

Refactoring and Brownfield Systems

17-695: Design Patterns

Chris Timperley

Fall 2026

Group Project

Group Project

- You will work in **teams of 3–4** on a **real open source codebase**
 - Teams have now been formed (see “Canvas > People > Groups”)
 - If you didn’t pick a team, we automatically put you into one
- Each individual will tackle **one feature request** on the codebase
 - The features will be shortly visible on the **course mirror** of the codebase
 - You will need to build a high-level understanding of the codebase, explore the design space and consider tradeoffs, and implement your changes while working around and refactoring existing code
- Teams will need to **coordinate and integrate** all of their changes into their own copy of codebase

Group Project: Extend a Real System

- Work in a **team of 3–4** on an existing open-source codebase
- Each person takes **one feature request**: understand the relevant code, explore alternatives, and design and implement a change
- **Improve the existing design** so your feature fits and similar changes become easier
- Combine your team's changes into **one working version**

Group Project: Getting Started

- Teams are listed under **Canvas** → **People** → **Groups**
- Each team is assigned a repository; every repo has an **assigned TA**
- Feature requests will appear as issues on the **course copy** of your repository
- Each person claims one request; the team works together in **its own fork**. **Don't submit your changes to the course copy!**
- Reach out to your TA for questions about the repository and requests

Group Project: Timeline

- **Sept 27:** Explain your proposed design, alternatives, and checks
- **Sept 28:** Review other teams' designs and use their feedback to revise yours
- **Oct 4:** Submit your implementation, evidence, final RFC, and slides
- **Oct 5 & 7:** Explain and defend your decisions 🧑🏻‍🎓

You work as a team, but you are graded individually.

Be ready to explain your own design, code, and evidence.

Sit with your project team

Find your team: Canvas → People → Groups

Project Teams 1–3: Audiobookshelf



audiobookshelf Demo Search..

27 Books Collapse Series All Auth

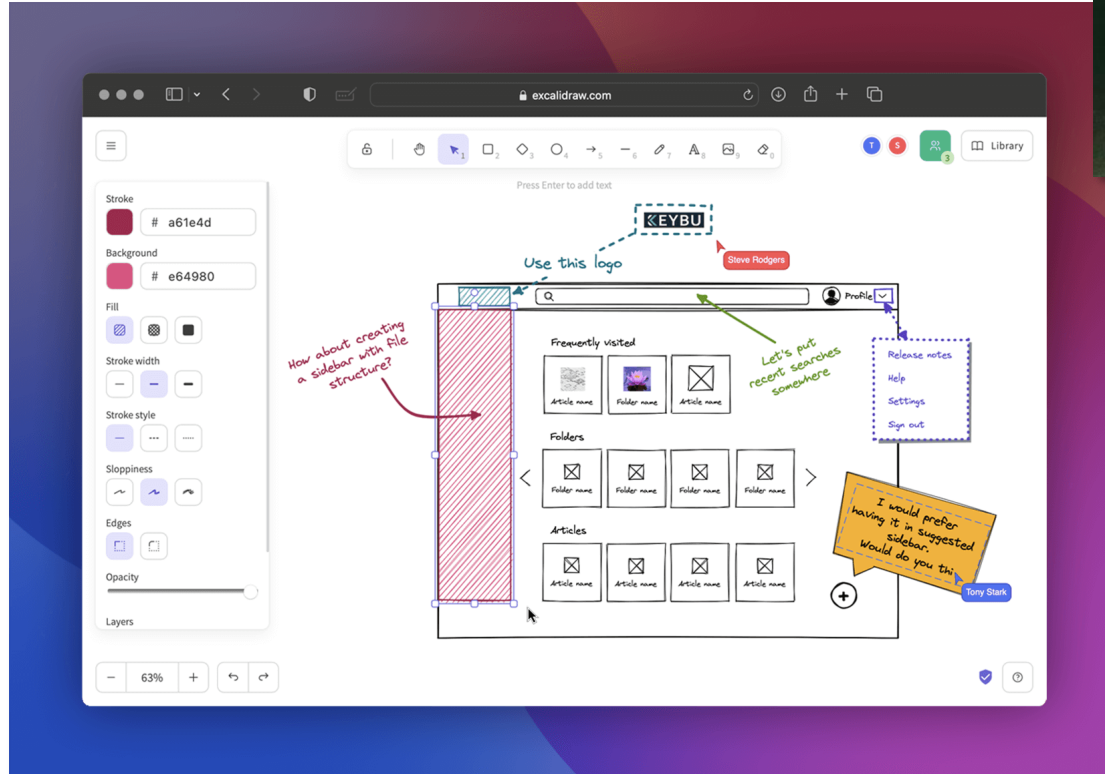
Home Library Series Collections Authors Playlists

Books Grid:

- Robinson Crusoe** by Daniel Defoe
- The Good Soldier** by Ford Madox Ford
- The Brothers Karamazov** by Fyodor Dostoyevsky
- Crime and Punishment** by Fyodor Dostoyevsky
- The War of the Worlds** by H. G. Wells
- Moby Dick** by Herman Melville
- The Call of the Wild** by Jack London
- Grimm's Fairy Tales** by Jacob Grimm, Wilhelm Grimm
- Gulliver's Travels** by Jonathan Swift
- A Journey to the Center of the Earth** by Jules Verne
- Twenty Thousand Leagues under the Sea** by Jules Verne
- The Wonderful Wizard of Oz** by L. Frank Baum

Player: The Call of the Wild
Jack London
© 52:28
7:12 / 14% Chapter 2 -41:08

Project Teams 4–6: Excalidraw



Project Teams 7–9: OpenCode



opencode GitHub Docs Data Zen Go Enterprise [Download](#)

The open source AI coding agent

Free models included or connect any model from any provider, including Claude, GPT, Gemini and more.

curl npm bun brew paru yay

curl -fsSL https://opencode.ai/v2/install | bash

```
Read Users/Desktop/Documents/local/pocketbook/verify.php
List Users/Desktop/Documents/local/pocketbook/resources/js
Read Users/Desktop/Documents/local/pocketbook/app/providers/AuthServiceProvider.php
Read Users/Desktop/Documents/local/pocketbook/resources/js/pages/auth/register.tsx
Read Users/Desktop/Documents/local/pocketbook/app/Http/Controllers/Auth/RegistrationController.php
Read Users/Desktop/Documents/local/pocketbook/database/factories/UserFactory.php
Read Users/Desktop/Documents/local/pocketbook/database/migrations/2025_08_26_100418_add_two_factor_columns_to_users_table.php

ctrl+right, ctrl+left to navigate between subsequent sessions

Now let me read the key files to understand the current structure:
- Read app/Actions/Fortify/CreateNewUser.php
- Read resources/js/pages/auth/register.tsx
- Read database/migrations/2025_08_26_100418_create_users_table.php
- Read tests/Feature/Auth/RegistrationTest.php

Let me also check the user model to see its fillable fields:
- Read app/Models/User.php
- Read database/factories/UserFactory.php

Now I have a complete picture. I'll add a date of birth field that validates the user is at least 18 years old. Let me create a todo list and implement this:
```

Implementing signup age-validation field
Content
25.08.2025
10:04
88.7% agent
 MCP
• Hard Connected
• IntelliJ-Boost Connected
• LSP
• PHP Intelliphense
• ElixirScript
• mason
• Todo
[] Create migration to add date_of_birth column to users table
[] Update user model with date_of_birth field and cast
[] Update CreateNewUser action with date_of_birth validation (the years)
[] Update registration form [register.tsx with date of birth field
[] Update UserFactory with

OC | Homepage button color change in repo ...

Homepage button color change in repo workflow

Find the homepage button and make it blue

I'll search for the homepage button in the codebase.

- * Grep "homepage|home.*button|Home.*button"
- * Grep "Homepage"

Let me search more broadly for buttons and home-related elements:

- * Grep "Home" (18 matches)
- * Glob "**/*.tsx" (100 matches)

I found several "Home" links. Let me look at the most likely candidates - the 404 pages and the header component which have "Home" links:

- Read packages/console/app/src/routes/[...404].tsx
- Read packages/console/app/src/component/header.tsx
- Read packages/enterprise/src/routes/[...404].tsx

I found "Home" buttons/links in multiple locations. Let me ask which one you want to modify:

~ Asking questions...

