UNIVERSIDADE FEDERAL DE PERNAMBUCO

CENTRO DE INFORMÁTICA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

DISSERTAÇÃO DE MESTRADO

RAFAEL BORBA COSTA DOS SANTOS

Fix or Rewrite: Navigating the Decision-Making in a Real Software Development Project

Recife

2024

RAFAEL BORBA COSTA DOS SANTOS

Fix or Rewrite: Navigating the Decision-Making in a Real Software Development Project

Projeto de Dissertação de Mestrado apresentado ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial à obtenção do título de Mestre em Ciência da Computação.

Mês/ano de ingresso: 08/2021

Área de concentração: Engenharia de Software

Linha de pesquisa: Engenharia de Software

Orientador(es): Prof. Dr. Sérgio Castelo Branco Soares

Coorientador(a): - Prof. Dr. Felipe Ebert

Recife

2024

ABSTRACT

This study investigates the decision-making process involved in choosing whether to fix or rewrite problematic software. It examines a case in which a development team was tasked with resuming a failed project, where critical bugs had prevented the software from going into production, leading to its abandonment. With limited information available, the team had to decide between attempting to fix the existing code or starting over with a complete rewrite. The research identified key factors that should have been considered when making this decision. Drawing from literature and case analysis, the study aims to provide guidelines for future decision-makers facing this challenge. The findings suggest that by analyzing key aspects such as the severity of bugs, low code maintainability, and poor adherence to requirements, the high probability of failure in the existing software could have been spotted, potentially preventing the team from working on an unfeasible codebase. A focus group was conducted, generating new hypotheses. Further research is needed to validate the conclusions across a broader range of cases and to propose a decision-making framework for similar situations.

SUMMARY

1 INTRODUCTION

Frequently, a software development project requires crucial decisions at the outset that might heavily influence its success. The challenge lies in the timing of these initial decisions, as they take place when the team has the least information about the context, scope and risks involved, insights that will only be better collected later on.

This study presents a case in which a development team was tasked with resuming a previous project that had failed in the past. Significant bugs were discovered, which the original developers were unable to fix. The new team was provided only with the source code and some initial requirements, and they had to decide whether attempting to fix those bugs or rewriting the entire program from scratch.

Which factors should they have had considered? What would've been the best decision? Was there an objective decision to be made? These are the questions we endeavor to answer.

Image 1: Illustration on the fixing vs. rewriting issue.

1.1 PROBLEM STATEMENT

The issue described on this study is similar to the maintenance versus replacement problem, which has been extensively studied [BENNATAN, 2000; BELADY and LEHAMN, 1985; SWANSON, 1976]. This field of research focuses on determining when it is no longer worthwhile to maintain legacy software and when rebuilding it is a better alternative.

Maintaining legacy software becomes increasingly difficult and time-consuming over time due to factors like the accumulation of technical debt and outdated technologies; however, rewriting it also presents concerns and risks, particularly in terms of costs and schedule overruns [BENNATAN, 2000].

Many researchers have attempted to identify the optimal time to cease evolving the old project and start working on a new one. As more enhancements are performed, legacy systems deteriorate and become more expensive to maintain [BELADY and LEHAMN, 1985]. Maintenance activities are broadly characterized as a sequence of corrective, adaptive, and perfective actions [SWANSON, 1976].

Although similar, the problem addressed in this study differs in several key aspects. First, problematic software is not exactly legacy software, as critical bugs have prevented it from going into production, whereas "legacy" typically refers to software that has been in use for many years. Second, fixing problematic software falls under a single type of maintenance (corrective) while maintaining legacy software often involves enhancements to keep up with ever-changing business rules. Finally, since not much time has passed since the software was written, outdated technology is not a primary concern. Nonetheless, the options remain the same: the existing code can either be evolved or abandoned, and once a decision is made and resources are allocated, reversing that decision becomes impractical.

Another related research field is software project failure [VERNER et al., 2008; KRISHNAN et al., 2004; PINTO et al., 1990]. A failure occurs when a development project does not meet its objectives, which can manifest in several ways, such as:

- Schedule overrun, causing delays in delivery.
- Budget overrun, leading to financial losses.
- Unmet requirements: incomplete, misunderstood or changing requirements.
- Poor quality: bugs, vulnerabilities or performance or usability issues.

In any case, the outcome of a failure is the project cancellation: the project is halted before completion, often due to critical issues, loss of trust or changes in priorities. Several factors can contribute to software project failures [VERNER et al., 2008], including:

- Poor project planning;
- Poor requirement management;
- Lack of stakeholder involvement;
- Human factors (team issues);
- Lack of risk management;
- Management issues, including ineffective leadership and oversights.

Understanding these factors and actively addressing them can help mitigate the risks of software project failures and increase the likelihood of success. Moreover, detecting these risk factors in the later stages of a project can help identify potential failures early, thereby preventing further losses.

The trade-off under analysis in this study can be stated as follows:

- *How can the best choice between fixing and rewriting problematic softwares be made, considering the risks associated with both alternatives, to maximize the probability of success?*

1.2 GOALS

The **general purpose** of this research is to investigate the literature on the fixing versus rewriting problem and summarize the main evidences and key findings available.

The **specific objective** is to assess whether there was an optimal decision to be made in the analyzed case and to write a summarized set of guidelines or takeaways applicable to this specific project. This can help future decision-makers facing similar situations and also provide insights for future research.

1.3 MOTIVATION

The problem brought to analysis was based on an R&D project sponsored by Canon Medical Systems do Brasil, a company specialized in medical equipment and nuclear medicine, in partnership with the Instituto Keizo Asami (iLIKA-UFPE), a non-profit organization that aims to produce and promote scientific knowledge and sustainable technologies.

The main objective of this project was to build a trustworthy and resilient system to ensure confidentiality, integrity, and availability in the transfer of medical images based on the Digital Imaging and Communications in Medicine Standards (DICOM). It was reported that prior to this partnership, several attempts had failed to achieve this goal. Although the project lasted 13 months in total, this case analysis focused solely on the initial decision-making phase.

As it will be demonstrated, there is a relative scarcity of studies addressing this specific problem, despite its potential significance in decision-making. In the case under examination, due to lack of best practices or supported guidelines, the practitioners opted for a middle-ground approach, dividing the team in two groups, one dedicated to understanding the flaws in the existing program, while the other simultaneously initiated a new project based on the requirements gathered.

The result was that the first group failed to resolve the issues with the original project within the allocated timeframe, leading to the termination of that branch. In essence, the task was completed: the team made efforts to rectify the old program, failed to make any progress, and moved forward, at the expense of some time and resources. However, ascertaining whether there was objectively a better decision to be made based on literature evidence may yield important guidelines for future reference.

1.4 LIMITATIONS

As stated previously, this study aims to synthesize the most useful guidelines and best practices found on the research topic when applied to the case under analysis and report them in a concise set of recommendations.

This guidance may or may not apply entirely to other projects. It does not intend to validate these assertions on a broader spectrum of cases or to establish a generic framework to support all decision-making processes, which are left as scope of future research.

In this chapter, we introduced the challenge of deciding whether to fix or rewrite problematic software, presenting the problem statement and its significance in software engineering. The case to be analyzed was contextualized within a failed software development project, highlighting the technical and managerial considerations inherent to such decisions.

The research goals, scopes, and limitations were outlined to ensure the study remains focused and its findings relevant. To advance toward potential insights, it is essential to explore the studied project in depth and understand the decision-making process. This will be the focus of the following chapter.

2 BACKGROUND

This section presents a brief explanation of the studied project and the challenges faced. The following description is a summarization extracted from the Requirements Document produced in partnership with the sponsor.

