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.

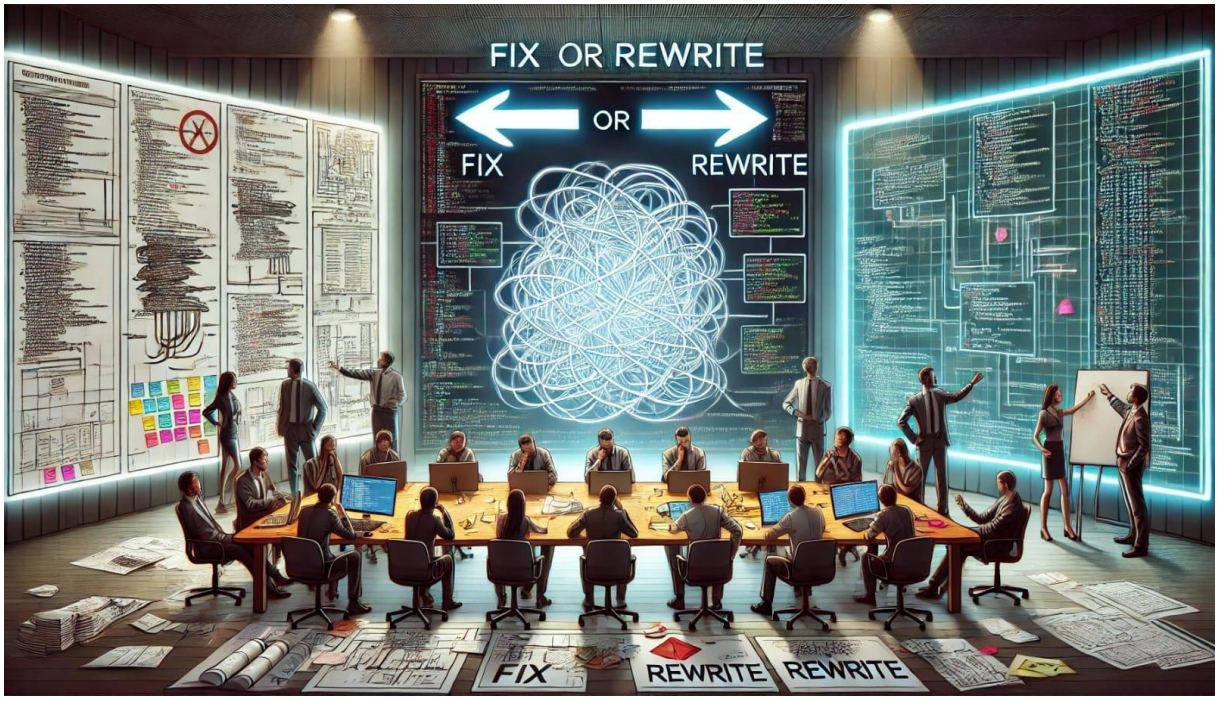

Image 1: AI-generated illustration on the fixing vs. rewriting issue.

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?*

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 Brownfield made efforts to rectify the old program, failed to make any progress, and moved forward, at the expense of 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.

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.

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 (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 M* (manager): can be used to configure *Uploader T* and *R* and also to monitor the network and the status of the transfers.

