

Experiment Plan – Scoping and Planning

Júlia Moreira Nascimento¹

¹Department of Software Engineering
Pontifical Catholic University of Minas Gerais (PUC Minas)
Av. Dom José Gaspar, 500 – Coração Eucarístico – Belo Horizonte – MG 30535-901

1. Basic Identification

1.1. Experiment Title

Quality and Maintainability in Legacy Software: An Empirical Analysis of Refactorings on GitHub

1.2. ID / Code

Experiment ID: PUCM-ES-RES-2025-001

1.3. Document Version and Revision History

| Field | Value |
|------------------|--|
| Document Version | v1.0 |
| Revision History | <ul style="list-style-type: none">v1.0 — Initial draft (21 Nov 2025) — Júlia Moreira |

1.4. Dates (creation, last update)

| Field | Value |
|---------------|------------------|
| Creation Date | 21 November 2025 |

1.5. Author (name, department, contact)

Author 1: Júlia Moreira, Software Engineering Department,
moreira01092002@gmail.com

1.6. Principal Investigator / Experiment Owner

PI / Owner: Nascimento (Principal Investigator / Study Lead)

1.7. Related Project / Product / Initiative

Recent research has explored the challenges and opportunities associated with refactoring legacy software systems, offering empirical insights highly relevant to this study. [Yanakiev et al. 2025] presents an experience report on applying SOLID principles to legacy codebases, showing that systematic design-driven refactoring can significantly improve maintainability while reducing structural complexity. Their work highlights practical obstacles faced by development teams when modernizing aging systems.

Similarly, [Ferreira et al. 2024] investigate refactoring practices in React-based web applications, revealing that front-end legacy systems also accumulate considerable

technical debt, often intensified by rapid framework evolution. Their findings demonstrate that refactoring improves readability and consistency, but requires careful coordination across UI and state-management layers.

In the context of collaborative development, [Coelho et al. 2025] conducted a qualitative study on refactorings induced by code reviews. Their results indicate that many refactorings emerge not from planned technical-debt management, but organically through the social dynamics of review discussions. This reinforces the socio-technical nature of legacy code improvement.

Focusing on large industrial projects, [AlOmar 2025] examines the dynamics of refactoring branches in the Qt ecosystem. The study uncovers patterns in how developers isolate, structure, and integrate refactoring work in complex repositories, identifying factors that contribute to both successful and problematic refactoring efforts.

Finally, [Chen and Hayashi 2024] proposes MORCoRA, a multi-objective refactoring recommendation approach that incorporates the availability of reviewers. Their work demonstrates how decision-support tools can optimize refactoring strategies in legacy systems by aligning technical needs with organizational constraints.

Together, these studies establish a strong foundation for understanding both the technical and social dimensions of refactoring in legacy software, directly supporting the motivation and methodology of the present research.

2. Context and Problem

Legacy software systems remain deeply embedded in the operational core of governments, financial institutions, and large enterprises. Despite their age, these systems continue to support mission-critical processes, but their maintenance and modernization impose enormous financial and organizational burdens. As reported in [?], corporations and governments have spent an estimated US\$35 trillion on IT products and services since 2010, with approximately 75% of this budget dedicated solely to maintaining existing systems.

A particularly critical issue concerns the financial impact of unsuccessful modernization and refactoring efforts. According to the same report, of the US\$2.5 trillion invested in attempts to replace legacy systems, roughly **US\$ 720 billion** has been wasted on failed modernization projects. These failures often stem from large-scale refactoring or system rewrites that do not reach deployment, exceed budget, or result in unstable systems once delivered.

Technical and organizational inertia further complicates modernization. Many legacy systems rely on outdated languages such as COBOL and run on hardware no longer supported by vendors, increasing the risk associated with any structural change. Executives often defer modernization efforts due to high uncertainty, potential operational disruption, or unclear return on investment. As a result, organizations are frequently pushed into a cycle where maintaining legacy systems becomes more feasible than replacing them, even if long-term costs escalate.

This phenomenon is reflected in public-sector spending: in fiscal year 2019, nearly 80% of the U.S. government's IT budget was allocated to the operation and maintenance of legacy systems alone, leaving little room for modernization initiatives. The burden

of keeping these systems functional often outweighs the resources available for strategic improvement.

Given this context, understanding the real impact of refactoring becomes essential. Refactoring is frequently presented as a lower-risk, incremental alternative to full system replacement. Instead of attempting to rewrite entire systems, teams can gradually improve maintainability, reduce technical debt, and modernize architecture piece by piece. However, the high historical cost of failed modernization efforts (including unsuccessful refactorings) highlights the need for empirical evidence about what characterizes successful versus unsuccessful refactoring efforts in practice.

Therefore, this study focuses on analyzing refactorings in open-source legacy repositories to better understand the patterns, characteristics, and outcomes associated with successful and failed refactoring attempts. Such insights aim to support developers, researchers, and organizations in making informed decisions about modernization strategies for long-lived software systems.

2.1. Problem / Opportunity Description

Legacy software systems constitute a substantial portion of the technological infrastructure supporting both public and private organizations. Although these systems often underpin critical business operations, they commonly exhibit issues such as low performance, poor maintainability, high structural complexity, and increasing evolution costs. As a result, development teams frequently rely on refactoring practices to improve the internal quality of the code without altering its external behavior.