2.1 UPLOADER

Uploader is a codename for the studied program that we are using so forth. It consists of an application to share medical images between hospitals and clinics using DICOM. It should allow users to set routes between nodes to which the studies must be automatically sent. The final goal of the system is to make exams images quickly available across many points, where doctors, technicians and patients might need them, instead of having to download them from a centralized server when necessary.

Each LAN, whether inside a hospital, clinic, or diagnostic center, can have the Uploader installed on a centralized host (also known as a Gateway). This host is responsible for receiving DICOM studies produced by various modalities, such as Magnetic Resonance Imaging (MRI), X-ray machines, and Computed Tomography (CT) scanners.

The Gateway's job is simply forwarding these studies to the next connected Gateway. It cannot store locally the studies for longer than strictly needed to make sure that all connected recipients have received them. Also, all the connected Gateways need to establish a secure channel to exchange the files (such as a VPN), since sensitive data will travel through the internet.

After receiving studies from a neighbor Gateway, it is optional (can be configured) to also forward those studies to a local Picture Archiving and Communication System (PACS), in order to persist them and/or to integrate with other DICOM applications. Any DICOM compatible software can be used for that purpose, such as Orthanc and DCM4Chee[1].

[1] Orthanc (https://www.orthanc-server.com/) and DCM4CHee (https://web.dcm4che.org/) are both open-source implementations of DICOM servers widely used for managing medical images.

The user of *Uploader* is often the Hospital's Network Manager who would authenticate, setup the Gateways and local PACS and connect them using a graphical user interface. He also monitors the health of the network and eventually consults transfer logs.

Other minor requirements have been described but they are less relevant to this case analysis. The full Requirements Document can be found in this research repository[2] (original in portuguese, anonymized).

Each Gateway is composed of three components, as shown in Image 2:

- *Uploader T* (transmitter): responsible for sending studies to the next peer.
- *Uploader R* (receiver): receives the incoming studies and stores them in the PACS.
- *Uploader G* (manager): can be used to configure *Uploader T* and *R* and also to monitor the network and the status of the transfers.

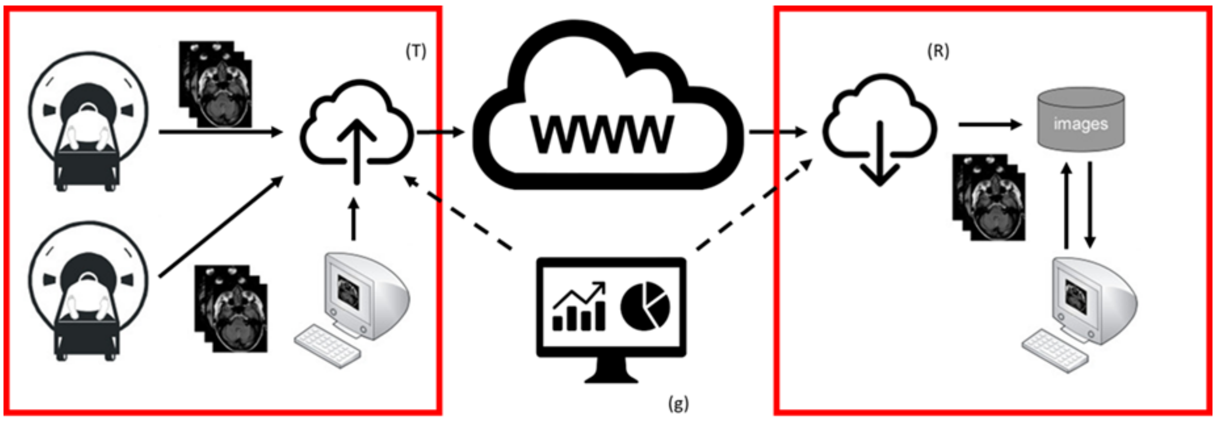

Image 2: *Uploader's* general architecture, its components T, G and R, and their interfaces.

Uploader T must integrate with the pre-existing hospital infrastructure and functions as a DICOM SCP, receiving studies via the C-STORE command specified in the DICOM standard. After receiving the studies, it will encrypt, compress and transfer them to the *Uploader R* of the next host, located on the network of a remote site.

Those three programs make up to 37 thousand lines of code (KLOC), which makes it a Medium-Large software project [OFFUTT et al., 1993]. The code is primarily written in

[2] https://github.com/rafaelborbacs/msc-fix-rewrite

Elixir, but other languages such as Java, Shell, Javascript and Python were used. The source code is proprietary and could not be disclosed.

2.2 THE CASE

At first glance, the software did not seem complex, and the team was confident that the errors could be corrected. However, after examining the test logs, the team discovered that the problems were more critical:

- Missing studies;
- Sudden stops and restarts;
- Incomplete transfers that paused for no apparent reason;
- Unsynchronized states, with the sender indicating a completed transfer while the receiver reporting otherwise;
- Duplicated studies;
- Duplicated series within studies.

The full Test Reports can be found in the repository (in Portuguese, anonymized).

As the kick-off approached, the managers had to decide between refactoring and rewriting. An agile development process had been chosen, so the scope of the first sprint was to be specified. Meetings were held to discuss that decision and several questions were raised:

- How easy is the code to understand and maintain?
- How well structured is its architecture?
- How well documented is the software?
- Which technologies were utilized?
- Does our team have any expertise with them?
- How serious are the errors found?
- Why were the original developers, presumably familiar with the code, unable to fix it?
- Can this project be classified as a software project failure?

There was a struggle to answer those questions due to lack of information about the project's previous history, once:

- No version control was available;

- No requirements document or user cases were created by the original team;
- The original programmers could not be contacted;
- No risks were assessed or monitored.

To summarize, the only reliable source of information were the code itself and the Tests Reports. On that account, a source code inspection took place. By then, it was obvious that there was no certainty about whether the program's malfunctions could be fixed in a reasonable amount of time. If they could be, then this probably would have been the best choice. Otherwise, the team might have found themselves trapped trying to refactor an infeasible codebase.

On the one hand, it seemed inefficient to waste the original software; but on the other hand, the refactoring effort could end up consuming even more time and energy. It was clearly a trade-off situation. A decision had to be made to maximize the probability of success, based on the project's subjective qualities, and the severity of the reported errors.

However, because no guidelines were available, the managers decided to split the difference. The development team was divided into two pairs of programmers, named Brownfield and Greenfield. The first group was responsible for trying to fix the old software, while the later was in charge of starting a new project to rewrite the program. The plan was to pick the best alternative after six sprints and abandon the less promising one.

The Greenfield vs. Brownfield approach originates from the fields of architecture and urban development, where these terms are used to compare the risks and costs associated with urbanizing a previously undeveloped, "virgin" area versus redeveloping land that was previously used for industrial or commercial purposes.

By choosing this strategy the managers accepted a certain loss in productivity in exchange of postponing the decision to the future when they would have more information. This was more of a practical decision than an empirically supported one.

