Keeping Everyone (Including Agents) on the Same Page

Documentation and Technical Debt

Zachary G. Ives & Harry Smith
University of Pennsylvania
Fall 2026

Today's Big Question

How do you keep a codebase's *intent and scope* understandable as it grows and evolves — and stop implementation technical debt from compounding faster than you can pay it down?

These problems are both associated with having a coherent shared vision of the project!

© 2026 Z. Ives & H. Smith

Want to Make a Computer Scientist Growl?

Utter the words "documentation"... or "technical debt."

They may generate an even stronger reaction than "testing."

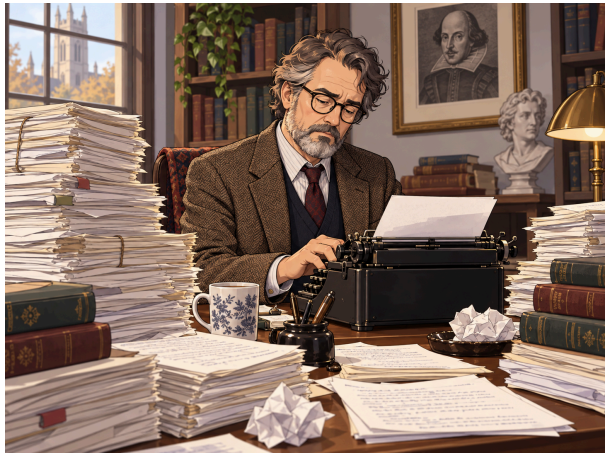
These aren't the creative and "fun part" of software...

> But: we have agents! Can they help?

© 2026 Z. Ives & H. Smith

Documentation

(Almost) Everyone Hates Documenting Things



"The fun part is over."

"But the code is right there!"

"Nobody is going to read it."

"If no one can understand it, that's job security, right?"

© 2026 Z. Ives & H. Smith

More Realistically...

Job security by obscurity is a bad idea today. It's unlikely your code is that valuable and impossible to replicate. Meanwhile, you won't be able to delegate!

The other issues are a matter of correct prioritization.

- In 6 months, you will forget a lot of your thinking and need to refresh. Not the architecture per se — the *reasons*, the trajectory, the priorities.
- Here, the code cannot say why, only how. It cannot tell them what you considered and rejected, what breaks if they change it, or how you interpreted a user need.

So: documentation, done right, is actually helpful to **you**! ... And a few others too...

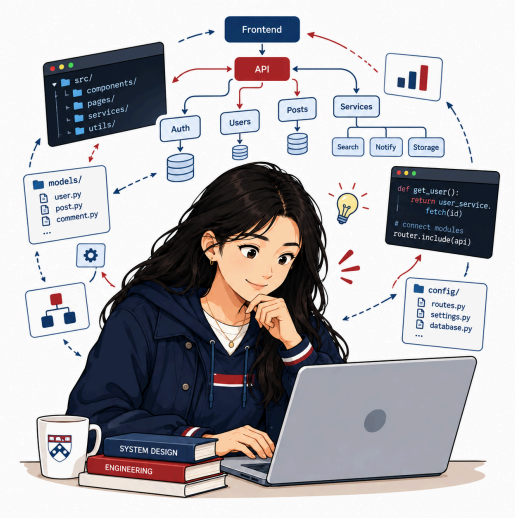
© 2026 Z. Ives & H. Smith

Three Important Audiences

You, in six months. The most reliable reader of your documentation. You will return to this code after a semester on something else with no memory of why `getPermission()` loads the guild before it resolves anything...

A new engineer on your team. They can (probably) read TypeScript. It's much harder for them to tell what are critical design decisions, which are historical accidents, and what is risky to change. Their first week is spent building a mental map, and you can help a lot!

An agent. It reads what you wrote as context before it acts. A vague document produces a plausible, confident, wrong change — as we discussed back in Week 3.



© 2026 Z. Ives & H. Smith

Two Key Audiences

The **human audience** — you as well as the future hire — need to especially understand the big-picture needs, they why, the why not, the dependencies.

- Overviews, road maps, diagrams, goals and motivations — are all an important part of the documentation
- Of course, specifications and use cases help too!

The agent adds additional requirements of **structure** and conciseness, because it does not skim and because of the limitations of the context window.

© 2026 Z. Ives & H. Smith

Putting Things into One Document

> You'll actually be doing this in Lab 5, and using it in Project 1.