However, in practice, many refactoring efforts fail to deliver the expected benefits. Several indicators reveal such failures, such as an unexpected rise in defects after refactoring, longer development time for new features, higher maintenance costs, and limited adherence to software engineering best practices. Organizations often invest significant financial resources in modernization initiatives that underperform due to poor planning, a lack of proper metrics, or an insufficient understanding of the actual impact of code changes.

This scenario presents a clear opportunity to empirically analyze both successful and unsuccessful refactorings in public GitHub repositories. Such an investigation can uncover patterns, success factors, and risk indicators that influence the outcomes of refactoring activities. Moreover, it can provide actionable insights for technical teams to better assess whether a refactoring effort is likely to yield meaningful improvements in software quality and maintainability, or instead generate additional cost and complexity.

2.2. Organizational and Technical Context

The refactoring and modernization of legacy software systems take place in environments marked by high operational dependency, resource constraints, and complex socio-technical dynamics. Organizations that rely on legacy systems—such as financial institutions, public agencies, and large enterprises—operate under strict requirements for availability, compliance, and security. These constraints make structural changes risky, as unexpected failures may lead to service disruption or regulatory impacts.

From a technical standpoint, legacy systems often depend on outdated languages, monolithic architectures, and limited automated testing, all of which increase the effort

required for safe and effective refactoring. These systems typically contain poorly documented code, obsolete frameworks, tightly coupled modules, and implicit behavior accumulated over decades. Additionally, knowledge gaps are common since original developers may no longer be available, and institutional memory is often incomplete.

Organizationally, there is constant pressure to balance innovation with operational stability. Development teams must sustain continuous delivery while managing extensive backlogs of technical debt. Budget constraints often prioritize short-term maintenance over long-term modernization, and decision-makers tend to be influenced by perceived risk rather than empirical evidence about refactoring effectiveness. This environment highlights the need for systematic empirical analysis, particularly in open-source settings where repository evolution can be directly observed.

2.3. Prior Work and Evidence (Internal and External)

Existing research underscores both the relevance and difficulty of refactoring in legacy contexts. Studies such as [Yanakiev et al. 2025] and [Ferreira et al. 2024] show that incremental refactoring can enhance maintainability and architectural consistency, though outcomes vary depending on organizational context and development practices. Empirical work indicates that refactorings in legacy systems are frequently intertwined with bug fixes, architectural erosion, and urgent maintenance tasks, complicating their evaluation.

Research also highlights the social dimension of refactoring. [Coelho et al. 2025] demonstrates that many refactorings arise during code review rather than through planned technical debt management. [AlOmar 2025] further observes that refactoring branches in large industrial repositories exhibit complex dynamics, and poorly structured refactoring efforts may lead to integration challenges or regressions. [Chen and Hayashi 2024] introduce optimization-based recommendation tools that consider reviewer availability, suggesting that organizational constraints significantly shape refactoring decisions.

External reports and industry investigations reinforce these findings. Large-scale audits and empirical studies reveal billions of dollars lost due to failed modernization initiatives. The IEEE Spectrum report indicates that approximately US\$720 billion has been wasted on unsuccessful attempts to replace or refactor legacy systems, emphasizing the need to understand the factors that differentiate successful refactorings from unsuccessful ones. Together, these works highlight that legacy refactoring remains an open and multifaceted challenge.

2.4. Essential Theoretical and Empirical Background

The theoretical foundation for this study is grounded in software maintenance and evolution theory, technical debt management, and empirical software engineering. Foundational models such as Lehman's laws of software evolution [Lehman 1980] indicate that software systems naturally degrade unless they are continuously adapted. Refactoring, as formalized by [Fowler 2018] and later expanded by [Mens and Tourwé 2004], is defined as a behavior-preserving transformation that improves internal software structure without modifying its external behavior.

Research on technical debt provides additional context. High levels of technical debt increase defect proneness, reduce adaptability, and inflate long-term maintenance

costs. Legacy systems commonly accumulate architectural debt, code smells, and duplicated logic, which heighten the complexity and risk associated with refactoring. Empirical studies show that low test coverage—a common characteristic of legacy systems—further increases the likelihood of regression failures following refactoring.

Recent empirical advancements incorporate socio-technical factors, revealing that the effectiveness of refactoring is influenced by communication patterns, workflow organization, and team coordination. Modern approaches include automated refactoring tools, multi-objective optimization techniques, and mining software repositories to identify real-world refactoring behaviors. These theoretical and empirical foundations inform the methodology of this study, which examines successful and unsuccessful refactorings across public GitHub repositories to better understand the practical challenges of modernizing long-lived software systems.

References

- AlOmar, E. A. (2025). Deciphering refactoring branch dynamics in modern code review: An empirical study on qt. *Information and Software Technology*, 177:107596.
- Chen, L. and Hayashi, S. (2024). Morcora: Multi-objective refactoring recommendation considering review availability. *arXiv preprint arXiv:2408.06568*.
- Coelho, F., Tsantalis, N., Massoni, T., and Alves, E. L. (2025). A qualitative study on refactorings induced by code review. *Empirical Software Engineering*, 30(1):17.
- Ferreira, F., Borges, H. S., and Valente, M. T. (2024). Refactoring react-based web apps. *Journal of Systems and Software*, 215:112105.
- Fowler, M. (2018). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 2 edition.
- Lehman, M. M. (1980). Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076.
- Mens, T. and Tourwé, T. (2004). A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139.
- Yanakiev, I., Lazar, B.-M., and Capiluppi, A. (2025). Applying solid principles for the refactoring of legacy code: An experience report. *Journal of Systems and Software*, 220:112254.