Sprint	From	To	Scope/Goals
1	14/02/23	28/02/23	Greenfield: - studying DICOM protocol - testing Sonador platform locally Brownfield: - setting up a test environment

			- installing Uploader
2	01/03/23	14/03/23	Greenfield: - understanding Sonador Gateway - setting up a test dataset Brownfield: - testing Uploader transmissions - trying to reproduce the Tests Reports
3	15/03/23	28/03/23	Greenfield: - testing DICOM transmissions with Sonador Brownfield: - investigating Uploader’s sudden restarts - codebase inspection - planning further tests
4	29/03/23	11/04/23	Greenfield: - understanding Sonador’s strengths and limitations - setting up a full scale test environment Brownfield: - stuck on Uploader’s malfunctions and freezes - although successfully sent a few files, failed to complete a single full transmission - not able to refactor the entangled source code - Finally, decided to move on from the solution
5	12/04/23	25/04/23	- Proof of concept with third-party libraries: Pydicom, DCM4che and Orthanc
6	26/04/23	09/05/23	- Comparison of DCM4Che and Orthanc solutions
7	10/05/23	23/05/23	- DCM4Che x Orthanc: performance tests
8	24/05/23	06/06/23	- DCM4Che x Orthanc: integration and security tests
9	07/06/23	20/06/23	- Enhancements in both solutions
10	21/06/23	04/07/23	- Systematic tests in both solutions to asses error proneness and performance: DCM4chee wins
11	05/07/23	18/07/23	- Orthanc: live demonstration with the sponsors
12	19/07/23	01/08/23	- Development, integration and tests of VPN solutions
13	02/08/23	15/08/23	- DCM4che: live demonstration with the sponsors
14	16/08/23	29/08/23	- Cloud tests: AWS - POC containernet
15	30/08/23	12/09/23	- Orthanc: studies removal after transmissions - Dcm4che: mirror component to allow zero-visibility

			transmissions
16	13/09/23	26/09/23	- Dcm4che x Orthanc: overall comparison. - Decision made to adopt Orthanc and discontinue Dcm4che
17	27/09/23	10/10/23	- Orthanc: new architecture design: Uploader Manager API and View - Transmission logs
18	11/10/23	24/10/23	- Uploader Manager and View prototypes
19	25/10/23	07/11/23	- Uploader Manager API - Wireguard VPN integration
20	08/11/23	21/11/23	- Uploader Manager View
21	22/11/23	05/12/23	- Integration tests - Step-by-step config and tests demo
22	06/12/23	19/12/23	- Code refactoring - Full solution demo
23	20/12/23	03/01/24	- Transmissions metrics and statistics
24	04/01/24	16/01/24	- Local PACS forwarding - Admin role session and dashboards
25	17/01/24	30/01/24	- Final presentation to the sponsor
26	31/01/24	27/02/24	- Bugs fixing and reports writing

Table 1: displays the project's timeline and the sprints accomplishments and main decisions.

The timeline of the Table 1 shows how the sprints played out for the entire project. As shown, after four sprints of no progress, it was determined that the Brownfield failed and this branch was terminated prematurely. The two developers involved reported the most critical troubles they have found:

- Low familiarity with the Elixir syntax and logic;
- Lack of documentation, specially Software Requirements, including use cases, and Architecture and API specifications;
- High coupling/Low modularity of components;
- Confusing mix of different programming languages. In some cases, Elixir, Shell and Python codes were combined into a single functionality;
- Suspicion of potential issues within the Erlang/Elixir VM environment causing the crashes, without concrete evidence;

- Code looked excessivly large and complex for the intended use cases.

These factors were reported as potential contributors to the project's failure. However, the lack of documentation was unanimously identified as the primary cause of the code's low understandability and challenging maintainability.

In this chapter, we outlined the context, requirements, and challenges of the Uploader project, emphasizing the critical issues uncovered in the existing software and the resulting decisions that shaped the project's direction. Through a detailed analysis of the software's architecture and its underlying problems, the stage was set for the trade-off between attempting a fix or embarking on a complete rewrite. These foundational insights not only highlight the technical and managerial obstacles faced but also underscore the complexity of making informed decisions under uncertainty. The next chapter will delve into the methodologies employed to gather data and analyze the case, in order to evaluate the decision-making process.

3 METHODOLOGY

To accomplish the research goals a number of steps were needed:

- Organizing all data the practioners had available on the project, technologies, scope and contexts that could have influenced their decision;
- Gathering evidence from the literature on the subject;
- Applying it to the case studied to assess if there was a right decision to be made.

Those activities were elaborated and planned as illustrated in Image 3.

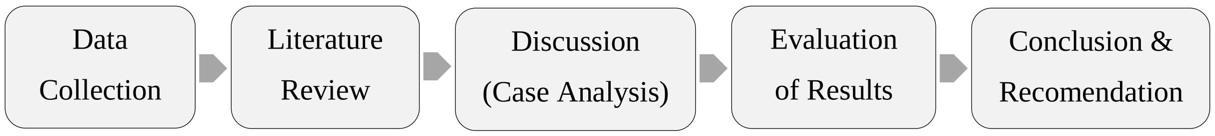

Image 3: the methodology planning.

3.1 DATA COLLECTION

The first work involved organizing all available data related to the projects, including the codebase, documents, emails, meetings records, and also the evaluation of technologies, scopes and contexts that could have influenced the decision-making. This was challenging as the data had to reflect what the practioners had available in that specific time:

- Project History: Collecting documents, sprint reports, e-mails and test reports. However, as it will be further explained, for the original project, given the lack of historic data and incomplete documentation, this step primarily relied on the Tests Reports and the codebase inspection.
- Chronological Organization: The data was then summarized into a timeline reflecting when the main activities took place, when the decisions were made and how they have impacted the team and the project.
- Scope and Objectives Definition: Clarifying the project's scope and objectives, including the critical functionalities required for the application under scrutiny and the specific issues encountered, such as missing studies, incomplete transfers, and synchronization problems.

3.2 LITERATURE REVIEW

The second step of this research was conducting a literature review to gather evidence on the fixing versus rewriting dilemma. This involved several key activities:

- Open search: The first step was to conduct an open search on Google Scholar to identify relevant keywords and construct a search query. The initial query included terms such as "software project failure," "technical debt," "refactoring," "rewrite," "risks," and "maintenance decision." The search aimed to discover other relevant jargons and keywords in order to elaborate a broader initial query.
- Rapid Review (RR): A Rapid Review methodology was chosen to investigate the literature due to its flexibility and efficiency in providing a controlled process to gather sufficient evidence quickly. This method allowed for a systematic yet expedited review. A RR is usually choosen over a traditional systematic review when a more flexible and less time consuming methodology is desired, while still providing a controlled process of obtaining sufficient evidence on the research question [CARTAXO et al, 2019].
- Backward Snowballing: To extend the dataset while executing the RR, backward snowballing was employed. Snowballing refers to a sampling method used in literature reviews. This approach is employed to identify relevant research papers on a particular topic. The idea is to start with a small set of well known or highly relevant papers and then use them as a "snowball" to find additional relevant sources by examining their references and citations. This method is empirically supported for enhancing the comprehensiveness of literature reviews.

The particular combination of using Google Scholar to put up an initial dataset and then snowballing to further extend the body of knowledge has been empirically supported [WOHLIN et al., 2022].

3.3 CASE ANALYSIS

The final step was applying the collected evidence to our case using narrative synthesis. This method combines findings from multiple studies in a coherent and descriptive manner, accounting for contextualization and interpretation. This included:

- Comparative Analysis: Comparing the practical problems and the challenges faced by the development teams with the evidence gathered from the literature.
- Decision Assessment: Evaluating whether the decision to divide the team into two groups (Brownfield and Greenfield) and the eventual abandonment of the Brownfield approach was supported by the literature findings.
- Guidelines Formulation: Developing a set of guidelines or takeaways based on the evidence to assist future decision-makers in similar situations.

In summary, the methodology combined data organization, a structured literature review, and the application of findings to a real-world case. This approach ensured a thorough grounded assessment of the fixing versus rewriting dilemma, providing valuable insights and guidelines for practitioners and researchers in the field of software development and maintenance. The use of Rapid Review and backward snowballing, along with a narrative synthesis, allowed for a comprehensive and efficient evaluation, addressing the specific case and its theoretical implications.