> The Developer Guide you write has a human engineer joining mid-semester who needs to know what Spacebar is, how to run it, and how to add a feature — and it has an agent loading it as context before it changes something.

Of course, these same instructions are useful for the agent audience as well, and are equally helpful in framing the context for agentic development.

> Both audiences need **specificity, not generality**. "The permission system handles authorization" gives no new information. "`Permissions.finalPermission()` in `src/util/util/Permissions.ts`, should resolve in the order: owner bypass → role union" tells them something they can act on.

© 2026 Z. Ives & H. Smith

Cover *Why* over What

Code already says *what* it does — that's what reading the source is for. Documentation earns its keep by saying **why**:

- Why does the permission system apply role overwrites before member overwrites? (Because member overwrites need to win — the spec requires it, and a comment saying so saves the next person from "fixing" it backwards.)
- Why is `@everyone`'s `id` the same as the guild's `id`? (A real Spacebar design choice, not obvious from the field name alone.)

Details about how and why things are factored, unexpected interactions, and so on are also critical. Again, architecture diagrams and layering help tremendously!

© 2026 Z. Ives & H. Smith

Additional Goals When Writing for an AI Reader (too)

An agent doesn't read your docs the way a human skims a README, getting the high level — the whole relevant chunk gets loaded into context, competing for space with everything else the agent needs to hold onto.

So, you should aim for:

- **Bounded sections** with categorical headings. These beat one sprawling wall of prose — an agent can retrieve "the module guide entry for `src/gateway/`" without pulling in five unrelated sections.
- **Front-loaded** answers beat build-up — state the invariant, then justify it, not the reverse.
- **Explicit invariants** beat implied ones — "all permission checks go through `finalPermission()`, never inline" is a sentence an agent can act on (and enforce, both in code generation and in test generation).

© 2026 Z. Ives & H. Smith

Using Claude to Help Generate Documentation

> An agent is good at the mechanical half of writing, and bad at the half that matters most.

It is fast and accurate at tracing, enumeration, and referencing (though those should be checked, at least by another agent, Roborev-style).

Diagrams are a huge win. Mermaid diagrams are specified in text, so an agent writes it easily and you can have it refresh a diagram when the code changes.

DIAGRAM	LETS US ILLUSTRATE
`graph LR`	What are the layers, and what crosses each boundary?
`sequenceDiagram`	What happens, in order, for one request?
`erDiagram`	What entities exist, and how do they relate?
`flowchart TD`	What are the branches in this decision?

© 2026 Z. Ives & H. Smith

Prompting for Useful Writing

A prompt that says "write documentation for `src/gateway/`" will be high-variance. To help:

- Name the **audience** — "a new engineer who knows TypeScript but has never seen this codebase" produces different prose than "a reference for myself."
- Name the **section and its shape** — "a module guide entry with Owns, Public interface, Depends on, Key invariants, Common pitfalls" gets you a filled template rather than an essay.
- Name the **evidence** — "cite `file:line` for every claim, and say so explicitly if you cannot find support for one" converts confident invention into a visible gap you can go check.
- Name what to **leave out** — "do not describe what the code does line by line; assume the reader can read it."

> Notice these are the same four things a good specification carries.

© 2026 Z. Ives & H. Smith

What about "AI Slop Writing"?

We don't want the AI to make our reader's eyes glaze over with AI slop-isms like:

- `Permissions are not optional, they are mandatory.`
- `This is a load-bearing module.`
- `Let's delve into the answer.`
- Lists with emoji bullets

You can help a bit with a file called `CLAUDE_WRITING.md` (next), but you should also make sure you go through and add more context, and make things more colorful to keep your reader focused. And, of course, to make sure what is being said matches the main points.