- * Build · claude-opus-4-5

Build Claude Opus 4.5 OpenCode Zen

esc interrupt ctrl+t variants tab agents ctrl+p commands

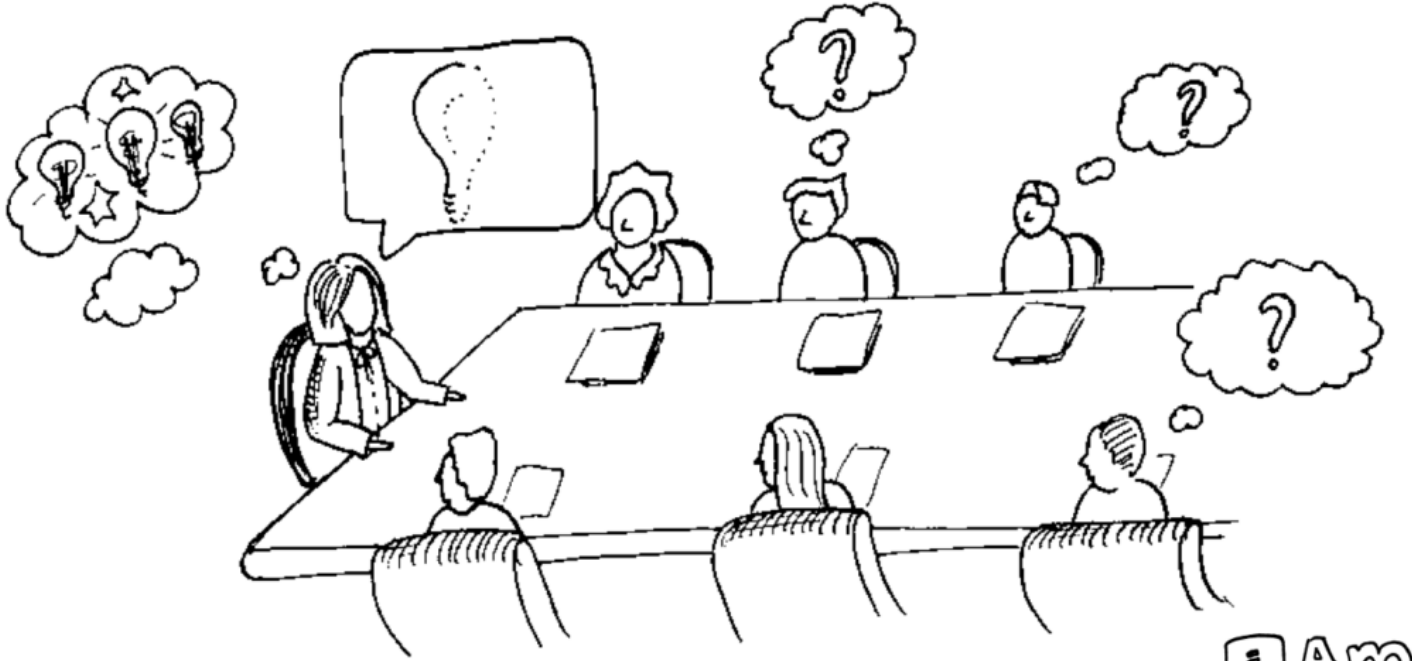
Get Your Project Running

- Clone your **assigned course repository**
- Follow its **development setup instructions** to install dependencies
- **Start the app locally** if installation finishes
- Aim for **one running instance per team** first
- If you get stuck, **record the command and error** for your TA

<https://github.com/orgs/CMU-17695/repositories>

**You will never understand
the entire system!**

Why? Because of Tacit Knowledge



Today: How to Tackle New Codebases

- **Goal:** develop and test a working model about how (part of) a system works
- **Working Model:** an understanding of the pieces of the system (components), and their interactions (connections)
- How to quickly **build, test and refine** models
 - explore various tools, tips, and techniques



essentially,
all models are wrong,
but some are useful

George E. P. Box

Program comprehension strategies

Novice

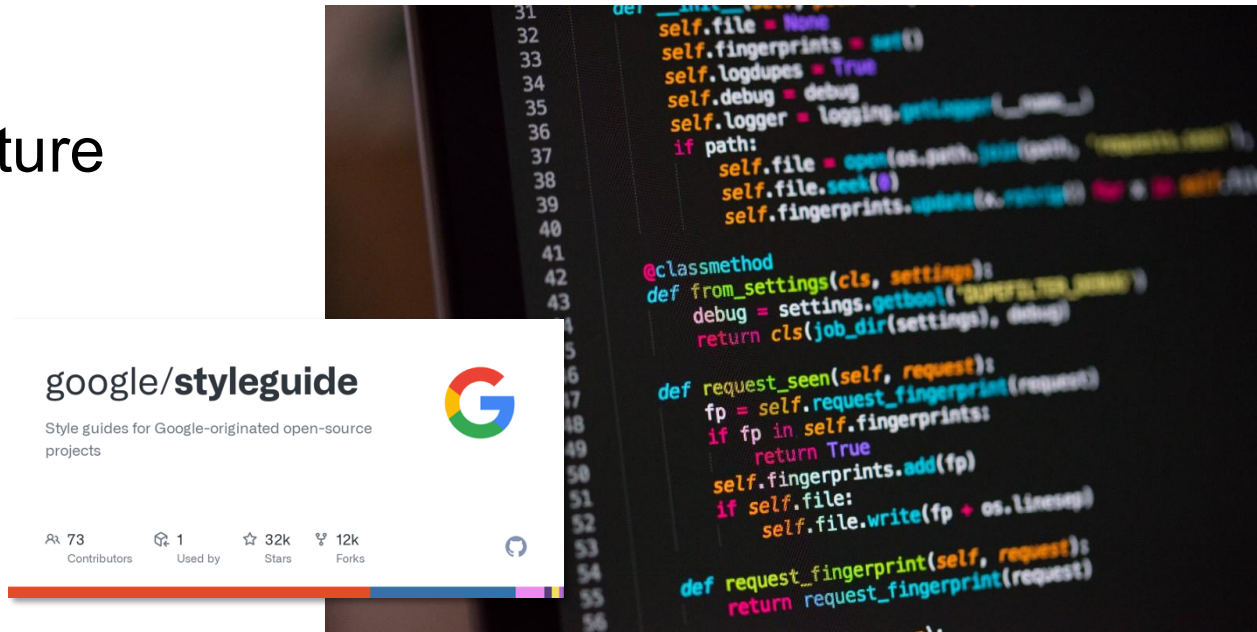
- Reads code line by line
- Revisits same code repeatedly
- Trial and error
- Only tests “happy path”

Expert

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

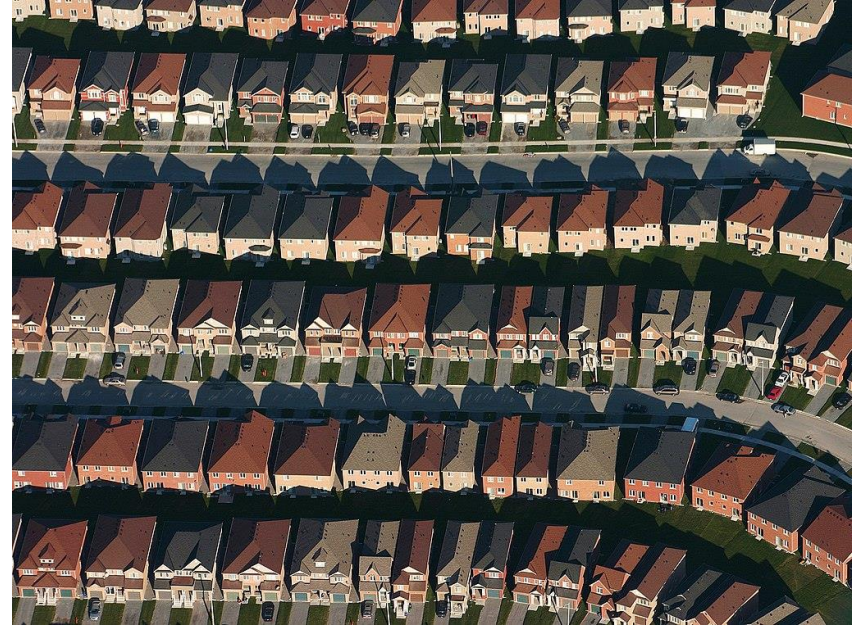
Observation: Software is full of patterns