3.4 ETHICAL CONCERNS

It is important to recognize though that applying theoretical evidence to a concrete case might be tricky and bias susceptible.

It must be aknowledged that the solo researcher of the present work was part of the development team in the analyzed project, although not responsible for the managerial decisions. Also, his supervisor on this research worked as a technology consultant. These conflicting roles may impose threats to the validity of this study.

4 RAPID REVIEW

4.1 PRACTICAL PROBLEM

A company has reported critical issues with its recently developed software, preventing it from being deployed into production. A new team faces the decision between fixing this software and rewriting it from scratch.

4.2 RESEARCH QUESTIONS

Main question:

- *How can the best choice between fixing and rewriting problematic softwares be made, considering the risks associated with both alternatives, to maximize the probability of success?*

Additional secondary questions might aid in answering the main question by addressing similar trade-offs:

- *How can critical technical debt, major software risks and software failure (or bankruptcy) be detected in a software project?*
  Identifying these elements may help prevent wasting time and resources on code that is unfeasible to fix.
- *How does software complexity affect its maintainability?*
  High complexity, entanglement and low modularity may indicate poor maintainability, which favors rewriting.
- *How can the decision between maintaining a legacy system and replacing it be made?*
  The factors that lead to abandoning a legacy system may also apply to a buggy software that is beyond repair.

It is important to register that the secondary questions were evaluated only in the context of answering the main question.

4.3 METHOD

To gather relevant evidence to answer the questions, these steps were executed:

1. Conducting an open search on Google Scholar to discover relevant keywords;
2. Constructing a search query from the articles found;
3. Running the query on the same platform: relevant articles linked to one of the research questions were freely harvested to constitute an initial dataset;
4. Executing backward snowballing to collect related works that matched the selection criteria;
5. Reviewing the final dataset for evidences that addressed the questions;
6. Reporting the findings along with their applicability to the case.

4.4 SEARCH STRATEGY

Source:

- *Google scholar*

Initial search query used:

- *software ((project (failure OR bankruptcy OR "technical debt")) OR (problem rewrite (refactoring OR refactor)) OR (risks maintenance decision))*

4.5 SELECTION PROCEDURE

Only studies matching all the following criteria were accepted:

- Papers published in journals or conferences;
- Written in english;
- Available on Google Scholar;
- With title and/or abstract addressing at least one of the research questions.

To ensure the reliability and relevance of the literature, the following selection criteria were applied to the initial dataset:

- Peer-Reviewed Publications: Only papers published in peer-reviewed journals or conferences were included to ensure the reliability and academic rigor of the sources.
- Language: Articles had to be written in English to maintain consistency and comprehensibility.
- Availability: The papers had to be accessible through Google Scholar.
- Relevance: The title and/or abstract of the papers had to address directly or indirectly at least one of the research questions, focusing on software maintenance, refactoring, rewriting, risks assessing or project management.

No quality assessments were made on the selected studies through the forward snowballing process. The detailed steps on how each dataset was found and refined can be found in the research repository.

4.6 FINAL DATASET

This section presents the final dataset of selected studies for review, along with a brief explanation of their scope, main findings, and the reasons for their inclusion, particularly in the context of their potential to aid in answering the research questions.

Fairbanks (#1) discusses the trade-offs between ignoring, refactoring, or rewriting problematic software. It highlights key factors like code entanglement, outdated domain models, and the necessity of major architectural changes as decisive points for choosing rewriting. The findings suggest that rewriting can be less risky and more cost-effective when critical flaws are present, directly addressing the main research question by providing criteria to evaluate software risks and complexity for decision-making.

In Barua and Tridas (#2), the authors develop an analytical model to determine the optimal time to rewrite software. They argue that frequent modifications and increasing system complexity raise maintenance costs, making rewriting necessary. This study supports detecting critical technical debt and identifying when complexity becomes unmanageable, aligning with the secondary research questions.

Krishnan et al. (#3) introduces a decision model to optimize software maintenance strategies. It finds that significant rework reduces costs and improves code consistency. The insights on when maintenance becomes impractical contribute to evaluating risks and defining thresholds for initiating a rewrite.

Charette (#4) highlights predictable yet often ignored risks leading to software failure, such as poor communication and immature technology use. It underscores the importance of proactive risk management to avoid failure, linking to the detection of critical risks in projects as part of the decision-making framework.

Cerpa and Verner (#5) investigate risk factors and management flaws contributing to software project failures. It identifies the lack of stakeholder involvement and inadequate requirements management as key risks.

Abe et al. (#6) focuses on identifying causes of software project bankruptcy, including misjudgments and insufficient risk assessment. It proposes early detection methods for potential failures, aligning closely with the secondary question on detecting major software risks to guide maintenance versus rewrite decisions.

Verner and Cerpa (#7) analyze 70 failed projects, showing common factors such as unmanaged risks and inadequate planning. By highlighting the critical role of project management, this study aids in understanding the factors leading to abandoning buggy or overly complex systems.

Curtis et al. (#8) use metrics to measure software complexity, showing how unstructured and undocumented code hinders maintainability. The findings highlight the importance of complexity evaluation in determining when to rewrite software, linking to questions about the impact of software complexity.

Kafura et al. (#9) examine the relationship between software complexity metrics and maintenance activities, finding that structural deterioration correlates with increased difficulty in maintenance. These insights emphasize monitoring complexity and structuredness, helping address the decision to maintain or rewrite software.

Rombach (#10) evaluates the maintainability of structured versus unstructured software. It concludes that structured software is easier to maintain, suggesting that low modularity in software favors rewriting.

Yau and Collofello (#11) propose stability measures to predict the ripple effects of code changes, linking program stability to maintainability. Their methods provide practical tools for assessing whether fixing software is viable.

Chan et al. (#12) model maintenance and rewriting efforts, recommending avoidance of rewrites for large systems unless significant benefits outweigh costs. It provides economic criteria for rewriting decisions, directly addressing the risks and trade-offs.

Gode et al. (#13) presents an economic model for software replacement, linking complexity increases to optimal rewriting times. It provides an approach to compare costs and benefits, supporting decisions between maintaining and rewriting legacy software.

Bennett (#14) discusses challenges in migrating legacy systems and emphasizes the need for structuredness and modern technologies. It also provides insights into maintaining versus abandoning legacy systems.

Boehm (#15) emphasizes early identification and mitigation of software risks. His framework aligns with the secondary question on detecting critical risks and supports reducing uncertainties in maintenance versus rewrite decisions.

Pinto et al. (#16) identifies controllable factors in project failures, such as inadequate planning and unrealistic goals. It advocates for better project definition and risk assessment.

Pressman (#17) identifies poor project management as a primary cause of software failure and highlights the need for rookie managers to develop skills in communication, negotiation, organization, and facilitation.

These studies constitute the knowledge base to be reviewed for insights and recommendations to support the decision-making. In the next section, the findings on the subject are presented and discussed.

4.7 REVIEW REPORT

This report was created after investigating all the studies in the final dataset. It synthesizes key findings and recommendations for practitioners and researchers in the field, following the principles of narrative synthesis. The solo researcher collected, analyzed, and explained the key findings, addressing the research questions and objective, before synthesizing them into a coherent description. This process involved summarizing the findings, drawing conclusions, and providing interpretations and contextualizations.

The study #1 highlights the common approaches to handling software issues, which include ignoring the problems, making incremental changes through refactoring, or completely rewriting the software. However, it notes that ignoring is only an option when the problems are small enough. A notable observation is that engineers often display a natural inclination towards fixing things through refactoring, possibly due to familiarity or perceived efficiency.

