

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• v1.1 — Second delivery (25 Nov 2025) — Júlia Moreira• v1.2 - Third delivery (28 Nov 2025) - Júlia Moreira

1.4. Dates (creation, last update)

Field	Value
Creation Date	21 November 2025
Adding Sections	25 November 2025
Adding Sections	28 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.

3. Goals and Questions (GQM Framework)

3.1. General Goal (GQM Template)

Goal: Analyze refactoring activities in legacy software repositories for the purpose of understanding their impact on internal quality and maintainability from the perspective of researchers and software engineering practitioners in the context of open-source GitHub projects that contain long-lived and structurally complex codebases.

3.2. Specific Objectives

- O1:** Identify structural and code-quality characteristics that differentiate successful refactorings from unsuccessful ones.
- O2:** Evaluate how refactorings affect maintainability indicators such as complexity, code smells, and change-proneness.
- O3:** Analyze socio-technical factors (developer activity, review patterns) associated with refactoring outcomes.
- O4:** Assess the reliability of repository metadata and automated detection tools for evaluating refactorings in legacy systems.

3.3. Research / Business Questions

3.3.1. Objective O1: Structural Characteristics

- Q1.1:** What structural properties of the code change before and after refactoring?

Q1.2: How frequently do unsuccessful refactorings introduce regressions or defect-prone modules?

Q1.3: Which types of refactorings correlate with measurable improvements in internal quality?

3.3.2. Objective O2: Maintainability Impact

Q2.1: How do complexity metrics evolve after refactoring?

Q2.2: Does refactoring reduce the number of detected code smells?

Q2.3: What is the effect of refactoring on future change-proneness?

Objective O3: Socio-technical Factors

Q3.1: How does developer experience relate to refactoring success rates?

Q3.2: What review patterns appear in successful versus unsuccessful refactorings?

Q3.3: Does team activity intensity influence the likelihood of refactoring introducing defects?

3.3.3. Objective O4: Metadata and Tool Reliability

Q4.1: How accurate are automated refactoring-detection tools when compared to manual validation?

Q4.2: What types of false positives or false negatives occur in the detection process?

Q4.3: How trustworthy are commit messages and repository metadata to classify refactoring intent?

3.4. GQM Table

Goal	Questions	Metrics
3*O1: Identify structural characteristics	Q1.1	M1: LOC variation; M2: Cyclomatic complexity variation
	Q1.2	M3: Post-refactoring defect count; M4: Defect density
	Q1.3	M5: Refactoring type distribution; M6: Static analysis quality score
3*O2: Maintainability impact	Q2.1	M2: Complexity variation; M7: Halstead complexity shift
	Q2.2	M8: Code smell count; M9: Code duplication ratio
	Q2.3	M10: Change-proneness index; M11: Churn volume
3*O3: Socio-technical factors	Q3.1	M12: Developer experience level; M3: Defect count
	Q3.2	M13: Review comments count; M14: Review participation size
	Q3.3	M11: Churn volume; M3: Defect count
3*O4: Tool reliability	Q4.1	M15: Tool precision; M16: Tool recall
	Q4.2	M15: Precision; M17: False positive rate
	Q4.3	M18: Metadata consistency index; M19: Commit-message accuracy

Table 1. GQM Mapping Table

3.5. Metrics Table

Metric	Description	Unit
M1: LOC variation	Difference in lines of code before and after refactoring	LOC
M2: Cyclomatic complexity variation	Change in average complexity in affected files	Absolute
M3: Post-refactoring defect count	Number of defects reported after refactoring	Defects
M4: Defect density	Defects per KLOC	Def./KLOC
M5: Refactoring type distribution	Frequency of each refactoring type	%
M6: Quality score (static analysis)	Score given by static analysis tools	0–100
M7: Halstead complexity shift	Variation in Halstead metrics	Absolute
M8: Code smell count	Number of code smells detected	Count
M9: Duplication ratio	Proportion of duplicated code	%
M10: Change-proneness index	Frequency of future changes	Changes/mo
M11: Churn volume	Total added/removed lines after refactoring	LOC
M12: Developer experience level	Number of past contributions	Commits
M13: Review comments count	Number of review comments	Comments
M14: Review participation size	Number of reviewers involved	People
M15: Tool precision	$TP / (TP + FP)$	%
M16: Tool recall	$TP / (TP + FN)$	%
M17: False positive rate	$FP / \text{total detections}$	%
M18: Metadata consistency index	Alignment between metadata and actual refactoring	%
M19: Commit-message accuracy	Accuracy of commit intent descriptions	%

Table 2. List of Metrics

4. Scope and Context of the Experiment

4.1. Scope (Included and Excluded)

Included

- Mining GitHub repositories.
- Extracting refactoring data.
- Manual and automated validation.

