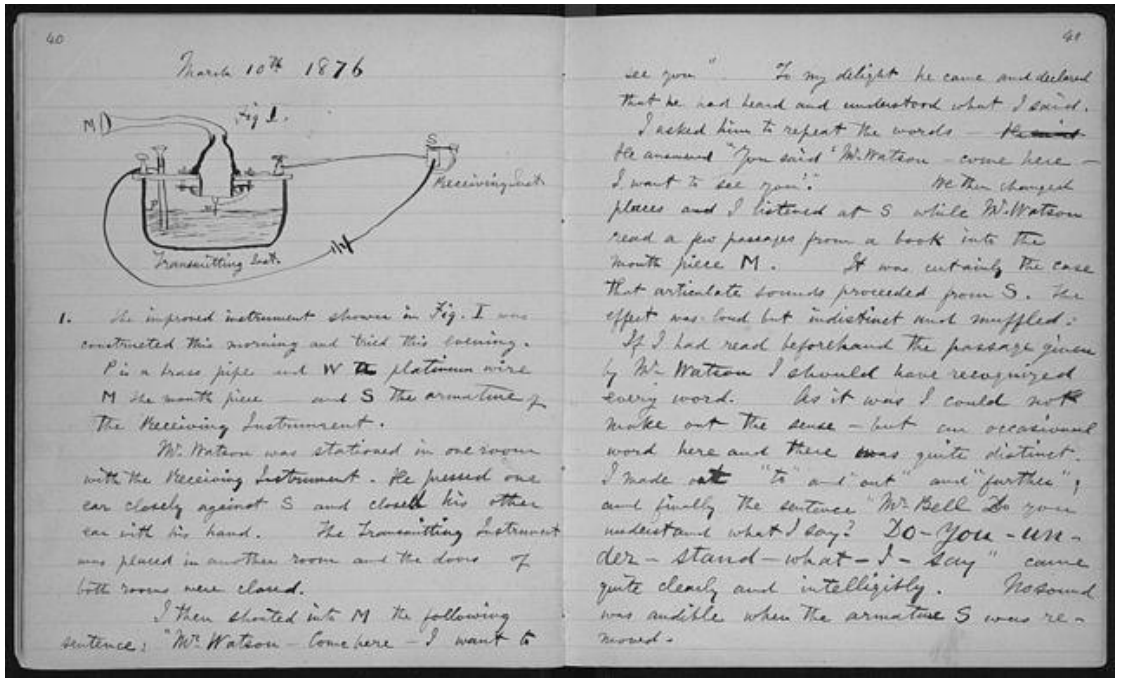


Step 0: Sanity check basic model + hypotheses

- 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!



March 10th 1876

Fig 1.

Receiving Instrument

Transmitting Instrument

1. The improved instrument shown in Fig. I was constructed this morning and tried this evening. P is a brass pipe and W the platinum wire M the mouth piece and S the armature of the Receiving Instrument.

Mr Watson was stationed in one room with the Receiving Instrument. He pressed one ear closely against S and closed his other ear with his hand. The transmitting instrument was placed in another room and the door of both rooms were closed.

I then shouted into M the following sentence: "Mr Watson – Come here – I want to see you" To my delight he came and declared that he had heard and understood what I said. I asked him to repeat the words – He answered "You said 'Mr Watson – come here – I want to see you'." We then changed places and I listened at S while Mr Watson read a few passages from a book into the mouth piece M. It was certainly the case that articulate sounds proceeded from S. The effect was loud but indistinct and muffled. If I had read beforehand the passage given by Mr Watson I should have recognized every word. As it was I could not make out the sense – but an occasional word here and there was quite distinct. I made out "to" and "out" and "further", and finally the sentence "Mr Bell Do you understand what I say? Do-you-un-der-stand-what-I-say" came quite clearly and intelligibly. No sound was audible when the armature S was removed.

Next time…

- Metrics and Measurement