The decision-making process is complex and often involves significant estimation and analysis. Key factors include estimating the time required for both fixing and rewriting, determining whether rewriting is feasible within the given budget and schedule, assessing the ability to precisely address the problems, and evaluating whether all necessary requirements are known to avoid replicating the old software. Generally, refactoring is seen as less risky and easier to integrate, but rewriting can be faster and more cost-effective.

Rewriting becomes the preferred option when the existing code is highly entangled, the domain model is outdated or incorrect, or when major changes in language, framework, or architecture are required. Ignoring the issues altogether is discouraged, as it can lead to further degradation of the codebase and communicate a lack of accountability, potentially worsening the situation (an idea supported by the broken windows theory).

While smaller problems are relatively easier to decide upon, large-scale software dilemmas are inherently complex. The study concludes that, in such situations, there is no definitive right choice. Instead, decision-makers should aim to maximize their chances of making the most favorable decision given the available information and constraints.

The study #2 emphasizes that as software systems grow in size, the complexity arising from increased control flows and inter-module interactions makes maintenance progressively more difficult. When a system undergoes a complete rewrite, it often becomes more modular, with fewer inter-module interactions, thereby reducing overall complexity and maintenance costs. This reduction is due to the restructuring and simplification that typically accompany rewriting.

Software complexity tends to increase over time as systems evolve, with structural deterioration driving higher maintenance costs. Compounding this issue is the lack of adequate documentation and personnel turnover, which exacerbates the difficulty of enhancing such systems. Consequently, as these challenges grow, the operational lifespan of a system often declines unless preventive measures are implemented to mitigate deterioration.

Conversely, adopting advanced technologies, such as fourth-generation programming languages (4GLs), can reduce development time and improve overall maintainability.

The study also suggests that switching to superior technologies tends to occur earlier than initiating a rewrite with the same technology, as organizations must seek to leverage advancements for greater efficiency. Despite the evident benefits of rewriting, many organizations persist with inefficient systems well beyond the optimal time for rebuilding, driven by inertia or the challenges of transitioning. This persistence is often linked to an oversight of critical costs, such as integration with existing applications and training staff on new systems, which are frequently neglected.

The study #3 underscores that while a major replacement of a software system may require a substantial initial investment, it often results in significant long-term benefits, including improved consistency and greater familiarity of the code among developers, which subsequently lowers maintenance costs. As software evolves, changes tend to increase its complexity, making maintenance progressively more challenging unless proactive measures are taken to counteract this deterioration. Preventive actions such as redesigning, updating documentation, and establishing robust change control mechanisms can enhance maintainability and slow down the degradation process.

The decision to perform a complete rewrite is typically made when maintenance becomes prohibitively expensive, reliability drops, responsiveness to changes slows, system performance falls below acceptable levels, or the system's functionality becomes outdated. The study notes that the structure of software deteriorates more rapidly as the frequency and volume of changes increase, leading to exponentially rising complexity. This phenomenon highlights the critical need for careful management of change rates.

Additionally, the study suggests that stochastic models can provide valuable insights into the optimal timing for replacing software. These models leverage historical data on system states and code changes to predict when continued maintenance is no longer viable, offering a data-driven approach to decision-making in software development.

The study #4 identifies several critical factors that commonly lead to project failures:

- Unrealistic or unarticulated project goals;
- Inaccurate estimates of needed resources;
- Badly defined requirements;
- Poor reporting of project’s status;

- Unmanaged risks;
- Poor communication among customers, developers, and users;
- Use of immature technology;
- Inability to handle the project’s complexity;
- Sloppy development practices;
- Poor project management;
- Inadequate stakeholder politics;
- Commercial pressures.

A particular danger for projects is the allure of adopting the latest, untested technologies in pursuit of a competitive edge. This "technological imperative" frequently leads to failure due to the immaturity or unsuitability of such technologies.

This work also emphasizes that poor project management is likely the single most critical cause of software failures, often manifesting as inadequate planning, lack of risk management, and an inability to handle complexity. Together, these factors underscore the need for robust project management practices and a clear understanding of both technical and organizational risks to enhance the likelihood of success.

The study #5 delves into management-related factors that frequently contribute to software project failures:

- Delivery date impacted the development process;
- Project under-estimated;
- Risks not re-assessed, controlled, or managed through the project;
- Staff not rewarded for working long hours;
- Delivery decision made without adequate requirements information;
- Staff had an unpleasant experience working on the project;
- Customers/Users not involved in making schedule estimates;
- Risk not incorporated into the project plan;
- Change control not monitored, nor dealt with effectively;
- Customer/Users had unrealistic expectations
- Process not reviewed at the end of each phase;
- Inappropriate development methodology for the project;
- Aggressive schedule affected team motivation;
- Scope changed during the project;

- Schedule had a negative effect on team member’s life;
- Project had inadequate staff to meet the schedule;
- Staff added late to meet an aggressive schedule;
- Inadequate time for requirements gathering.

The study #6 examines the phenomenon of software project bankruptcies and identifies key factors that lead to such outcomes. It highlights that most bankruptcies are revealed during the testing phase, a stage where critical failures and the inability to meet requirements become apparent. A primary cause of these failures is misjudgment, which can stem from inadequate initial case studies, insufficient fact-finding efforts, and reliance on overly optimistic reports.

This misjudgment often results in critical decisions being based on incomplete or inaccurate data, setting the stage for project failure. The study underscores the importance of thorough and realistic analysis during the planning and early execution stages. By improving fact-finding processes and ensuring accurate reporting, organizations can significantly reduce the risk of software bankruptcy and better manage the challenges associated with complex software development projects.

The study #7 explores a comprehensive range of factors contributing to software project failures, emphasizing that these failures often result from a combination of technical, project management, and business-related issues. High staff turnover emerges as a recurring challenge, disrupting continuity and increasing project risks. Other significant factors include:

- organizational structure;
- unrealistic goals;
- software that fails to meet the real business needs;
- sloppy development practices;
- inadequate scheduling and project budget;
- inaccurate estimates of needed resources;
- inability to handle project complexity, unmanaged risks;
- use of immature technology;
- neglected customer satisfaction;
- weak product quality management;
- absense of leadership, upper management support;
- personality conflicts;

- business processes and resources;
- poor, or no tracking tools.

The study highlights that excessive rework often signals failure, particularly when costs exceed the value-added work or when inadequate planning and specifications lead to frequent change requests. This results in schedule and budget overruns, ultimately undermining the project's viability. Furthermore, failure can stem from underestimated project scope, poor understanding of that scope, lack of stakeholder involvement and conflicts within the team.

The study #8 focuses on the impact of software complexity on maintainability and performance, introducing various metrics to assess this complexity. Among these, the frequency and distinctness of operators and operands, decision tree length, and the number of independent control paths within a program are highlighted as key measures of complexity. It was observed that the number of statements in a program is strongly correlated with the ease of understanding and modifying the code, making this a practical metric for evaluating software maintainability.

Furthermore, the study confirms that as software undergoes repairs and updates, both code and structural complexity tend to increase. This compounding effect makes the system progressively harder to maintain over time. The findings underscore the importance of monitoring internal complexity metrics, as these have a more significant influence on maintenance efforts than external factors such as implicit information flows.

The study #9 investigates the relationship between software changes and increasing complexity, also emphasizing how repairs and modifications can lead to structural and code-level deterioration. It finds that each repair effort not only increases the overall metrics of complexity but also affects the structural integrity of the system, making it harder to maintain with each version. This pattern of degradation aligns with broader findings that software systems become more challenging to maintain over time.