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

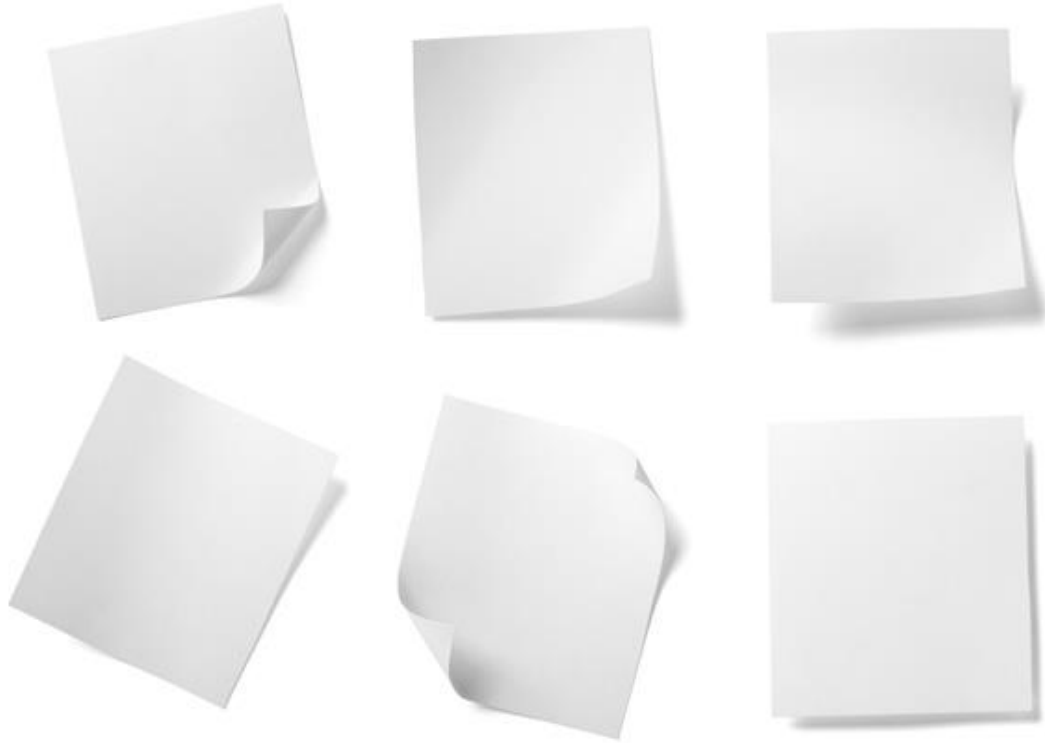


Observation: Software is massively redundant

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



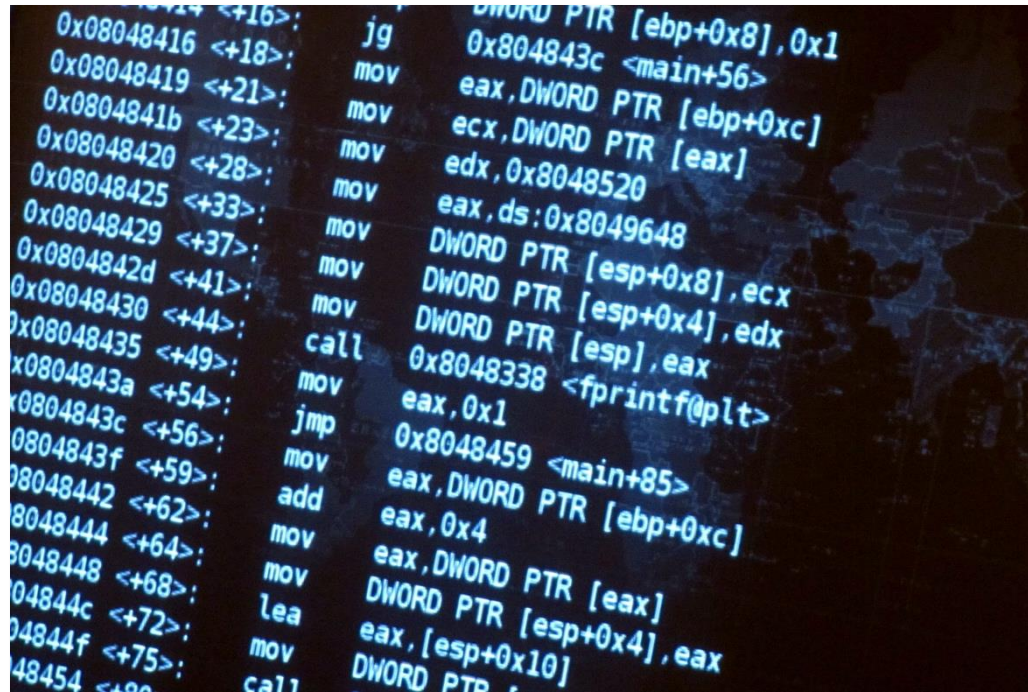
Observation: Code must run to do stuff...



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



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



```
0x08048416 <+16>: jg     DWORD PTR [ebp+0x8],0x1
0x08048419 <+18>: mov    0x804843c <main+56>
0x0804841b <+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 <+78>: call
```

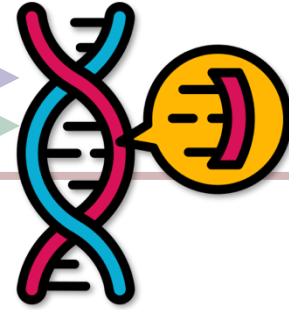
How to build, test, and refine mental models



Examine
artifacts
without
running code



Probe
running system
to observe
behavior



Modify
code, rebuild,
and assess
impact

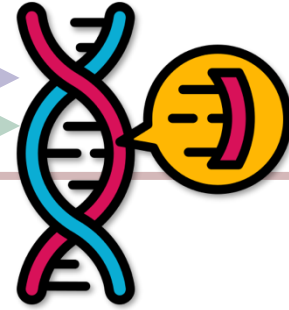
How to build, test, and refine mental models



Examine
artifacts
without
running code



Probe
running system
to observe
behavior



Modify
code, rebuild,
and assess
impact

Can code be examined, probed, and modified?

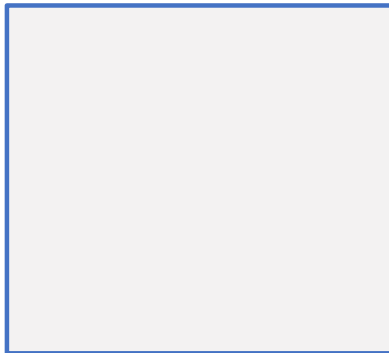
White-box



Source code built locally

- ✓ examine
- ✓ probe
- ✓ modify

Grey-box



Binaries running locally

- | Open Source | Closed Source |
|-------------|---------------|
| ✓ examine | ✗ examine |
| ✓ probe | ✓ probe |
| ✗ modify * | ✗ modify * |

Black-box



Server-side apps running remotely

- | Open Source | Closed Source |
|-------------|---------------|
| ✓ examine | |
| ⚡ probe | Talk to NSA |
| ✗ modify | |

Creating a model of unfamiliar code



Source code built
locally

Activity: Get Your Bearings

1. Get the course copy running locally with your team.
2. Try your assigned behavior in the running app.
3. Find the code that controls it. What supports your choice?

Repository

Behavior to investigate

Audiobookshelf

The skip-forward button during playback

Excalidraw

Choosing a rectangle's stroke color

OpenCode

The placeholder text in the prompt input

Get one local instance running first; investigate together. Record commands and blockers.

[Audiobookshelf sample MP3: "Carefree" — Kevin MacLeod \(incompetech.com\)](#)

[Free with attribution • License and credits](#)

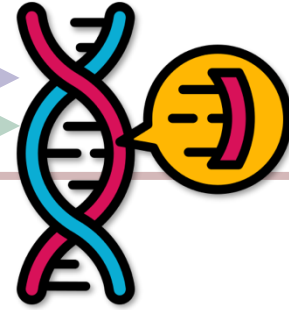
How to build, test, and refine mental models



Examine
artifacts
without
running code



Probe
running system
to observe
behavior



Modify
code, rebuild,
and assess
impact

How to build, test, and refine mental models



Examine
artifacts
without
running code



Probe
running system
to observe
behavior



Modify
code, rebuild,
and assess
impact

Examine artifacts to build a mental model



Ask

- How do we build / test / run it?
- How is this system structured?
 - Where are the **entrypoints**?
 - Where are the **seams**?
Can we probe them?
 - Where is data persisted?
- What technologies does it use?
- What are its stated features? Limitations?
- Is the project active?

Scan

- Source: code
- Build/CI: package.json, Docker, workflows
- Config: env vars, config.json, ...
- Docs: README, Documentation
- History: commits, issues, PRs, projects

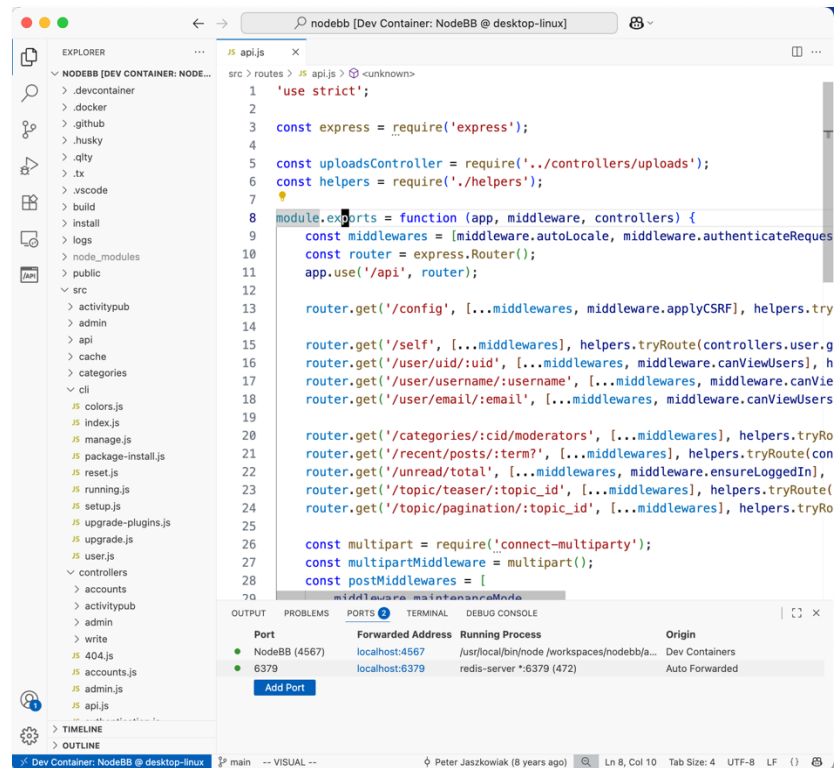
Goal

- a **build/run** command
- an **entry point** that you can target
- a **seam** that you can probe

Tip: Configure and use your IDE to its full potential

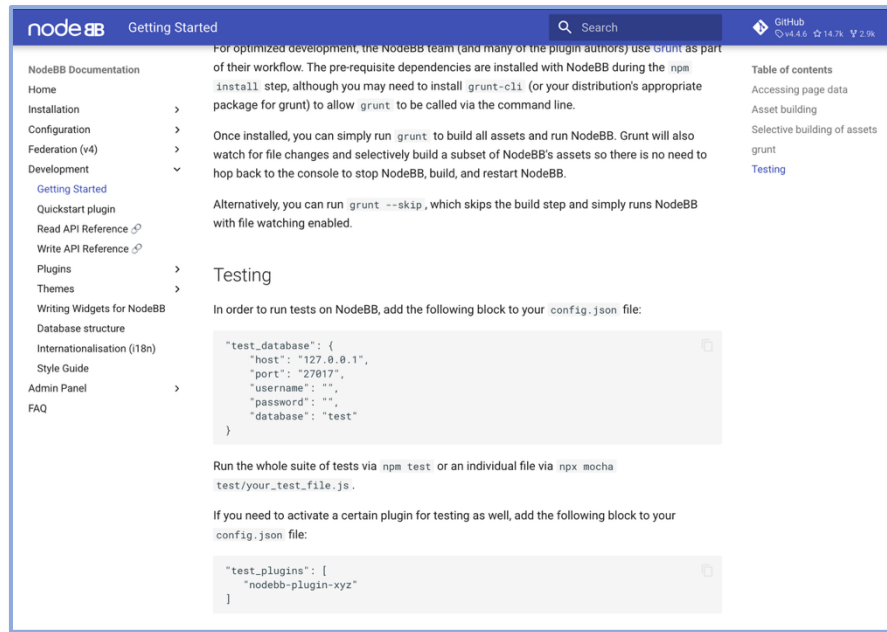


- **Right click** on code to learn more
 - variables, functions, classes, modules, ...
 - **Go to Definition**, **Go to References**, **Rename Symbol**, **Refactor**, ...
- **Install and explore IDE Extensions**
 - **Redis**, **ESLint**, **OpenAPI Editor**, **LiveShare**, ...