© 2026 Z. Ives & H. Smith

CLAUDE_WRITING.md: Conventions for How It Writes

You can have `CLAUDE.md` reference a file (say, `CLAUDE_WRITING.md`) on writing style.

```
## Style & Tone
* **Actionable & Direct:** Use second
  person, present tense ("You run
  `npm start`").
* **No Marketing Filler:** Ban words
  like "powerful", "seamless",
  "robust", "leverage", "delve".
* **Simple Syntax:** Write short
  sentences. Prefer a period to a
  semicolon. Target an 8th-grade
  reading level.
* **No Conversational Padding:** Avoid
  intros like "Let's dive in".
```

```
## Precision & Accuracy
* **Hard Evidence:** Every technical
  claim must cite `file:line`.
* **Define Acronyms:** Spell out
  acronyms once, at their first use.
* **Design Rationale:** When two
  designs were possible, state which was
  chosen and why.
* **Define Boundaries:** Explicitly
  state what a module is *not*
  responsible for.


## Formatting Rules
* **Punchy Lists:** Limit bullet points
```


© 2026 Z. Ives & H. Smith

Own the Documentation

A lot of details can be handled by the agent. Your manual will be most valuable if you put in the time to describe things like — why this design and not the obvious alternative, what breaks if you change it, which parts are deliberate and which are accidents nobody has cleaned up yet.

© 2026 Z. Ives & H. Smith

So: Iterate and Supervise, Don't Delegate Judgment

The working pattern is a loop, not a handoff.

- You draft the structure and the claims you know.
- The agent fills in enumeration, finds the call sites, drafts the diagram.

You read what comes back and correct it — and the corrections are the valuable part, because each one is a place where your model of the system and the agent's disagreed, and you had to go find out which of you was right.

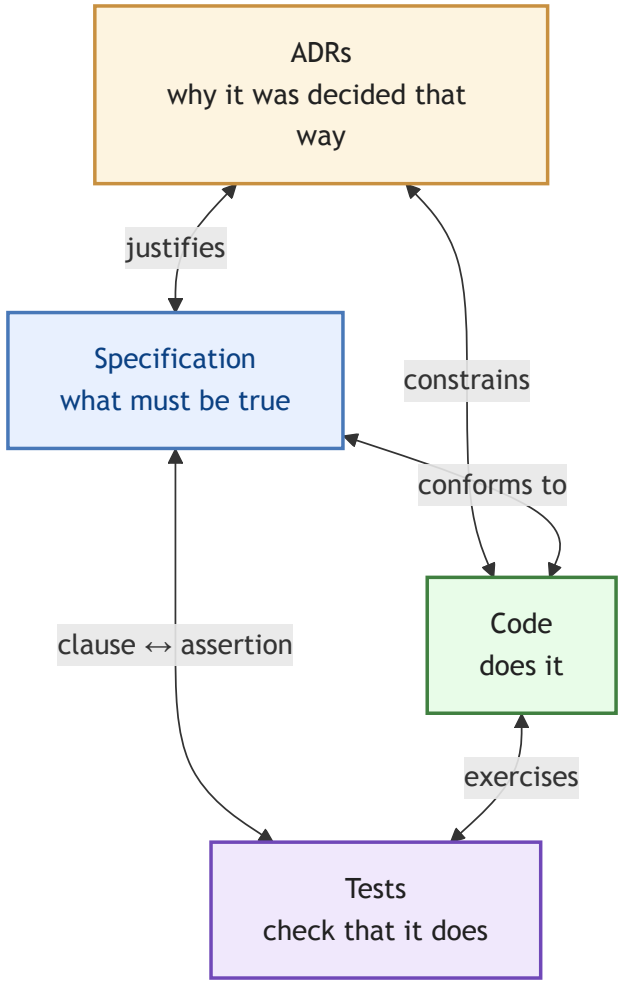
© 2026 Z. Ives & H. Smith

The Result: 4 Artifacts for 1 System

Now you are maintaining four descriptions of the same system — at different levels of formality!

They drift. Someone patches the code under deadline pressure and the spec goes stale. Someone updates the spec and the tests keep asserting the old behavior.

> Drift is worse than absence. An out-of-date manual or spec is misleading!



 © 2026 Z. Ives & H. Smith

Tracking and Triggering Based on Drift

An agent can check a document against the source. Walk the spec clause by clause and report which ones the implementation appears to violate. Treat the output as hypotheses to verify (as in Project 1).

CI can fail on it. Your pipeline already fails when tests break. It can equally fail when a documented file path no longer exists, or when a spec file is older than the module it describes.

A commit hook can catch it at the source. A hook is a script git runs at a fixed point — `.git/hooks/pre-commit` runs before a commit is recorded, and a non-zero exit aborts it. Drift caught there never enters the history.

A review agent can enforce it continuously. `roborev` reviews every commit against the ADRs and flags a change that contradicts a recorded decision.