It also highlights that complexity is not evenly distributed across a system. Instead, the combined effects of numerous changes often result in non-localized impacts, which can disrupt the overall structure and cohesiveness of the software.

The study #10 explores the role of software structure in determining maintainability, revealing that structured systems are significantly easier to maintain compared to their unstructured counterparts. It emphasizes that internal complexity metrics, such as code length and structure, are the most reliable predictors of the effort required for maintenance tasks.

The study #11 examines how program stability influences software maintainability, demonstrating that design choices significantly affect the ease of maintenance. Programs incorporating data abstractions are generally more stable and easier to maintain than those lacking such structures. As complexity increases, the understandability of the program diminishes, leading to greater challenges in maintaining the software. Programs with lower complexity not only exhibit better stability but also allow for easier identification and resolution of issues.

The study #12 provides an economic perspective on the decision-making process for software rewriting, emphasizing the challenges associated with large-scale applications. It advises against complete rewrites for large systems unless the expected benefits significantly outweigh the costs. This is because a substantial portion of the effort in rewriting is spent redeveloping existing functionalities, which may not justify the expense.

The analysis highlights the importance of assessing the scale of the software and the economic implications before committing to a rewrite. For smaller systems, where functionality and complexity are limited, rewriting might be more feasible. However, for larger applications, the study underscores the need to weigh the potential gains against the extensive effort required to rebuild foundational elements.

The study #13 explores the economics of software replacement, emphasizing that as complexity increases, the optimal time to rewrite a system often becomes apparent. It argues that structured code not only facilitates maintenance but also delays the need for a complete rewrite by improving system adaptability and maintainability. By focusing on structured programming practices, organizations can mitigate the challenges associated with escalating complexity and extend the lifespan of their software.

The study #14 identifies the top risk factors commonly encountered in software projects and outlines effective risk management techniques to address them. Key risks include personnel shortfalls, unrealistic schedules and budgets, developing the wrong functionalities or user interfaces, and frequent changes in requirements. It highlights that effective mitigation strategies such as detailed cost and schedule estimation, incremental development, prototyping, and fostering team collaboration can significantly reduce these risks.

The study also addresses challenges like gold-plating, where unnecessary features inflate project scope, and the use of immature technologies. It recommends rigorous requirements scrubbing, cost-benefit analysis, and early user participation to ensure that projects stay focused and viable. Additionally, it emphasizes the need for robust risk assessment and management practices, including benchmarking, simulation, and comprehensive pre-project analysis, to avoid pitfalls such as performance shortfalls or excessive reliance on unproven components.

The study #15 emphasizes the importance of identifying and addressing risks early in the software development process. It argues that proactive risk management not only reduces long-term costs but also prevents potentially disastrous outcomes. By mitigating risks at the earliest stages, project teams can avoid many of the challenges that typically arise during later phases, such as escalating costs, missed deadlines, and unmet requirements.

The study highlights that effective risk management is a foundational practice for successful software development. It encourages practitioners to incorporate structured risk identification and mitigation strategies into their workflows to enhance project stability and ensure a smoother path to completion. This proactive approach serves as a safeguard against the unpredictable nature of software projects, where unmanaged risks can spiral into failure.

The study #16 identifies ten critical success factors for software projects:

- Project Mission: Initial clearly defined goals and general directions.
- Top Management Support: willingness of top management to provide the necessary resources and authority/power for project success.
- Project Schedule/Plan: a detailed specification of the individual action steps for project implementation.
- Client Consultation: communication, consultation, and active listening to all impacted parties.
- Personnel: recruitment, selection, and training of the necessary personnel for the project team.
- Technical Tasks: availability of the required technology and expertise to accomplish the specific technical action steps.
- Client Acceptance: the act of “selling” the final project to its ultimate intended users.
- Monitoring and Feedback: timely provision of comprehensive control information at each stage in the implementation process.

- Communication: the provision of an appropriate network and necessary data to all key actors in the project implementation.
- Trouble-shooting: ability to handle unexpected crises and deviations from plan.

Finally, the study #17 also examines reasons for software project failures, attributing many of them to poor project management practices. It states that inadequate experience among project managers often leads to unrealistic goals, poor scheduling, and insufficient communication. These shortcomings result in frequent changes to the project scope, mismanagement of risks, and unaddressed issues during critical stages of development. Another factor contributing to failure is the lack of tools and methodologies tailored to track progress effectively and manage complexity.

5 DISCUSSION

The findings reported in the last chapter are now discussed through evidence briefings, presented in a storytelling narrative format. This aims to provide an overview of the evidences when applied to the studied case.

Deciding whether to fix the existing code or to rewrite it is a significant decision that depends on various factors. In this situation, usually there are three options: doing nothing (ignore), making incremental changes (refactor) and writing a new program from scratch (rewrite). Although, this statement is valid only when the problems are small enough that can be ignored [FAIRBANKS, 2019]. The first and most obvious conclusion is that ignoring was not an option in our case.

Also, the presence of many critical errors reported was an indicative of fundamental design flaws [BARUA and TRIDAS, 1989; BENNETT, 1995], which would be confirmed by the detection of entanglement and low modularity in the codebase inspection [FAIRBANKS, 2019].

Upon analyzing the scenarios, if fixing the program were chosen as the path forward, it would require a series of maintenance tasks to ensure the software becomes compliant and coherent. Maintenance can be classified into adaptive, perfective, and corrective types [CHAN et al., 1996]. In this study, we focused on corrective maintenance, which is typically triggered by software failures detected during testing or operation.

Once a particular maintenance objective is established, the team must first understand what they are to modify. They must then modify the program to satisfy the maintenance objectives. After modification, they must ensure that the modification does not affect other portions of the program. Finally, they must test the program. The following aspects of a software were found to be important to execute corrective maintenances [ROMBACH, 1987]:

- <u>Maintainability</u>: the effort in staff-hours per maintenance task;
- <u>Comprehensibility</u>: the isolation effort (effort to decide what to change) in staff-hours per maintenance task, or the average amount of rework (all effort spent for changing already existing documents such as requirements, designs, code, or test plans) per system unit as a percent of all effort spent per unit throughout the lifecycle;
- <u>Locality</u>: the number of changed units per maintenance task, or the maximum portion of the change effort spent in one single unit per maintenance task;

- Modifiability: the correction effort in staff-hours per maintenance task and unit;
- Reusability: the amount of reused documentation as a percent of all documentation per maintenance task.

However, without a version tracking, it was challenging to understand the code's history, changes, and the rationale behind them. The fact that the code was medium-large sized made the decision even harder [FAIRBANKS, 2019].

Another recurrently studied factor is the psychological complexity, which refers to characteristics of the software that make it difficult to understand and work with [GODE et al., 1990]. There is a large number of complexity metrics, such as KLOC, quantity of variables, interfaces and different logical paths. However, to analyze software complexity metrics objectively, in this particular case, has shown to be less effective because of, once again, lacking of history. Most of the studies try to correlate those metrics with past behavior, for example, to address how hard it is expected to perform a maintenance task in one complex module in comparison with another [CURTIS et al., 1979], or to predict programming times comparing one to the next version of a given software increasingly more complex [KAFURA et al., 1987].

Since there was no version backlog and the previous history was unknown, a baseline to estimate code refactoring and bug fixing activities wasn't available. Although, it was reported by the practioners that the code seemed too large for its use case, which is a weaker evidence but does suggests high complexity and leads to harder maintenance [FAIRBANKS, 2019] and low understandability [YAU and COLLOFELLO, 1980].