- Artifacts: commits, diffs, PRs, static-analysis reports.
- Legacy modules with at least 5 years of activity and 500 commits.

Excluded

- Direct interaction with developers (no interviews).
- Proprietary/private repositories.
- Non-refactoring commits.
- Runtime or performance evaluation.

4.2. Study Context

- Organization type: Open-source, distributed.
- Project type: Legacy repositories (5 years).
- Criticality: Medium.
- Participant experience: Mixed, from novice to senior contributors.

4.3. Assumptions

- Repository history is not rewritten.
- Static-analysis and refactoring-mining tools operate stably.
- Public GitHub metadata remains accessible.
- Selected repositories follow version-control good practices.

4.4. Constraints

- Limited time for manual validation.
- Dependence on tool accuracy.
- GitHub API rate limits.
- No possibility of clarifying commit intent with developers.

4.5. Limitations

- Results may not generalize to proprietary systems.
- Metadata noise may affect classification.
- Refactorings mixed with feature changes may be difficult to isolate.
- Repository sample may not be fully representative.

5. Stakeholders and Expected Impact

5.1. Primary Stakeholders

- Software engineers.
- Researchers in software maintenance.
- Open-source maintainers.
- Technical leaders.
- Tool developers.

5.2. Stakeholder Interests

- Engineers: Evidence-based understanding of refactoring risks.
- Researchers: Empirical patterns for new studies.
- Maintainers: Reduce defects and technical debt.
- Tech leads: Better modernization decisions.
- Tool developers: Validation of detection tools in real-world contexts.

5.3. Impact on Process / Product

- Additional overhead in data preparation.
- Possible delays due to validation steps.
- Improved understanding of refactoring risks.
- More informed modernization strategies.

6. High-Level Risks, Assumptions, and Success Criteria

6.1. High-Level Risks

- GitHub API limitations.
- Inaccurate or incomplete metadata.
- Tool false positives/negatives.
- Time constraints.
- Large datasets requiring long processing times.

6.2. Global Success Criteria (Go/No-Go)

The experiment is successful if:

- 70% of detected refactorings are manually validated.
- Tool precision $\geq 75\%$ and recall $\geq 70\%$.
- At least 80% of research questions are answered.
- Clear patterns emerge distinguishing successful from unsuccessful refactorings.

Otherwise: **No-Go**.

6.3. Early Stop Criteria

- GitHub API becomes inaccessible.
- Detection tools fail or are unstable.
- Repositories are deleted or made private.
- Required computation or storage is not available.

7. High-Level Risks, Assumptions, and Success Criteria

7.1. High-Level Risks (business, technical, etc.)

Identify the main business and technical risks (delays, environment failures, data unavailability, etc.) at a macro level.

- Delays caused by long processing time of large repositories.
- Failures or instability in analysis environments or execution pipelines.
- GitHub API rate limiting or temporary unavailability.
- Missing, inconsistent, or incomplete repository metadata.
- High false-positive or false-negative rates in refactoring detection tools.
- Insufficient time for manual validation of detected refactorings.
- Loss of access to repositories (deletion, archiving, becoming private).
- Unexpected technical constraints such as insufficient storage or computing resources.

7.2. Global Success Criteria (Go / No-Go)

Define the conditions under which the experiment will be considered useful and viable.

Success Criteria (Go):

- At least 70% of detected refactorings can be manually validated.
- Tool precision $\geq 75\%$ and recall $\geq 70\%$.
- At least 80% of research questions receive meaningful, data-supported answers.
- Clear empirical patterns distinguishing successful from unsuccessful refactorings.
- All essential datasets and repositories remain accessible during the study.

No-Go Conditions:

- Inability to validate a significant portion of detected refactorings.
- Tool accuracy below acceptable thresholds.
- Inconsistent or inconclusive results that do not support meaningful insights.