Tip: Consider documentation and tutorials judiciously

- Info on how to build the system, its dependencies, and how to use it
- Great for finding **entry points**
-  **Often out of date!** Treat as a starting point rather than truth



Tip: Use AI to explain parts of the code — but be careful

- Ask **specific questions** about a behavior or change you need to understand
- Request **files, functions, and call paths** that support the explanation
- Ask it to **distinguish verified findings from assumptions**
- Follow the references and **check important claims** with tests or probes
- Use AI to develop your understanding. You should be able to **explain the code yourself**



Activity: Examine the System



Work as a team. Trace one behavior in your running course copy.

- Audiobookshelf: the skip-forward button during playback.
- Excalidraw: choosing a rectangle's stroke color.
- OpenCode: the placeholder text in the prompt input

Search the source; follow references to a likely handler or component.

Sketch a small model: user action → code → visible result.

Be ready to show the file/function and one assumption to test.

<https://github.com/orgs/CMU-17695/repositories>

How to build, test, and refine mental models



Examine
artifacts
without
running code



Probe
running system
to observe
behavior



Modify
code, rebuild,
and assess
impact

Probe to test your mental model



Hypothesis → Experiment

- Introduce a probe to observe the system at a given seam or entry point
- Use the observed behavior to confirm or refute your hypothesis
- Gradually build confidence in your understanding of system behavior
- **Example:**
When I click X, handler Y runs
→ Set a breakpoint in Y then trigger X

Probes & Triggers

- Add [breakpoints](#), [logpoints](#), and step
- Logging: `./nodebb dev`
- Print statements
- Bruno / Postman / curl / httpie
- Database viewers

Goal

- One confirmed or refuted hypothesis
- One short note (trigger → code path → signal)
- One next probe or modification

Tip: Instrument the source code

- **Print debugging**

```
console.log('Administrator found, skipping Admin setup');
```

- Quick and easy
- Cons: need to rebuild + restart; easy to commit by accident

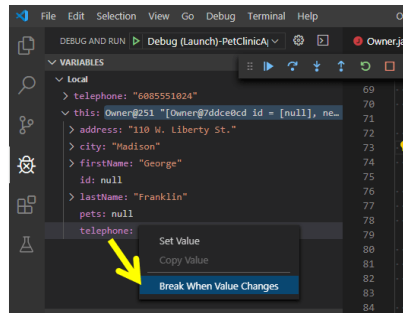
- **Structured logging**

```
winston.warn(`Flooding detected! Calls : ${socket.callsPerSecond}, Duration : ${socket.elapsedTime}`);
```

- Add levels, timestamps, and context; better for collecting data in deployment
- Cons: need to rebuild + restart

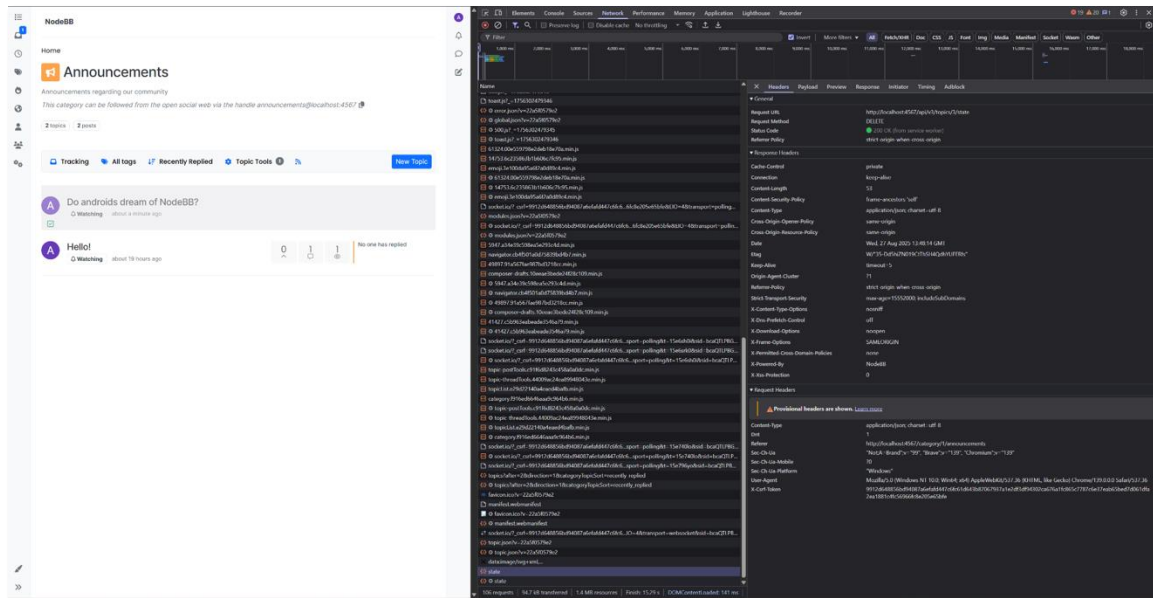
- **Debuggers**

- Inspect locals, call stack, evaluate expressions
- Add breakpoints as you go; no need to rebuild + restart
- No changes to the code means no risk of accidental changes
- We will explore the debugger in more depth later in the course



Tip: Use developer tools in the browser to spy on traffic

- Spy on web traffic while you use the app
- [Chrome DevTools](#) (also used by Brave)
- [Firefox Dev Tools](#)
- [Safari Web Inspector](#)
- **Bonus:** Use [Bruno](#), [Postman](#), [httpie](#), or [curl](#) to trigger API requests



Activity: Probe the System



Test the model you built during Examine.

Predict: *“When we do X, this code runs and we observe Y.”*

Add a breakpoint or log at the location you identified.

Trigger it: skip forward, choose a stroke color, or open the prompt.

Inspect the values: seek time, selected color, or displayed text.

Compare your prediction with the result; revise your model if needed.

Be ready to show: trigger → code location → observed evidence.

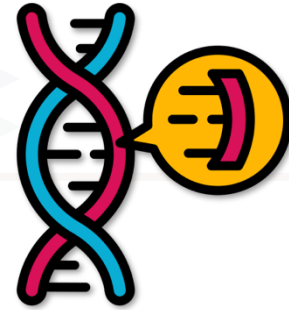
How to build, test, and refine mental models



Examine
artifacts
without
running code



Probe
running system
to observe
behavior



Modify
code, rebuild,
and assess
impact

Modify code to validate your model



Plan and execute your change

- What **behavior** should change if your model is correct?
- What's the **simplest change** that you can make?
- What **signal** can you observe? (user interface, API, logs, database, test case)
- Rebuild the code and see what happens!
- Tip: **delete debugging** is a powerful tool

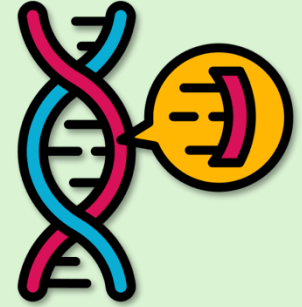
Assess impact

- Did the **predicted signal** change?
- If yes, your model **holds** for now.
- If not, you need to **revise** it.

Goal

- One change with a clear effect
- A note of what it confirms or refutes
- A next step (examine, probe, modify)

Activity: Modify the System



Make one small source change based on your model:

- Audiobookshelf: make the skip-forward button advance 5 seconds.
- Excalidraw: change the default stroke color for new rectangles.
- OpenCode: change the prompt placeholder to “What shall we build?”

Predict the result, make the edit, then rebuild/restart as needed.

Repeat the same action. Did the predicted behavior change?

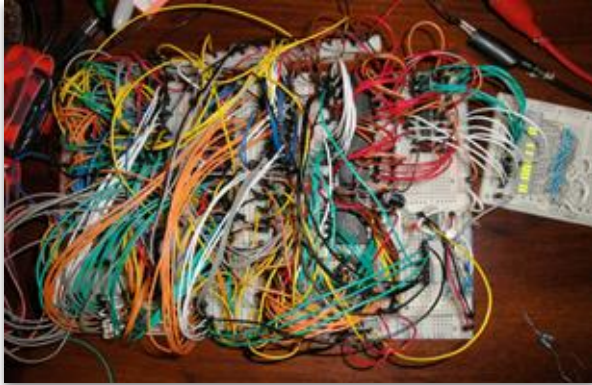
Be ready to discuss your edit, its effect, and what you learned.

What needs to change in the design?

😞 **Why not just ask Claude to make the change?**



Internal Quality



Is the code well structured?

Is the code understandable?

How well tested is the code?

External Quality



Does the software crash?

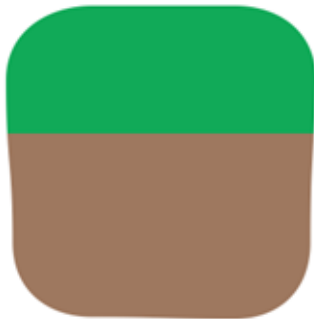
Does it meet the requirements?

Is the UI well designed?

Technical Debt

*Any software system has
a certain amount of
essential complexity
required to do its job...*

*... but most systems
contain **cruft** that makes it
harder to understand.*



*Cruft causes changes
to take **more effort***

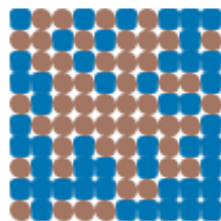


*The technical debt metaphor treats the
cruft as a debt, whose interest payments
are the extra effort these changes require.*

<https://martinfowler.com/bliki/TechnicalDebt.html>

Internal quality makes it easier to add features

If we compare one system with a lot of cruft...