Furthermore, the absense of software documentation (specially a Requirements Document, including Use Cases, and Risks Management) was a red flag as well, since it is evidence of difficulty to enhance and maintain systems [FAIRBANKS, 2019; KAFURA et al., 1987] and improved overall risks [ABE et al., 1979]. That also indicates poor project management which has been pointed out as the single greatest cause of software failures [KRISHNAN et al., 2004; PRESSMAN, 1998].

Adittionally, choosing Elixir as main technology can be viewed as a questionable decision, as the use of immature technologies is linked to project failures and low maintainability [KRISHNAN et al., 2004, ABE et al., 1979]. Well established languages like

Python, C++, Java and Javascript have dozens of times the usage of Elixir[3] and could be better suited.

To complete the decision, it was necessary to assess the success/failure chances of the poject. Software project failure is not a rare event. In fact, it accounts for over 30% of the projects [ABE et al., 1979]. Software failue can be defined as the total abandonment of a project before or shortly after it is delivered, or as a synonym of Software Bankruptcy, when it is acompanied with heavy financial damage and/or loss of reputation by not meeting the target date or an excess over the budget by approximately 20% [KRISHNAN et al., 2004].

The search for factors that influence the project's success or failure has been of great interest to both researchers and practitioners. One stream of work focus on developing decision rules and/or decision support systems to aid in making systematic decisions on whether projects should be terminated [PINTO et al., 1990]. As mentioned, there're two kinds of failure (or bankruptcy): expenditure over the budget and not meeting the target date. The studied case fits the later.

Again, the absense of basic planning, design and developemnt documents plays a huge role in improving chances of failure. We found evidence that this Code-Driven development process induces high-risk commitments. It tempts people to say "Here are some neat ideas I'd like to put into this system. I'll code them up, and if they don't fit other people's ideas, we'll just evolve things until they work." This sort of approach usually works fine in some well-supported minidomains but, in more complex application domains, it most often creates or neglects unsalvageable high-risk elements and leads the project to disaster. [BOEHM, 1991]

To summarize, we verified in the case studied several failure-linked factors reported in the review [CHARETTE, 2005; CERPA and VERNER, 2009; ABE et al., 1979], such as:

- Inaccurate estimates of needed resources;
- Badly defined requirements;
- Poor reporting of project's status;
- Unmanaged risks;
- Use of immature technology;
- Inability to handle the project's complexity;
- Sloppy development practices;

[3] Source: https://madnight.github.io/githut

- Poor project management;
- Project under-estimated;
- Risks not re-assessed, controlled, or managed through the project;
- Delivery decision made without adequate requirements information;
- Risk not incorporated into the project plan;
- Change control not monitored, nor dealt with effectively;
- Inappropriate development methodology for the project.

Although we found advice to avoid complete rewrite when the application concerned is large, because much of the effort will be expended on redeveloping the initial software functionality [GODE et al., 1990], the requirements specifications were not as large, which, again, indicated poor design/implementation and indicated that a better structured code could be easier to understand and maintain.

A final thought is that not all requirements elicited were present in the original project, so even if the bugs could be corrected, enhancements would still be needed. In this scenario, a total replacement, even requiring significant investment, such a rework would also improve consistency and increase familiarity of the code to the developers, which would lower further maintenance costs. Hence, we found evidence that rewriting would have been the less risky option for our specific case.

6 EVALUATION OF RESULTS

This chapter presents an evaluation experiment conducted to understand the perceptions of practitioners involved in the studied project regarding the findings previously presented. We made them revisit the decision-making process, now aware of the evidences found in the literature review, and evaluate whether or not they would come to similar conclusions.

6.1 FOCUS GROUP

Inspired on [KAMEI, 2022], a focus group was chosen to evaluate the decisions made in the project about whether fixing or rewriting the software, based on findings from the literature review and the project's outcomes. This assessment were conducted with practitioners involved in the project to gather their insights and agreement with the conclusions made, and aims to bridge the gap between academic research and practical experience, providing insights for future decision-making. It also intended to obtain a broader, richer, and more collaborative perception of the evidences found when applied to this case and possibly stretch the guidelines or yield new hypotheses for future research.

According to [MORGAN, 1996], the focus group method is a research technique used to collect data through group interactions on a topic determined by the researcher. In this research, we followed the process proposed by [KONTIO et al., 2008].

6.2 FOCUS GROUP PLANNING

As mentioned, our final goal was to assess whether the participants would come to similar conclusions by learning the findings collected in the literature review. Therefore, we began our research by defining the research questions as:

- *What are the perceptions of the practitioners involved in the studied project on the findings and recommendations presented?*
- *If the decision were to be made again, would they consider a different approach?*

In order to plan the group session, the first step was to define Roles. The **Moderator-Researcher** was responsible for facilitating the interaction and discussions between the **Participants**, following a predefined questioning structure. Moreover, this researcher was responsible for allowing the integration and expression of opinions and making interventions when the discussion encountered conflicts. The author of this study played this role.

After having the roles established, this focus group structure was draw. The session was planned to begin with the Moderator-Researcher explaining how the focus group would work and describing the problem under evaluation. For each subject investigated, each participant was encouraged to write down their thoughts, viewpoints, and whether or not they agreed. Then, by stating open questions, the Moderator-Researcher created space for the intended discussion. The whole session was recorded for subsequent data extraction and analysis. Here's a breakdown of the session:

- **Introduction** (2 minutes)
    - Welcomed participants and introduced the purpose of the focus group.
    - Explained the structure and rules for the discussion.
    - Ensured confidentiality.
- **Project Recap** (5 minutes)
    - Presented the project's timeline and reminded participants of the decisions made and the outcomes produced.
    - Asked the first open question:
        - Do you agree that was the best decision at the time?
- **Presentation of Findings** (10 minutes)
    - Presented a concise summary of the key findings from the literature review as described in the previous chapter.
- **Discussion** (20 minutes)
    - Asked participants for their initial reactions to the presentation.
    - Asked the second open question:
        - Would you change the decisions made in the project? How?
- **Conclusion** (1 minute):
    - Explained the next steps and how their feedback would be used.

6.3 SELECTION OF PARTICIPANTS

It is recommended ensuring the participants have the appropriate knowledge to participate in a focus group session [FRANÇA et al., 2015]. Although it is conventional to make focus groups with between six and twelve participants, for this study, given the small size of the team and the research specific goals, we gathered only two (but relevant) participants. We selected the two practitioners directly involved in the decision-making:

- Participant A: The Team's Project Manager; and
- Participant B: The Tech Leader of the Sponsor.

6.4 CONDUCTING THE SESSION

The meeting was held remotely in Portuguese in early August 2024. It lasted 40 minutes, as planned. In addition to the moderator and the two participants, we included an outside observer who contributed to enriching the discussion by raising questions and commenting on the participants' inputs after they had finished.

6.5 RESULTS

To assist with data extraction, we used an online version of Microsoft Word (Office 365) with the transcription feature. The transcription can be found in the research repository. We then synthesized the data collected into a coherent narrative, using descriptions, summarization and direct quotes. The final result was the following Summary Report highlighting the participants' perspectives.

After the first open question was posed, both participants agreed that the decision made in the project was correct and reasonable. They acknowledged that an attempt to address the issues was necessary, even though the reported errors seemed too critical to be resolved and the overall chances of success seemed low.

Participant B noted that the project was challenging and justified the decision to split the team, with half of the development resources focused on understanding what had been previously developed. However, he stated that the timeframe allocated for this task (six sprints, or three months) was excessive.

