COMP 3550

9.1 — WHAT IS LEGACY CODE? (AND WHY IT'S NORMAL)

Week 9: Legacy Software, Architecture Recovery & Change

LEGACY ≠ BAD CODE

“Legacy code is code without tests.”
(Not a judgment on quality, a statement about maintainability)

Why It’s Normal

- Software ages naturally, environments, tools, and needs evolve
- Long-lived systems inevitably become legacy over time
- Rewriting from scratch is risky and expensive
- Legacy code often represents valuable domain knowledge baked in

LEGACY ≠ BAD CODE

“Legacy code is code without tests.”
(Not a judgment on quality, a statement about maintainability)

- Not inherently bad — legacy code can be brilliant and battle-tested
- Often old or written in outdated languages/frameworks
- May be undocumented or have missing context
- Sometimes no active maintainers — team has moved on
- Business-critical systems can be legacy and still work flawlessly

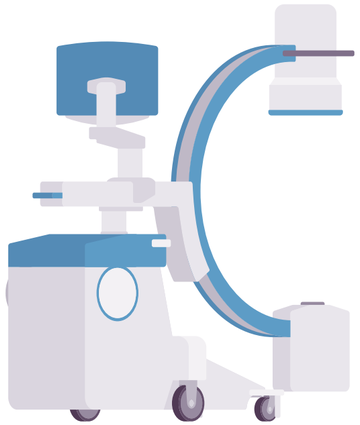
Legacy is a signal that the maintenance context has changed not that the code is inherently poor

EXAMPLES OF LEGACY IN REAL SYSTEMS

TONS OF:

- Banking Software
- Medical Software
- Aviation Systems
- Government Technology

but why?



CHALLENGES OF LEGACY CODE

Fear of Breaking Things

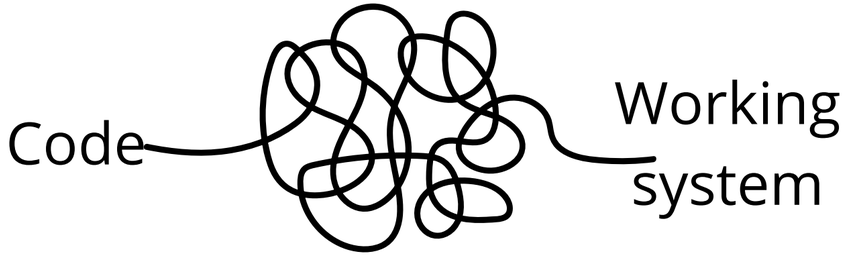
- No tests → changes feel risky
- “If it ain’t broke, don’t touch it” mentality

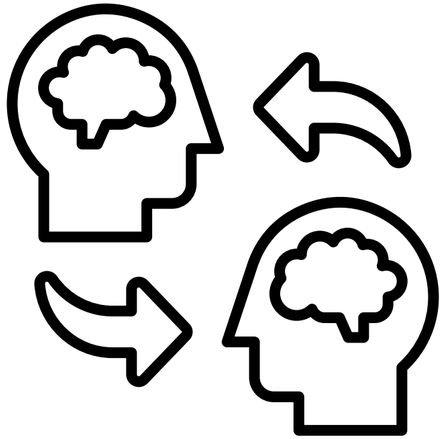
Hard to Refactor

- Tight coupling, tangled dependencies
- Outdated patterns or language features

Lacks Documentation or Context

- Original authors gone
- Minimal/Non-existent knowledge transfer





CHALLENGES OF LEGACY CODE

Fear of Breaking Things

- No tests → changes feel risky
- “If it ain’t broke, don’t touch it” mentality

Hard to Refactor

- Tight coupling, tangled dependencies
- Outdated patterns or language features

Lacks Documentation or Context

- Original authors gone
- “Tribal knowledge” not recorded

We end up with a **confidence gap**: developers hesitate to make improvements because they don’t fully understand the impact

MINDSET SHIFT — “DON’T BE A HERO”

Big rewrites are high risk:

- Unknown dependencies
- Hidden business logic
- Time & budget blowouts
- Legacy code’s value is in what already works

MINDSET SHIFT — “DON’T BE A HERO”

Favor Small, Safe Changes

- Add tests first (if missing)
- Tackle one module or function at a time
- Validate each change before moving on

Embrace Incremental Understanding

- Learn the system by working with it, not against it
- Ask “What’s the smallest improvement I can make today?”
- Document as you go — future you will thank you

PROJECT PAUSE & REFLECT

Find a class in your project that lacks tests or documentation. Would a new dev call this legacy?

How can we ensure our future selves remember what we did in our projects?