Defining Legacy Code

Legacy code is often defined simply as untested code or as old and outdated code, but according to **Nicolas Carlo**, it is valuable code that has an impact (that comes from being in production, serving clients...) and that you are afraid to change. The reluctance to modify it comes from risky factors such as the lack of tests, the complexity of the code and the possible missing documentation or knowledge. As such, maintaining this code can be painful but it avoids long-term consequences.

Refactor VS Rewrite

Although at first glance they may appear similar, there are some key differences between refactoring and rewriting when it comes to legacy code:

- **Rewriting** is a highly risky approach as there may be no way to guarantee whether the new code fulfills every function that the old one did, and as such it often fails. Nicolas Carlo only recommends rewriting when building a new product based on the old one from the ground up, or when changing a small and well-defined isolated part of the code.
- **Refactoring** is the best approach overall, as it is safer and sustainable. It can also be implemented in an incremental way, and it is highly beneficial to incorporate it into the daily work, although this may not be viable in every case.

There are also cases in which **neither approach** should be used, mainly when the code lacks safety nets (such as tests, monitoring and documentation), and that means that only minimal "duct tape" fixes until these safety nets are resolved.

Management and culture

Developers see some problems coming that can be solved with refactoring, however it is not done because management does not allow this. This is because developers use technical vocabulary and cannot communicate properly the value and importance of these changes. There are two ways to fix this problem:

- **Quantifying Technical Debt:** Assess the costs associated with unresolved bugs and the opportunity costs of delayed features.
- **Aligning with Business Metrics:** Demonstrate how technical enhancements can positively impact service level agreements (SLAs), customer satisfaction, and overall business performance.

The problem with legacy code is the way people operate in it, which is not helpful. The culture we usually have in businesses makes visible only new features and the makers of those features. People maintaining the project keeping it up to date, do not get recognized. Some ways of solving this problem:

- **Recognition of Maintenance Work:** Carlo recommends acknowledging and celebrating efforts such as dependency upgrades during sprint reviews, thereby encouraging proactive maintenance.
- **Encouraging Incremental Refactoring:** Carlo advocates for integrating refactoring into regular development workflows.

Key Techniques

- **Behavioral Analysis**: Try to understand how the code was created using version control systems to know more about the legacy code. Key points as how many times a file was changed or what developer was in charge of each part of the code
- **Hotspot Analysis:** Find "hotspots" in the code. Hotspots are files that have suffered a lot of changes or have a high complexity.
- **Automated Refactorings:** Delegating tedious refactoring tasks, such as variable renamings to automatic tools or artificial intelligence
- **Inverting Dependencies:** Separating business logic from side effects. Isolates the core logic from volatile external dependencies.
- **Naming as a Process:** Visualizing naming not as a one-shot decision but as a evolving iterative activity. Better names are found as the knowledge about the code gets deeper. Names can be a signal for refactoring.
- **Mikado Method:** An iterative process where you try to complete a task in a small period of time, identifying smaller subproblems each time until you can complete them in less than the established time. Helps tackle very large problems.

AI in Legacy Code

AI in legacy code can be used as a supporting tool when working with legacy code, since it can help you understand the code, create new tests or refactor the code.

However, we should not grow reliant on AI to do these jobs for us, since it will probably introduce mistakes somewhere along the lines. If we are not involved in the process, we won't realize since we won't be able to know if the work done by AI is correct or not without previous tests on the code.