> *"In my view, a slightly shorter timeframe would have been ideal for attempting to gain knowledge from the legacy system before moving on" Participant B.*

Participant A added that setting a clear deadline in this situation was crucial to avoid indefinitely postponing the decision. However, he acknowledged that, in retrospect, it is difficult to determine whether the allocated timeframe was excessive since the outcome of that decision is already known.

In the second part of the session, after receiving a concise review of the findings and conclusions discussed in the previous chapter, the participants expressed their thoughts.

Participant B stated that he wouldn't change the decision to try to repair the old software, despite the undeniable number of failure-linked factors present in the project. In his opinion, an attempt was still required, but the risk factors discovered could have been used to shorten the timeframe allocated for this endeavor, given the high probability of failure.

> *"There was an insistence on justifying the decision, but in my view, we could have reached the same conclusions a little more quickly based on the premises presented" Participant B.*

On the other hand, Participant A said that applying this method made it clear that rewriting the software was a more reasonable path. However, he suggested that the effort to fix the old software provided a better understanding of the system's domain.

> *"Perhaps trying to make sense of buggy software accelerates the team's process of gathering knowledge" Participant A.*

This unexpected viewpoint challenges the idea that resources spent on fixing a system are wasted when the software ultimately needs a rewrite. An interesting hypothesis for future work could be that attempting to fix problematic software might actually shorten the overall project timeline by accelerating the team's learning curve on the problem domain and consolidating lessons learned and "things to avoid" in the new project.

Additionally, Participant B emphasized that in this particular project, the factor of "immature technology" was key, as nobody on the team had knowledge of it, and acquiring this expertise would be difficult. Due to the lack of community support, the bug-fixing attempt was likely to fail.

Finally, both participants agreed that a generic framework to semi-automate similar decisions could be developed using statistical data on a large set of projects. Such a framework could speed up the decision-making process by assessing the probabilities of success or failure of any project based on its inherent qualities.

> *"Opting to rewrite the program might suggest high costs, but it can also lead to significantly lower risks. The framework has to consider both cost and risk metrics" Participant A.*

This tool would reduce the pressure on decision-makers by allowing them to make objective decisions based on evidence and statistics. Building such a framework may be challenging but is also one of the most promising goals in this research field.

.

7 CONCLUSION

Software projects will always call for trade-off decisions because of human factors and non-deterministic events that lead to not fully predictable results. When faced with the fixing versus rewriting dilemma, all the practioner can do is to investigate the current stage of the project, apply the best guidelines available, make a decision and hope for the best [FAIRBANKS, 2019].

We propose a breakdown of the most usefull guidelines applicable to our case:

- Severity and scope of errors: if the bugs are small, isolated and the affected module can be specified, fixing may be more efficient; if the bugs are pervasive and affect critical functionalities, rewriting is most likely necessary.
- Code quality and maintainability: high-quality, well-structured and well-documented codes are easier to fix; poorly written, undocumented and entangled codes are cost-effective to rewrite.
- Technical debt: if the software has accumulated technical debt, rewriting can offer a fresh start, reduce maintenance costs and enhance developers familiarity. Those advantages usually pay off in the beginning of the project.
- Requirements: if the current software struggles to meet all the known requirements, rewriting can provide a more robust foundation, because enhancements will be necessary anyway.
- Technological advancements: if the existing software relies on outdated or immature technologies, rewriting can leverage modern tools and more tested frameworks.
- Team expertise: the availability of developers who are proficient in the existing codebase is crucial; if developers who can understand quickly the code are scarse, building a new code can be time-saving.

In this case, we demonstrated that, given the available information, opting for rewriting would have been the less risky alternative, despite some benefits of attempting a fix. While there is not always a definitive answer, the checklist above can help guide similar decisions. Further research could validate these findings across a broader range of cases and quantify how these factors individually influence the final decision.

8 REFERENCES

BENNATAN, E.M. "On Time Within Budget, John Wiley and SODS" (2000).

BELADY, LASZLO A., and M. M. LEHAMN. "Program evolution: Processes of software change." *Academic Press*, (1985).

SWANSON, E. BURTON. "The dimensions of maintenance." Proceedings of the 2nd international conference on Software engineering. (1976).

VERNER, JUNE, JENNIFER SAMPSIN, and NARCISO CERPA. "What factors lead to software project failure?" 2008 second international conference on research challenges in information science. IEEE, (2008).

OFFUTT, A. JEFFERSON, MARY JEAN HARROLD, and PRIYADARSHAN KOLTE. "A software metric system for module coupling." Journal of Systems and Software (1993).

CARTAXO, BRUNO, et al. "Software engineering research community viewpoints on rapid reviews." ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). IEEE, (2019).

WOHLIN, CLAES, et al. "Successful combination of database search and snowballing for identification of primary studies in systematic literature studies." Information and Software Technology 147 (2022).

FAIRBANKS, GEORGE. "Ignore, refactor, or rewrite." IEEE Software 36.2 (2019): 133-136.

BARUA, ANITESH, and TRIDAS MUKHOPADHYAY. "A cost analysis of the software dilemma: to maintain or to replace." Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences. Volume III: Decision Support and Knowledge Based Systems Track. IEEE (1989).

KRISHNAN, MAYURAM S., TRIDAS MUKHOPADHYAY, and CHARLES H. KRIEBEL. "A decision model for software maintenance." Information Systems Research (2004).

CHARETTE, ROBERT N. "Why software fails [software failure]." IEEE spectrum (2005).

CERPA, NARCISO, and JUNE M. VERNER. "Why did your project fail?." Communications of the ACM (2009).

ABE, JOICHI, KEN SAKAMURA, and HIDEO AISO. "An analysis of software project failure." Proceedings of the 4th international conference on Software engineering. (1979).

CURTIS, BILL, et al. "Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics." IEEE Transactions on software engineering (1979).

KAFURA, DENNIS and GEEREDY R. REDDY. "The use of software complexity metrics in software maintenance." IEEE Transactions on Software Engineering 3 (1987).

ROMBACH, H. DIETER. "A controlled expeniment on the impact of software structure on maintainability." IEEE Transactions on Software Engineering 3 (1987).

YAU, STEPHEN S., and JAMES S. COLLOFELLO. "Some stability measures for software maintenance." IEEE Transactions on Software Engineering 6 (1980).

CHAN, TAIZAN, SIU LEUNG CHUNG, and TECK HUA HO. "An economic model to estimate software rewriting and replacement times." IEEE Transactions on Software Engineering (1996).

GODE, DHANANJAY K., ANITESH BARUA, and TRIDAS MUKHOPADHYAY. "ON THE ECONOMICS OF THE-SOFTWARE REPLACEMENT PROBLEM." (1990).

BENNETT, KEITH. "Legacy systems: Coping with success." IEEE software (1995).

BOEHM, BARRY W. "Software risk management: principles and practices." IEEE software (1991).

PINTO, JEFFREY K., and SAMUEL J. MANTEL. "The causes of project failure." IEEE transactions on engineering management (1990).

PRESSMAN, ROGER. "Fear of trying: the plight of rookie project managers." IEEE software 15.1 (1998).

KAMEI, FERNANDO KENJI. "Understanding and supporting the decision-making whether to use grey literature in software engineering research." (2022).

MORGAN, DAVID L. "Focus groups." Annual review of sociology (1996).

KONTIO, JYRKI, JOHANNA BRAGGE, and LAURA LEHTOLA. "The focus group method as an empirical tool in software engineering." Guide to advanced empirical software engineering. Springer London (2008).

FRANÇA, BRENO BERNARD NICOLAU, et al. "Using Focus Group in Software Engineering: lessons learned on characterizing software technologies in academia and industry." CIbSE (2015).