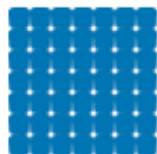


the cruft means new features take longer to build



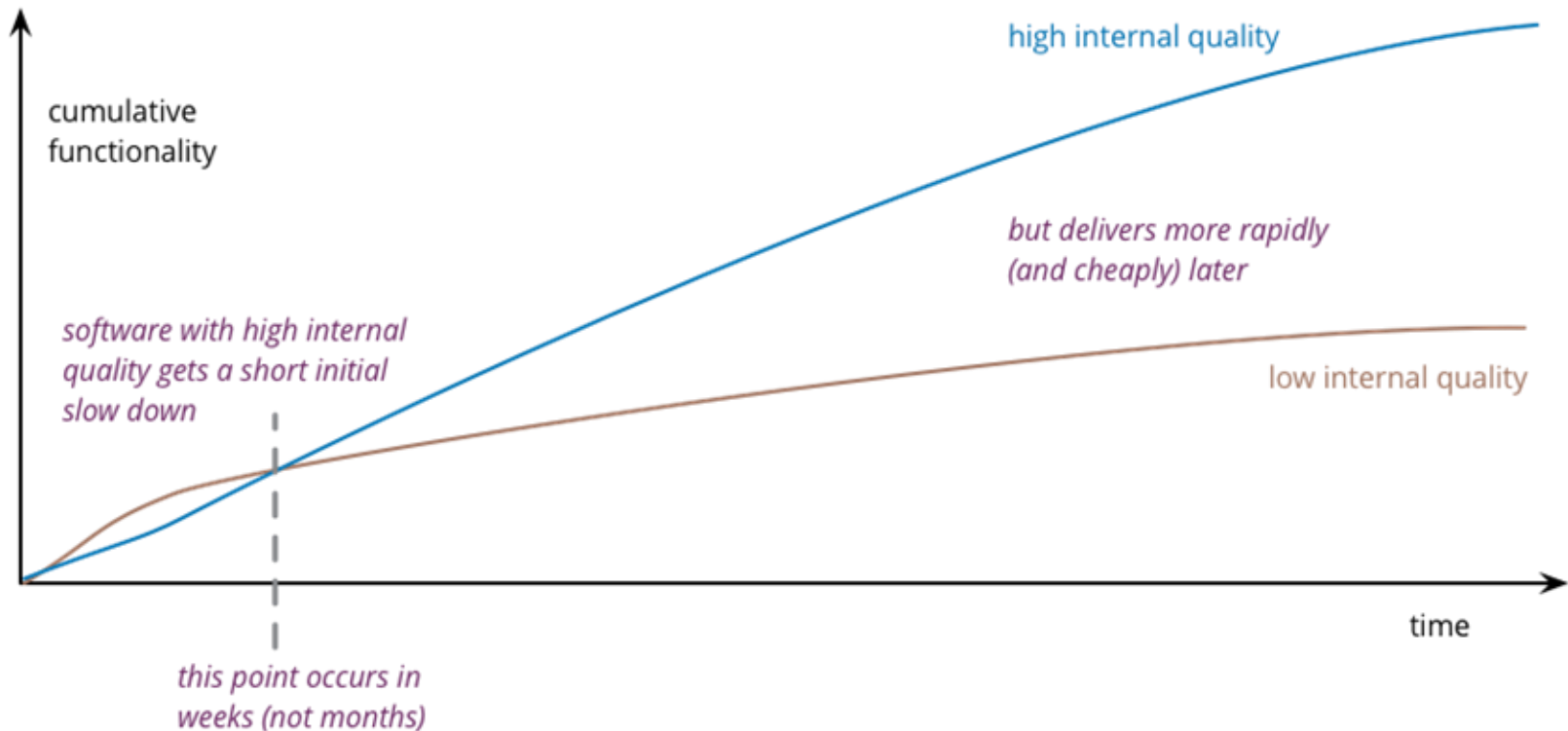
this extra time and effort is the cost of the cruft, paid with each new feature

...to an equivalent one without

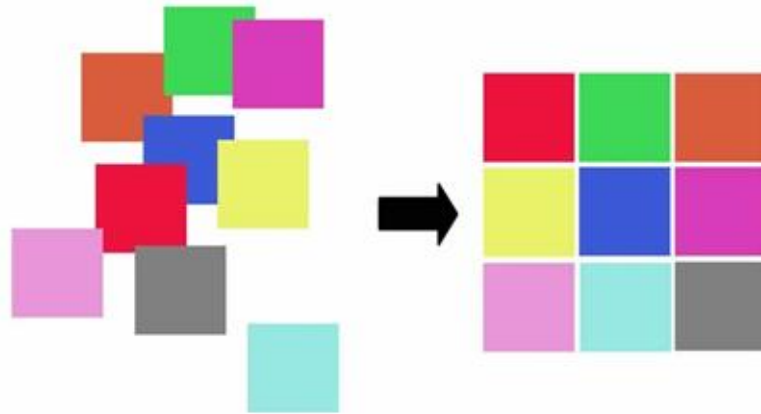


free of cruft, features can be added more quickly

High internal quality is an investment



Refactoring is the process of modifying existing code without changing its external behavior



Where should I start?



Why are you refactoring?

Performance

Where are the bottlenecks? Is there low-hanging fruit?



Technical Debt

Where is the bulk of complexity? Where are the smells?



Modernization and Feature Extensions

What is your goal? What are the least changes required?