© 2026 Z. Ives & H. Smith

Architecture and Documentation Matters!

There are examples from open source where a project that was well-architected and documented, often beat better-resourced frameworks.

- Vue.js - progressive web framework (competitor to React)
- Django - Python web backend
- SQLite and BerkeleyDB

© 2026 Z. Ives & H. Smith

Documentation, In Summary

Three readers — future-you, a new engineer, an agent — and the first two want the same thing, so a document has two jobs. Carry the **why**, which the source cannot, and carry **structure**, which is what makes it findable by something that does not skim.

- Delegate the enumeration and the diagrams.
- Keep the reasoning: the agent can describe every design in this codebase and explain none of them.

Assume it will drift, and put something in the pipeline to keep things synced.

> Create documentation to help your team maintain the big picture that isn't in the code.
>
> This is much more possible today, than it was even 12 months ago — and the effort-to-payoff curve is different!

21

© 2026 Z. Ives & H. Smith

Technical Debt

Remember This, from Week 1?

Week 1 warned about it in one line: *your agent adds a new permission check function each time you do something new, each doing a slightly different thing.* Six weeks of "add a permission check" prompts, and here is what that actually looks like in a repository:

```
checkPermission        validatePermission      verifyUserPermission
hasChannelAccess       canUserRead             getEffectivePerms
```

None of them tested. Each named after whichever function the agent happened to see most recently in its context window.

This is **Failure Mode 2 — Accelerated Technical Debt** — seen from the inside. It is the default outcome when nobody is watching for it, not an edge case.

© 2026 Z. Ives & H. Smith

Debt Is a Loan You Took Without Reading the Terms

Technical debt is future work you implicitly committed to by taking a shortcut now. Every duplicated function above is a debugging session someone will owe later — with interest, because now there are *six* places a permission-logic fix has to land, and no way to know that from reading any one of them.

Agents make this **exponentially** worse than human teams do, for a specific mechanical reason:

- A human reusing a function has to *find* it first — friction that nudges toward reuse.
- An agent regenerates plausible code fresh from whatever's in its context window *right now*. If the existing `checkPermission` isn't in context, it doesn't exist as far as the agent's concerned. It writes a new one instead.

© 2026 Z. Ives & H. Smith

An Example (Based on the Real World)

A startup with a handful of pilot customers, each with staged deliverables on an engineering timeline. The Engineering Director ran an Agile process — sprints, with specific customer-facing deliverables at the end of each one.

- The road map was broken into major milestones and releases.
- Features were defined, dependencies were determined, and sprint lengths were spelled out.

> Development was focused, above everything else, on hitting dates and milestones.

That sounds like good management, and in most respects it was. Every sprint shipped. Every customer got what they were promised, when they were promised it.

© 2026 Z. Ives & H. Smith

Two Sequenced Customers Meant Two Sequenced Sprints

Customer A needed data in a key/value store, plus files, and a user interface to fetch, search, and manipulate that data.

Customer B needed tabular data — CSV loaded into a relational database — and a user interface to fetch, search, and manipulate that data.

Promises were made about deliverables for Customer A. Since Customer B signed on at a later date, they were given later deliverables.

- Two sprints. Two deliverables, built in sequence. Design for customer A was done independent of Customer B's needs.

© 2026 Z. Ives & H. Smith

Breakout -- Discuss What was Lost with this Approach?

Five minutes, in pairs. Look at those two descriptions again.

If we build these in separate, uncoordinated sprints — what opportunities did we just miss? And what could we have done differently, without slipping either delivery date?

© 2026 Z. Ives & H. Smith

What Was Actually Different?

Read the two requirements again.

Both needed a UI. Both needed fetch. Both needed search. Both needed manipulation of records. What differed was one thing: **where the bytes live.** A key/value store with files for one, a relational database for the other.

- Could we develop one abstraction and API for "collections of things"?
- Could we modularize the storage?

We built two of everything except the UI. Two search implementations, two fetch paths — and then maintained both, because each had a customer attached to it.

> The component we did not build was an interface: `get(id)`, `find(query)`, `put(object)`. One UI above it, two adapters below it. Customer B's sprint would have been *faster*, not slower, because most of it already existed.

© 2026 Z. Ives & H. Smith

Why Two Sequenced Projects, Then?

Nobody was careless, and nobody was wrong about their own sprint.

