IMPROVING LEGACY CODE

Víctor Llana Pérez

Mario Orviz Viesca

David Pedregal Ribas

Ana Pérez Bango

DEFINING LEGACY CODE

Old and outdated code

Untested code (Michael Feathers)

> "Legacy code is valuable code that you are afraid to change."
>
> - Nicolas Carlo
>
> - Code still runing in production
> - Provides value and has an impact

Costly in the long run if not maintained, even if the process is feared or even painful

REFACTOR

REWRITE

- Safer and more sustainable
- Improve code as you go, as new features are implemented you evolve the system (not always viable)

If there are no safety nets :

- Only use "duct tape" fixes (minimal and temporary)
- Setting safety measures comes first

- Auatomated testing
- Monitoring logs
- Lost Documentation
- Lost Knowledge

- High risky
- Lack of a way to validate if it works

Use only when

- New system that is based on an old one
- Changing only a small and well-defined part (isolation of a monolith)

MANAGEMENT

Problems

- We anticipate problems
- They do not understand us

Solutions

- Quantifying technical debt
- Aligning with Business Metrics

By translating technical challenges into business terms, developers can facilitate management's understanding and support for necessary refactoring initiatives.

CULTURE

Problems

- People
- How they operate
- Visibility of new features

Solutions

- Recognition of maintenance work
- Encouraging incremental refactoring

Rewriting code later due to new requirements or scale isn't a failure, it's often a sign of success and learning.

Integrate it into daily work. Dedicated sustainability weeks are better than nothing but less ideal

KEY TECHNIQUES

Different techniques to apply for
dealing with legacy code

BEHAVIORAL ANALYSIS

Analyze legacy code as if it was a crime scene

- You don't know what happen there
- Usage of GIT as a "forensic" tool to understand what happened through the development of such code
- Detect how often a file was changed (hotspot analysis)
- Know which developers were in charge of what parts of the code

Adam Tornhill:

- *Your Code as a Crime Scene*
- *Software Design X-Rays*

HOTSPOT ANALYSIS

Detect critical points in the legacy code

- Parts of the code that have suffered a lot of changes (many commits)
- Complex pieces of code
- Some languages provide tools to detect complex code
- Trend of lines tends to follow the trend of complexity
- Build use case to present to management

AUTOMATED REFACTORINGS

Automatization of some refactoring actions (renaming a variable)

- Some IDEs provide this functionalities
- Underutilized
- The usage of these tools can prevent introducing bugs when refactoring
- Speeds up the refactoring
- We can use AI to improve this process (we must have a solid test suite)

INVERTING DEPENDENCIES

Separating business logic from side effects

- Increase readability
- More testable code
- Easier to refactor
- Isolates the core logic from volatile external dependencies.
- Dependency Inversion Principle (SOLID principles)

NAMING AS A PROCESS

- Better names are found as the knowledge about the code gets deeper
- Refine the name during the process of developing the code
- Improves code readability
- Names also act as a signal for refactoring (funciton DoAAndBAndC())

Inspired by Arlo Belshee. Naming as an evolving iterative activity

MIKADO METHOD

- Iterative process for very large tasks
- Do as much as you can in 10 minutes, erase and repeat
- Create a mind map of all the tasks needed to complete the original problem
- When you can finish a task in less than 10 minutes, commit
- Following this method encourages good practices

AI IN LEGACY CODE

- Supporting role
- Problems evaluating results
- Analyzing, testing and refactoring

11/04/2025

QUESTIONS