**Beware the Allure of Clean Code!
everything has an opportunity cost**



**Instead: Scope your ambitions
don't go on a boundless refactoring quest**



**Identify
Boundaries**



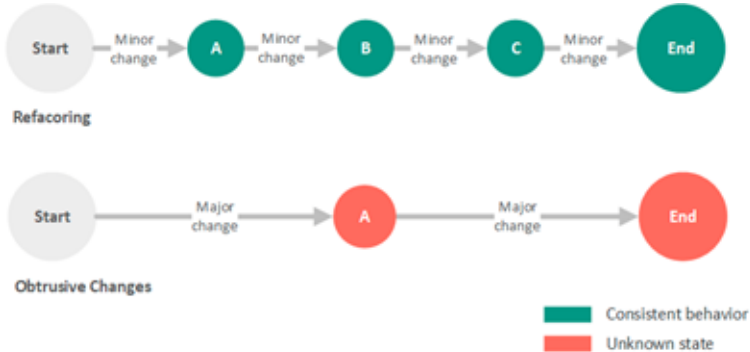
**Milestones and
Success Criteria**



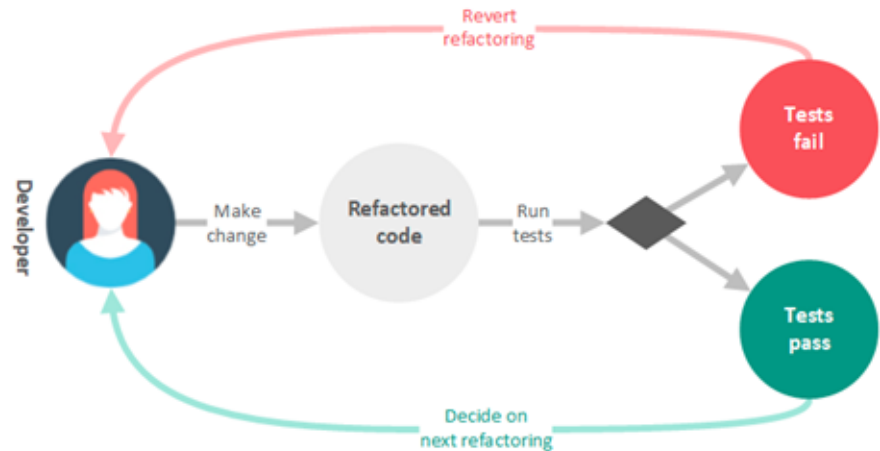
**Time-Boxed
Experiments**

Break down large changes into many smaller ones

Use tests to avoid introducing regressions into the code

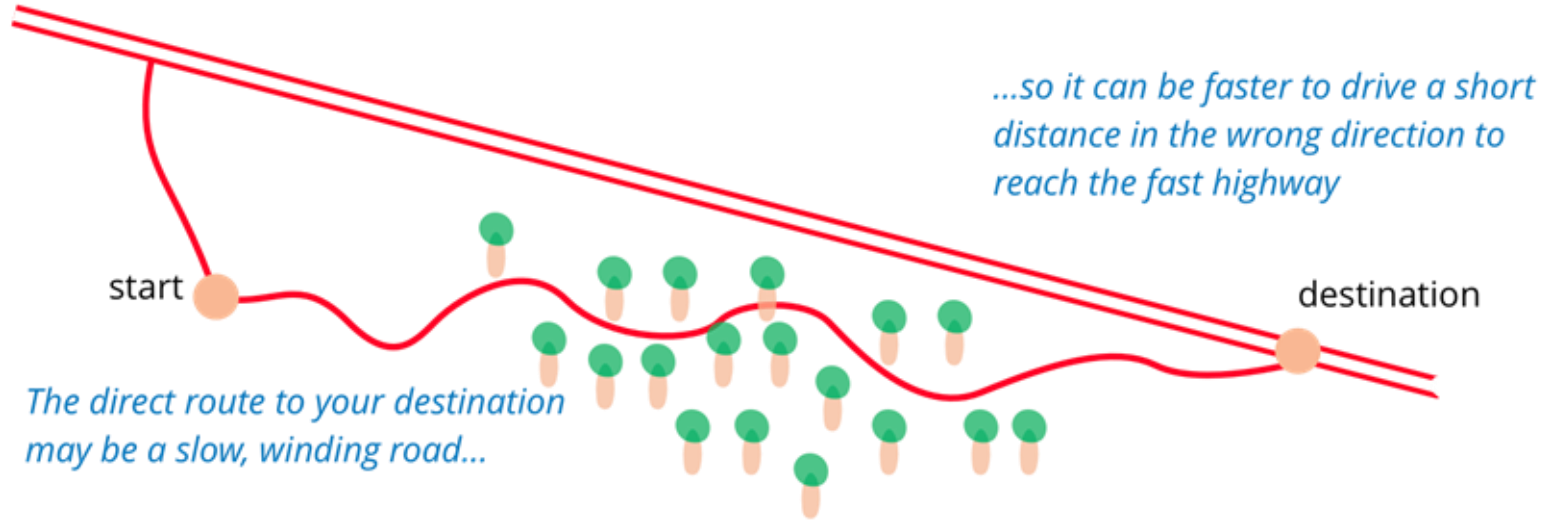


Use version control and CI to maintain a working version of the project



Preparatory Refactoring

reduce the risk of major refactoring ahead of time



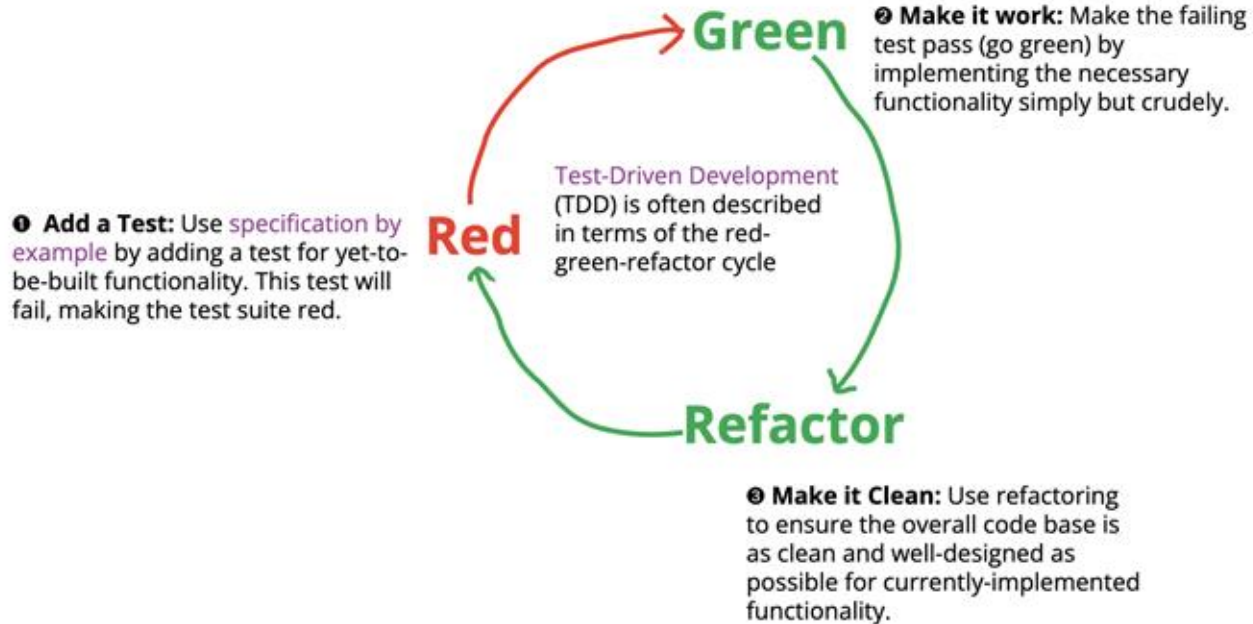
Preparatory Refactoring

reduce the risk of major refactoring ahead of time

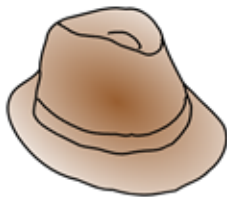


Aside: Refactoring Workflows

Refactoring as part of Test-Driven Development (TDD)



Core Idea: Two Hats of Software Development



Refactoring

When refactoring every change you make is a small behavior-preserving change. You only refactor with green tests, and any test failing indicates a mistake. By stringing together a series of small changes like this you can move more quickly and with less risk because you shouldn't get trapped in debugging.



Adding Function

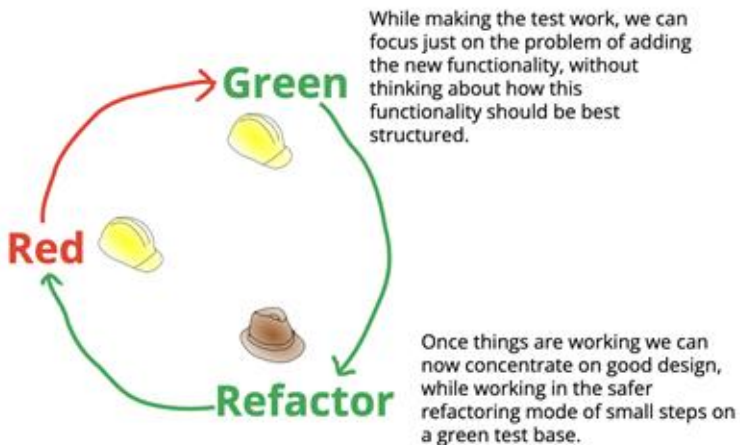
Any other change to the code is adding function. You will add new tests and break existing tests. You aren't confined to behavior-preserving changes (but it's wise to keep changes small and return to green tests swiftly).

During programming you may swap frequently between hats, perhaps every couple of minutes. But...

You can only wear one hat at a time

Workflow: TDD Refactoring

Separating refactoring keeps work focused



I call this kind of refactoring workflow:

TDD Refactoring

Workflow: Litter-Pickup Refactoring

Improve quality by resolving small issues as you develop



Eww - this code's pretty awful

As we work with a code base, we'll often run into messy areas of the code base. Maybe they were written by a less capable developer, or by a usually capable developer on a bad day, or we didn't understand how to do it properly six months ago....

Whatever the reason, by cleaning up code as we work in it, we make things quicker for us the next time we need to work with it. Perhaps even make it quicker to change now.

This is often referred to as the **camp site rule**. Always leave the code better than when you found it.



Workflow: Comprehension Refactoring

Refactor the code to make it easier to understand



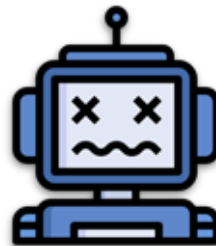
What's this block of code doing?



It's squishing the fibbly-bar.



*So should we extract it into a
`squishFibblyBar` function?*

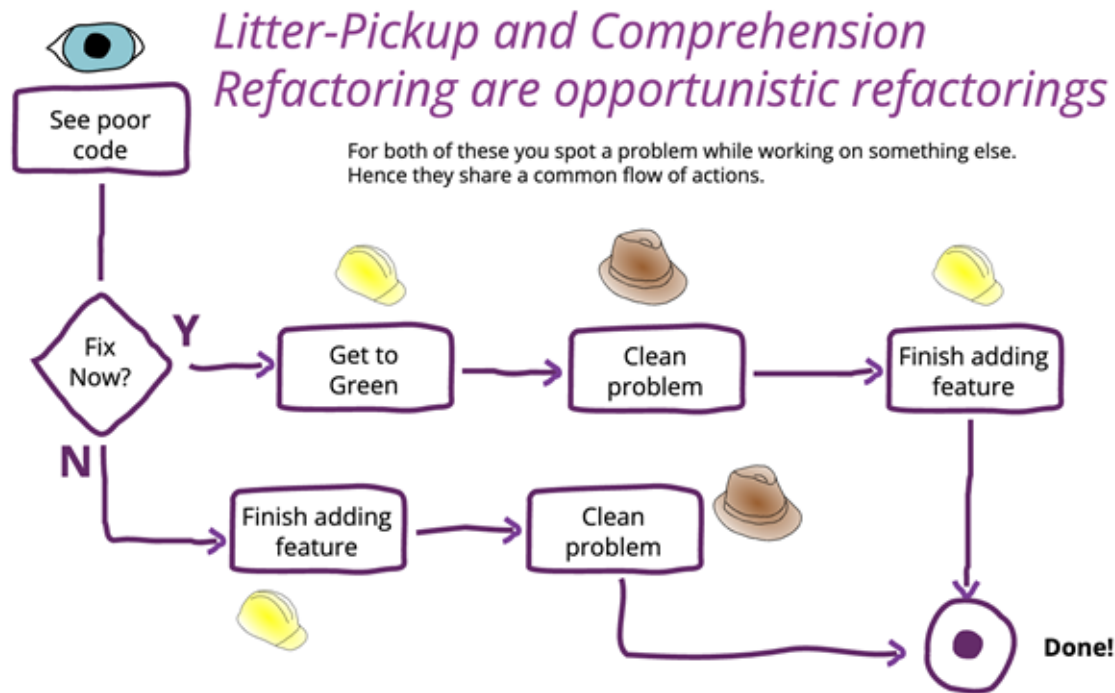


Good programmers value code that is easy to understand, and thus cheap to use and modify. But clear code, like clear writing, is hard to do. Often you can only tell how to make it clear when someone else looks at it, or you come back to it at a later date.

Ward Cunningham explained it like this. Whenever you have to figure out what code is doing, you are building some understanding in your head. Once you've built it, you should move that understanding into the code so nobody has to build it from scratch in their head again.

(You could argue that since incomprehensible code is bad code, then comprehension refactoring is a form of litter-pickup. I think of them as different since the trigger is different.)

Opportunistic Refactoring



Workflow: Preparatory Refactoring



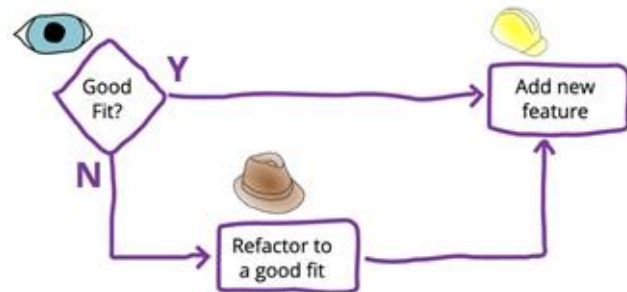
If only we'd built this class with this block of code factored out into a hook function, then this new feature would be easy.

Often you start working on adding new functionality and you realize the existing structures don't play well with what you're about to do.

In this situation it usually pays to begin by refactoring the existing code into the shape you now know is the right shape for what you're about to do.

By making this change with your refactoring hat on, you can make the new functionality change much easier. Often this pays off to the level that the overall change is faster than if you tried to add the change without the preparation.