Each sprint was scoped as a **customer deliverable** rather than a **system capability that needed to be designed for evolution**. Customer A's sprint had no reason to ask what Customer B would need, because Customer B was next quarter's problem and asking would only put the date at risk.

The information was available the whole time. Both customers were already signed. What was missing was the architect looking across sprints.

> Sprints are not the villain here. Sprints optimize locally, correctly, and something has to look globally — and that something is an engineer, not a process.

© 2026 Z. Ives & H. Smith

The Long-Term Debt

Having largely redundant code incurs a long-term cost.

1. If we need to make a modification (say with a new field) this needs to be propagated, and made consistent, and tested, for two different code paths.
2. If there are two potential variations on the storage, there is a good chance of a third coming up. We don't have an architecture that properly factors the components.

Someone will have to pay down the road — less understanding, more bugs, more tests, refactoring, redesigning, ...

© 2026 Z. Ives & H. Smith

Claude's Implementation Matches Our Example

Every prompt is essentially a sprint.

Ask an agent to build Customer A's stack and it will, competently. Open a fresh session and ask for Customer B's and it will build that too — from scratch, because the first stack is not in this context window and therefore does not exist.

The agent optimizes for the deliverable in front of it, with whatever it can currently see, and it has neither memory nor incentive to notice that it is writing its second search implementation this month.

> The difference is speed. That startup took two quarters to build two of everything. An agent can do it before lunch, and it will look like productivity the entire time. But it's accruing future debt!

© 2026 Z. Ives & H. Smith

Should We Build Everything for Extensibility

The obvious lesson is "build a common abstraction up front," and that lesson is wrong often enough to be dangerous.

- With one example in hand, **unless you have a lot of experience**, you do not know where the variation will be.
- Abstract too early and you will guess the wrong axis — you will build a storage interface that assumes queries return rows, and the second customer will want a stream or an object.

Seeing real world cases helps us figure out the interface. Find data points to know *which dimension actually varies*.

© 2026 Z. Ives & H. Smith

What Makes an Interface Robust

An interface that names the **operation** outlives the details of its parameters or underlying implementation. One that names the **implementation** does not.

`find(query)` still means something when the key/value store becomes Postgres. `runSqlQuery(sql)` only ever meant one thing, and it leaked the database into every caller the day it was written.

Create two inexpensive tests before you commit to an interface.

- Could you write a second implementation of it that is genuinely different?
- And if you swapped implementations, would any caller have to change?

> We used to have a "rule of 3" -- three copies before abstracting. But since agents reduce the barrier, we can lower the threshold.

© 2026 Z. Ives & H. Smith

Redesign / Refactor: When to Change the Architecture

Architecture is not something you fix on a schedule. It is something you notice at a threshold, and there are recognizable signals.

Examples:

- One change requires edits in several places, and nothing in any one of them tells you about the others.
- You cannot say where a new feature belongs without starting the sentence "it depends."
- Features get *slower* to add as the system grows.
- You are writing the third branch or the third adapter for what is clearly one concept.
- **Your agent keeps regenerating functions instead of reusing.**

© 2026 Z. Ives & H. Smith

This Is a Case of What We Called "Deep Modules"

Last week we discussed a **deep module** — one entry point, a clear contract, complexity hidden inside — like the `get`/`find`/`put` interface.

> Architecture, as we discussed last week, and debt-prevention, as discussed this week — are the same lever, pulled at different points in the process. One shapes the codebase so debt is harder to create; the other is what you do once it exists anyway. The result of a refactor is in fact a new architecture.

© 2026 Z. Ives & H. Smith

Safe Refactoring: You Can't Just Stop and Rewrite

Production code can't take a two-week outage while you clean it up. Three established patterns let you refactor *underneath* a system that has to keep running.