7.3. Early Stop Criteria (Pre-Execution)

Describe conditions under which the experiment must be postponed or cancelled before it begins.

- Loss of access to repositories or persistent GitHub API instability.
- Technical failure of detection tools or inability to produce stable outputs.
- Unavailability of essential resources (computing power, storage, dataset access).
- Ethical concerns or violations of repository terms of use.
- Major changes in research scope or constraints that invalidate the experimental plan.

8. Conceptual Model and Hypotheses

8.1. Conceptual Model of the Experiment

The conceptual model assumes that characteristics of refactoring activities—such as the technique used, developer experience, and context of change—fluence outcomes such as defect introduction, maintainability improvement, and future change-proneness. Systematic refactoring guided by best practices is expected to reduce structural issues, while poorly validated refactorings may introduce regressions.

8.2. Formal Hypotheses (H_0 , H_1)

- $H_0^{(1)}$: There is no significant difference in code complexity before and after refactoring.
- $H_1^{(1)}$: Refactoring reduces code complexity relative to pre-refactoring versions.
- $H_0^{(2)}$: Refactoring does not change the number of post-change defects.
- $H_1^{(2)}$: Refactoring reduces post-change defect occurrence.
- $H_0^{(3)}$: Developer experience has no relationship with refactoring success rate.
- $H_1^{(3)}$: More experienced developers perform refactorings with higher success rates.
- $H_0^{(4)}$: Automated tools do not perform better than random classification.
- $H_1^{(4)}$: Automated detection tools achieve significantly higher precision and recall than random chance.

8.3. Significance Level and Power Considerations

The experiment adopts a significance level of $\alpha = 0.05$. The expected sample size from multiple repositories supports moderate statistical power for detecting medium to large effects. The goal is to maintain statistical power of at least 0.8 in comparisons where sample size is adequate.

9. Variables, Factors, Treatments, and Study Objects

9.1. Study Objects

- Commits involving refactoring.
- Code diffs before and after refactorings.
- Pull requests related to refactoring discussions.
- Static analysis outputs (complexity, code smells).
- Legacy modules with at least 5 years of activity.

9.2. Subjects / Participants (Overview)

Participants are the developers who contributed to the analyzed repositories, ranging from novice to senior maintainers. No direct recruitment occurs, as the study relies on repository mining.

9.3. Independent Variables (Factors) and Levels

Refactoring type: Extract Method, Rename, Move, Inline, etc.

Developer experience: Low, Medium, High (based on contributions).

Module complexity level: Low, Medium, High.

Repository activity context: Stable vs. High-Churn.

9.4. Treatments (Experimental Conditions)

Control Group: Non-refactoring commits in comparable modules.

Treatment 1: Refactorings classified as successful.

Treatment 2: Refactorings classified as unsuccessful.

Optional Treatment 3: Refactorings performed during high-churn periods.

9.5. Dependent Variables (Responses)

- Post-refactoring defect count.
- Variation in cyclomatic complexity.
- Change-proneness index.
- Code smell count.
- Structural metrics (LOC, duplication).
- Review activity metrics.

9.6. Control / Blocking Variables

- Repository age.
- Programming language.
- Test coverage category.
- Module domain (UI, backend, etc.).
- Module size and structure.

9.7. Potential Confounding Variables

- Developer motivation or workload.
- Differences in code-review rigor across teams.
- Commits mixing refactoring with feature changes.
- Tool accuracy variations across programming languages.
- Effects of project release cycles or freeze periods.

10. Experimental Design

10.1. Design Type

The experiment uses a partially blocked observational design:

- Blocked by repository to control project-level variance.
- Random sampling of refactoring and non-refactoring commits.
- Factorial-style comparisons for multiple metrics.

10.2. Randomization and Allocation

- Commits are randomly sampled from each repository.
- Refactorings are randomly selected for manual validation.
- All random assignments use a reproducible pseudorandom process with a fixed seed.

10.3. Balancing and Counterbalancing

- Balanced sampling ensures similar quantities of commits across repositories and refactoring types.
- Counterbalancing mitigates order effects during manual evaluation.
- Stratified sampling maintains proportional representation across project sizes.

10.4. Number of Groups and Sessions

- **Groups:** Control commits, successful refactorings, unsuccessful refactorings.
- **Sessions:** Each commit undergoes one evaluation session; evaluators may perform multiple sessions, but each commit is assessed once.

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.