A common analogy for this is prep work before painting. Scraping the surface and taping the edges isn't doing the painting, but often makes the actual painting go faster - and results in a longer lasting paint job.



Kent Beck
@KentBeck



for each desired change, make the change easy
(warning: this may be hard), then make the easy change

1,203 6:07 PM - Sep 25, 2012



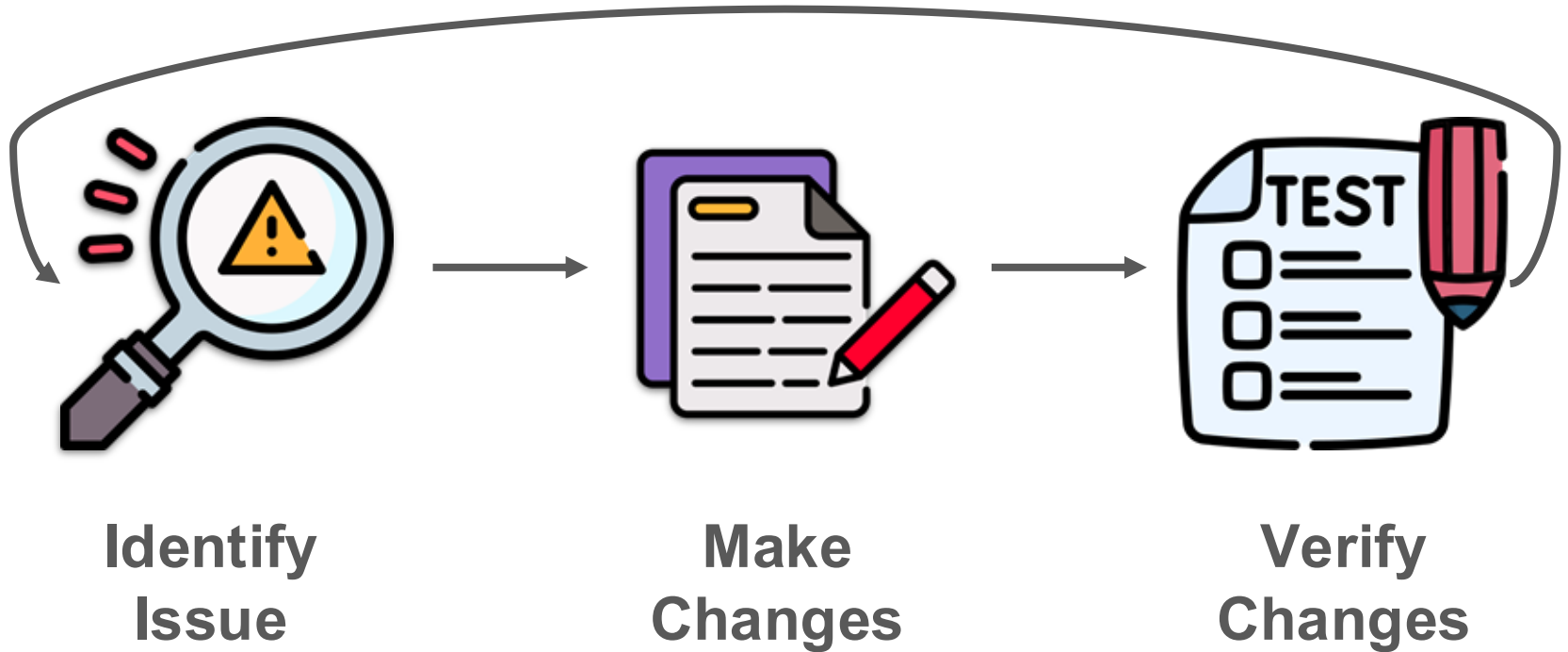
933 people are talking about this



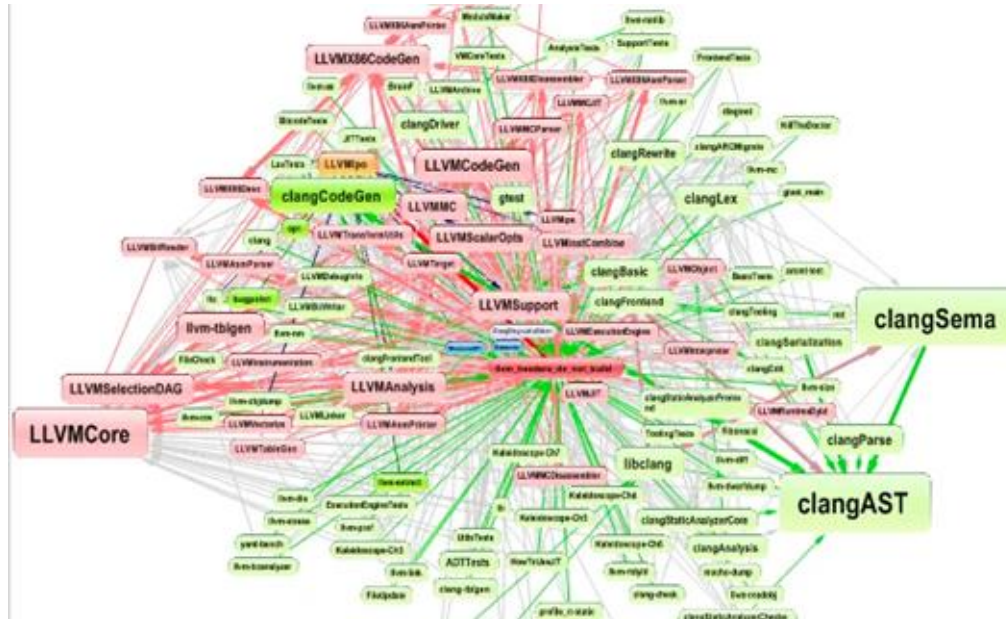
How do I avoid breaking things?

Safe Refactoring: Change > Test > Commit

how can you verify that you didn't break things

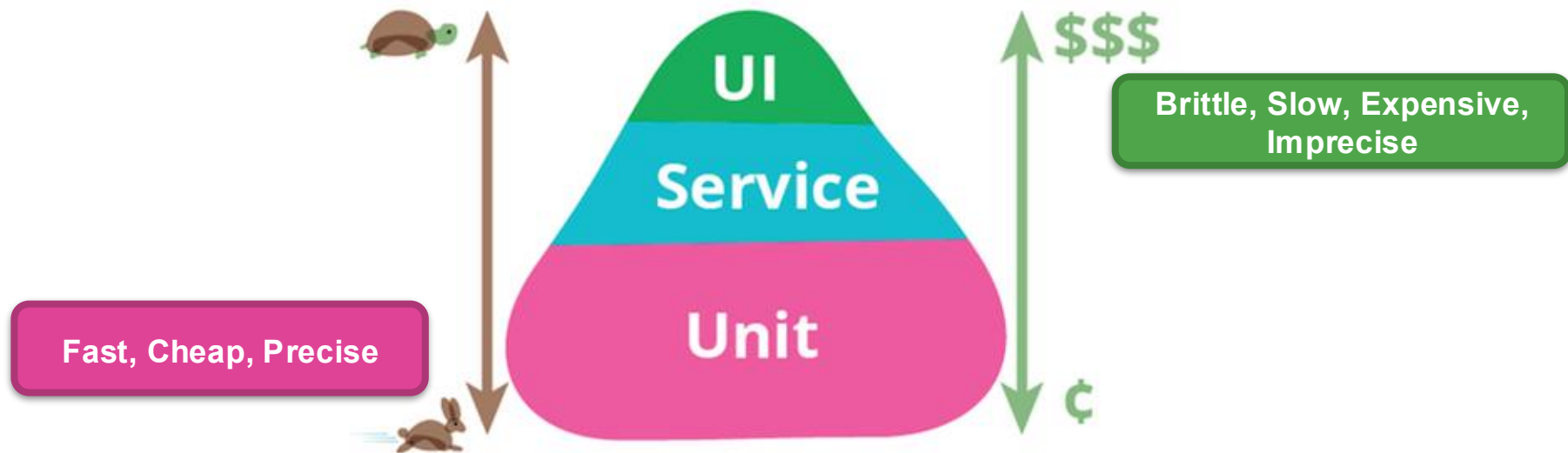


I don't have tests and the code is too difficult to test



Testing Pyramid Orthodoxy

Unit tests are traditionally preferred over E2E tests



Characterization Testing (Snapshot Testing)

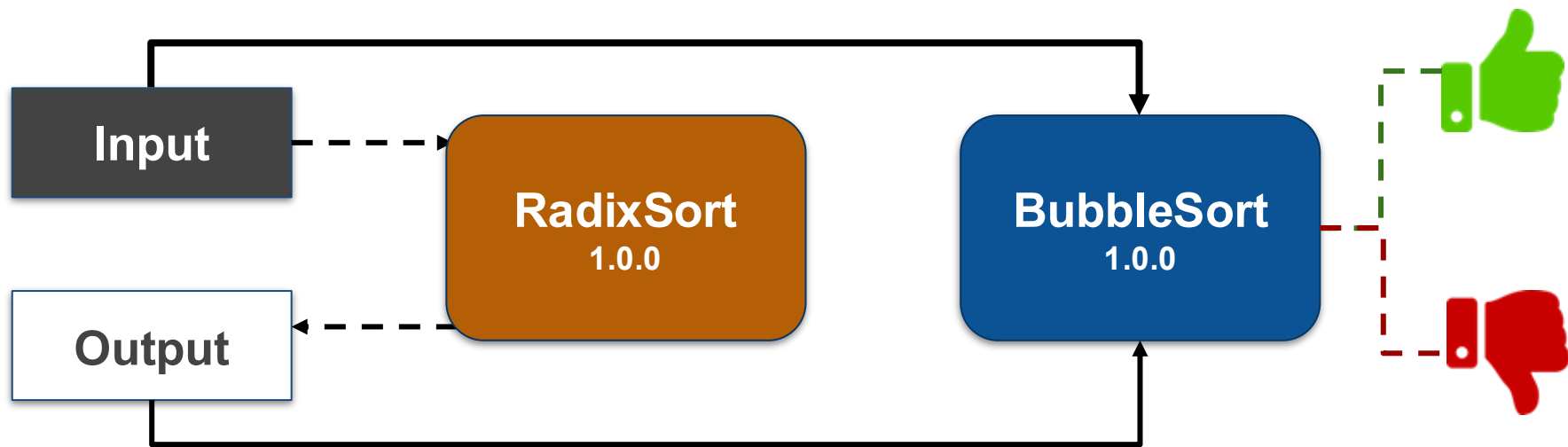
use existing code to create E2E test oracles

- Test system under different scenarios. Use the concrete output of existing system as oracle.
- A good fit for systems that have easily observable, synchronous input-output behavior
- Assumes a **fixed, unchanging environment** is used during both characterization and test runs.
- Assumes deterministic test outcomes!