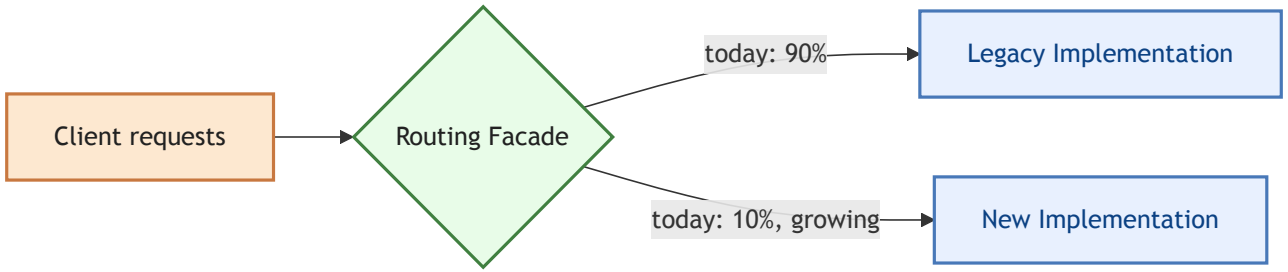
- **Strangler Fig** — route traffic through a facade; shift it from old to new gradually
- **Branch by Abstraction** — introduce an interface, migrate behind it, swap the implementation, then remove the old one
- **Parallel Run (Dark Launch)** — run new code alongside old, compare outputs, don't serve real traffic from it yet

All three share one property: **at every intermediate step, the system still works.** That's the whole point.

© 2026 Z. Ives & H. Smith

The Strangler Fig

Named for the vine that grows around a host tree, gradually taking over, until eventually nothing of the original is left — a name from Martin Fowler, applied to software instead.

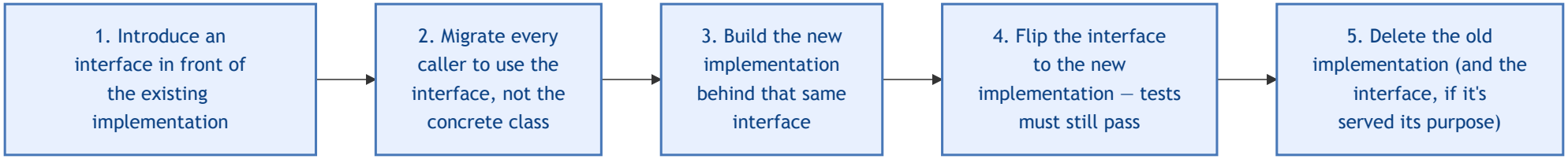


The facade's routing ratio shifts from 100/0 to 0/100 over time — a config change, not a code change, at each step. Once nothing routes to the legacy path, delete it. Nobody experienced a cutover; there wasn't one moment where everything switched at once.

© 2026 Z. Ives & H. Smith

Branch by Abstraction

For refactors that can't be split traffic-by-traffic — a single code path, not two systems.



At every step, `npm test` passes. That's what makes it safe to do incrementally instead of on a branch nobody dares merge.

© 2026 Z. Ives & H. Smith

Parallel Run

When the old code is the only specification you have, run both and compare.

The old code path still serves the response; the new one's output is computed, compared, and logged.

- Low-risk
- Essentially generates test cases from common requests

Permission resolution is close to the ideal candidate:

```
const legacy = legacyResolve(user, guild, channel);
const next   = finalPermission({ user, guild, channel });
if (legacy !== next) log.warn("perm mismatch", { user: user.id, legacy, next });
return legacy;                          // old path still serves
```

When the mismatch log goes quiet for long enough, retire the old code and move to the new.

© 2026 Z. Ives & H. Smith

What Parallel Run Costs

You pay for the computation twice, which rules it out where the operation is expensive or has side effects — you cannot parallel-run a `DELETE`.

The harder cost is the comparison. Two implementations can be correct and still disagree in ways you have to classify: a different key order, a set that is not an array, a rounding difference in the last digit. Writing that comparison honestly is most of the work, and writing it *loosely* gives you a green log and no information.

> All three patterns share the property that matters. At every intermediate step, the system still works and the tests still pass — which is what makes it safe to stop halfway when something more urgent arrives, and something always does.

© 2026 Z. Ives & H. Smith

Agents as Refactoring Engines

The same incremental, test-anchored structure that makes these patterns safe for humans is exactly the shape that keeps an agent's refactor honest:

- **Bad prompt**: "Refactor the permission system." Unbounded, unverifiable, invites an agent to rewrite everything at once with nothing anchoring correctness.
- **Good prompt, step 1**: "Extract `Permissions.finalPermission()` into its own module with unit tests. Existing behavior must not change — all current tests must still pass."
- **Good prompt, step 2**: "Now consolidate `checkPermission`, `validatePermission`, and the other four into calls to `finalPermission()`. One call site at a time. Run the test suite after each one."

> Be sure you have tests at each step as an independent verification of (likely) correctness.

© 2026 Z. Ives & H. Smith

When the Old Code Is the Only Spec

```
// Generated once, against the old implementation, before touching anything.
const CASES = [
  { in: { roles: ["everyone"], overwrites: [] },              out: "0x400" },
  { in: { roles: ["mod"], overwrites: [denyView("mod")] },    out: "0x0"   },
  // ... one entry per input you decided mattered
];

for (const c of CASES)
  assert.strictEqual(newResolve(c.in).toString(16), c.out, JSON.stringify(c.in));
```

These are **characterization tests** — Michael Feathers' term for tests that describe existing behavior rather than required behavior.

© 2026 Z. Ives & H. Smith

Choosing the Inputs

The technique is only as good as the input set.

Read the old code for its branches. Every `if`, every early return, every `default`, every `catch` is a path, and a path with no input reaching it is a path your refactor can silently break. Agents are good at this: *"list every distinct execution path through this function, and for each one give me an input that reaches it."*

Take the common cases from reality. Production logs, an existing test fixture, a sample of real requests.

Push on the boundaries. Empty and single-element and many. Zero, one, maximum. Absent versus present-but-null. For permission resolution: no roles, one role, overlapping roles, an overwrite that mentions no bits.

Include the failures. Inputs that make the old code throw are behavior too, and "it used to throw and now returns undefined" is a regression that a happy-path suite will not catch.

© 2026 Z. Ives & H. Smith

What Characterization Tests Are Not

They lock in the bugs along with everything else. If the old code got the admin bypass wrong, your generated cases now assert that it stays wrong.

During a refactor that is exactly what you want — behavior must not change, and a test that enforces it is doing its job. The danger is forgetting which kind of test you are holding.

> A characterization test says **"this is what it did."** A conformance test — Project 1's kind — says **"this is what it must do."** One is derived from code, the other from a specification.

After a refactor, go back through the generated cases and ask which ones you would have written from the spec. The ones you would not have written are worth a second look.

© 2026 Z. Ives & H. Smith

Doing this for Real

Lab 5 is an attempt to make our advice from today actionable. You'll find at least three real debt items in Spacebar and summarize each one's risk, remediation effort, and what you would actually do about it.

That last column is where the judgment lives.

This will build towards our goals of Project 1.

© 2026 Z. Ives & H. Smith

Recap: Documentation

- **Three readers, two requirements.** Future-you and a new engineer want the same thing, so a document carries **why** — which the source cannot — and **structure**, which makes it findable by a reader that loads rather than skims.
- **Delegate enumeration, keep the reasoning.** An agent drafts module guides and diagrams well, and cannot tell you why a design beat its alternative. The corrections you make are where the understanding is.
- **Assume drift.** Build in mechanisms at commit-time to keep things in sync, and to notify when significant drift has happened.

© 2026 Z. Ives & H. Smith

Recap: Technical Debt

- **Agents accelerate debt mechanically, not carelessly.** They regenerate from whatever is in context rather than searching for what exists. Six near-identical permission functions is the default; so is building two of everything for two customers.
- **Abstraction is a bet you place on the second example, not the first.** The second instance tells you which dimension varies. An interface that names the operation survives a swap; one that names the implementation does not.
- **Re-architect on signals, not on a schedule** — one change touching many places, "it depends" when asked where a feature goes, features slowing as the system grows, your agent regenerating instead of reusing.
- **Safe refactoring keeps the system working at every step.** That property is what makes Strangler Fig, Branch by Abstraction, and Parallel Run safe — and it is what keeps an agent-driven refactor honest.

© 2026 Z. Ives & H. Smith

Further Reading

Documentation

- Diátaxis — four documentation types (tutorial, how-to, reference, explanation) and why mixing them makes all four worse
- A Philosophy of Software Design — Ousterhout; the chapters on comments and on why "the code is self-documenting" fails
- Write the Docs: Style Guides — what goes in a `CLAUDE_WRITING.md`, from people who maintain them professionally

Technical debt and refactoring

- Big Ball of Mud — Foote & Yoder, the canonical technical-debt paper
- Strangler Fig Application — Martin Fowler
- Branch by Abstraction — Martin Fowler
- ParallelChange — Martin Fowler; the expand/migrate/contract discipline behind all three

© 2026 Z. Ives & H. Smith

Questions?

zives@engineering.upenn.edu · sharry@engineering.upenn.edu