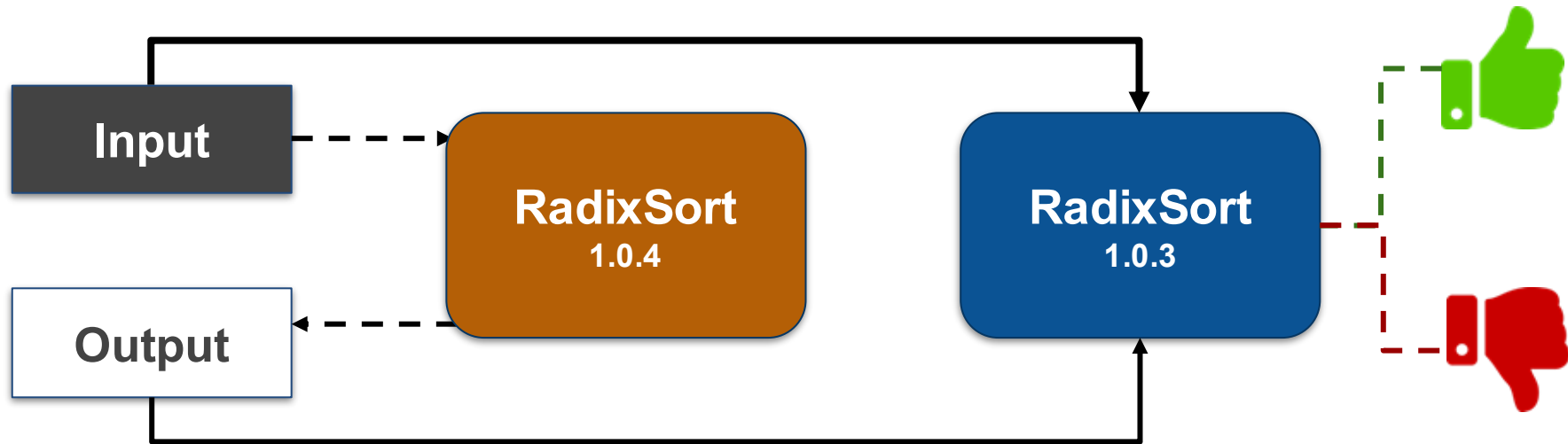
Characterization is a form of Differential Testing

Use an existing, functionally-equivalent implementation as a reference

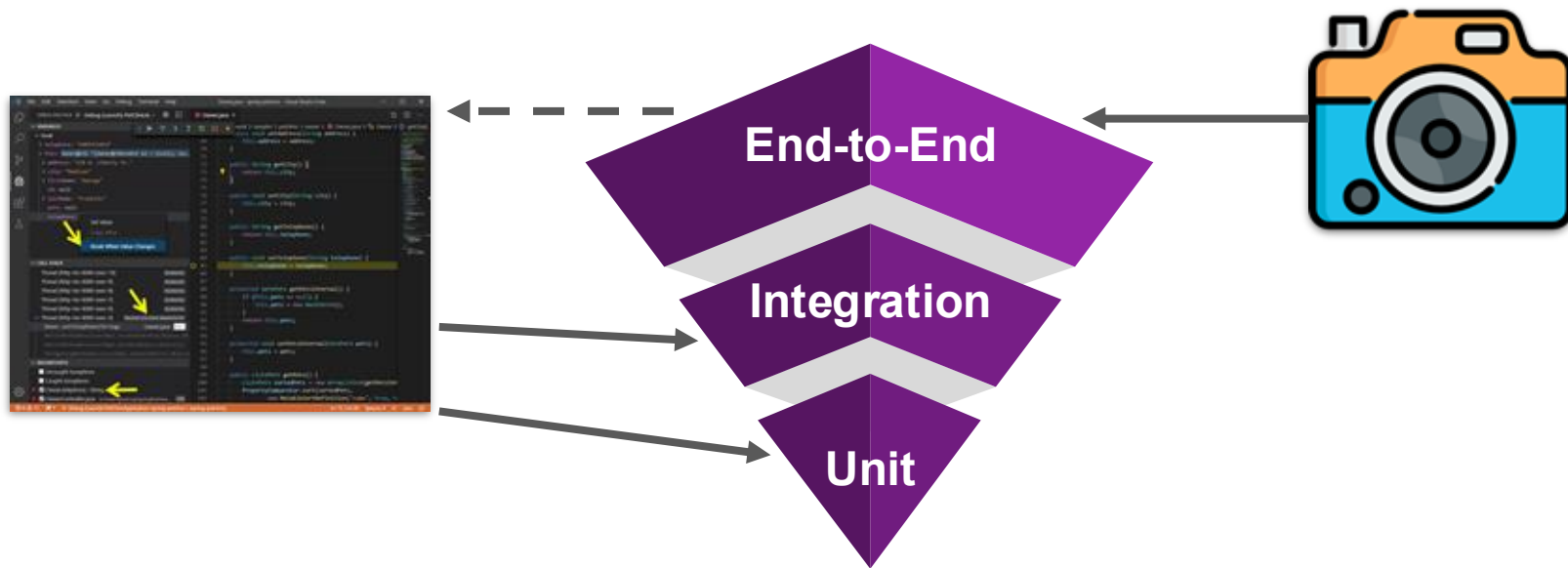


Characterization is a form of Differential Testing

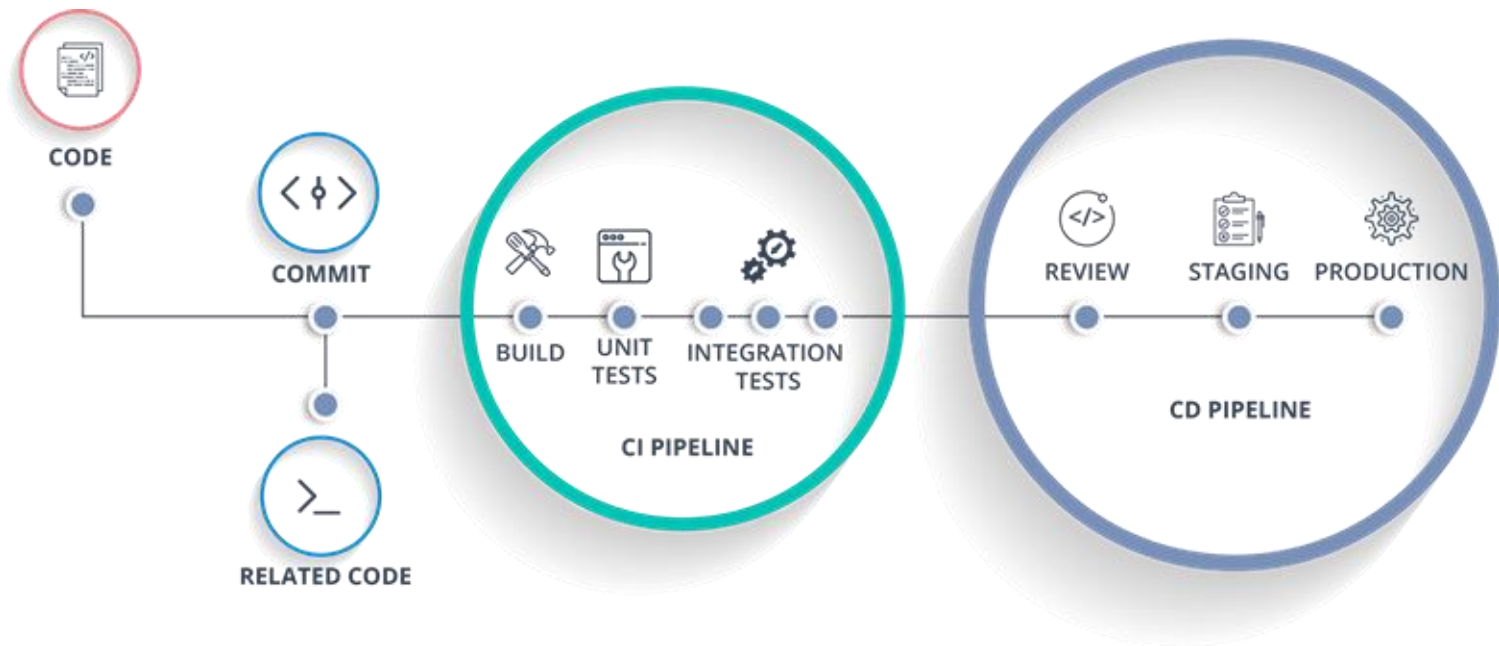
Alternatively, we can use an older, correct implementation



Flip and fill the testing pyramid as you refactor use characterization tests to generate faster tests



Use continuous integration and precommits! you shouldn't need to manually run tests and lint



Key Takeaways

- Understanding unfamiliar code means building and testing a working model through **examining** artifacts, **probing** behavior, and making small **experimental** changes
- Internal quality affects how easily we can understand and extend a system; **technical debt** adds effort to future changes
- Refactoring improves the structure of existing code **without changing its observable behavior**, making new features easier to implement
- Safe refactoring relies on checks of existing behavior and **small, verifiable steps**, with scope guided by the change we need to make