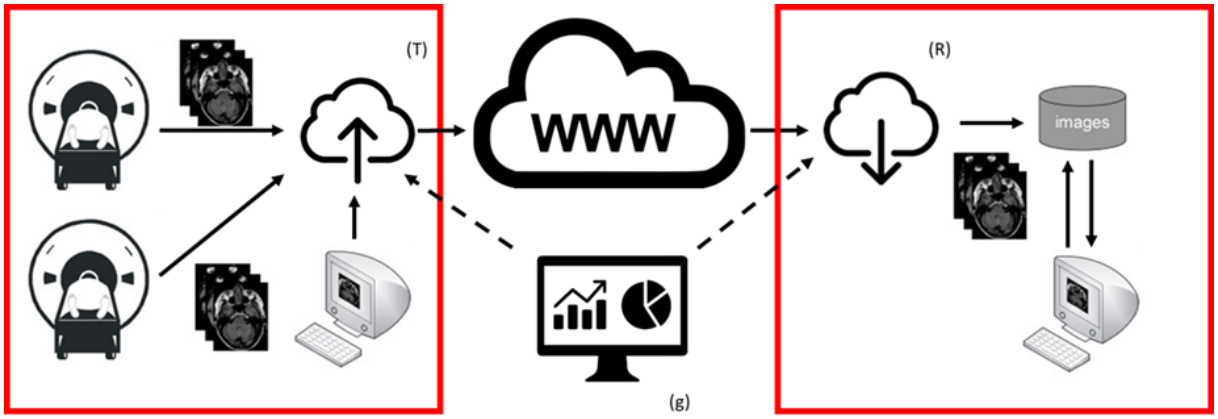

Image 2: describes *Uploader* general architecture, its components and interfaces.

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 primarly written in 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 unfeasible 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 decide 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.

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;
- Potential problems internal to the Erlang/Elixir VM environment causing crashes;
- Code looked excessivly large and complex for the intended use cases.

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 whilest 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. 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?*

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 "thecnical 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

Order	Ttitle	Abstract
1	Ignore, Refactor, or Rewrite [FAIRBANKS, 2019]	Imagine that you have some code written, but it has problems. The problems are small enough that you could imagine rewriting the code completely, and you must choose what do. You could do nothing (ignore it), make incremental changes (refactor it), or write new code from scratch (rewrite it). How do you choose? What factors do you consider? There's already a lot of guidance. In fact, the very existence of refactoring on the list of choices is special because the idea of refactoring code wasn't well formed until the 1990s. When you refactor code, you make changes that improve its structure but do not change its visible behavior, and our tools are increasingly good at supporting refactoring, helping us make sweeping changes safely.
2	A cost analysis of the software dilemma: to maintain or to replace [BARUA and TRIDAS, 1989]	Although software maintenance claims a significant part of the data processing budget, very few authors have examined the tradeoffs between software maintenance and software replacement. We hypothesize that due to frequent modifications and functional enhancements, the system complexity increases rapidly, leading to a sharp increase in maintenance cost. Thus, there may exist a time when it is optimal to rewrite the system completely, which results in a reduction of complexity and subsequent maintenance cost. In this paper, we develop an analytical model for determining the optimal rewriting time. We consider two rewriting strategies involving old and new (superior) technologies. Several interesting propositions with managerial implications emerge out of the analysis. These include the impacts of increasing maintenance requirements and

		unstructuredness of the technology upon the optimal rewriting time, the differences in replacement times for old and new technologies, and the effects of system integration requirements on replacement decisions.
3	A Decision Model for Software Maintenance [KRISHNAN et al., 2004]	In this paper we address the problem of increasing software maintenance costs in a custom software development environment, and develop a stochastic decision model for the maintenance of information systems. Based on this modeling framework, we derive an optimal decision rule for software systems maintenance, and present sensitivity analysis of the optimal policy. We illustrate an application of this model to a large telecommunications switching software system, and present sensitivity analysis of the optimal state for major upgrade derived from our model. Our modeling framework also allows for computing the expected time to perform major upgrade to software systems.
4	Why software fails [software failure] [CHARETTE, 2005]	Most IT experts agree that software failures occur far more often than they should despite the fact that, for the most part, they are predictable and avoidable. It is unfortunate that most organizations don't see preventing failure as an urgent matter, even though that view risks harming the organization and maybe even destroying it. Because software failure has tremendous implications for business and society, it is important to understand why this attitude persists.
5	Why did your project fail? [CERPA and VERNER, 2009]	We have been developing software since the 1960s but still have not learned enough to ensure that our software development projects are successful. Boehm2 suggested that realistic schedule and budgets together with a continuing steam of requirements changes are high risk factors. The Standish Group in 1994 noted that approximately 31% of corporate software development projects were cancelled before completion and 53% were challenged and cost 180% above their original estimate.13 Glass discussed 16 project disasters.5 He found that the failed projects he reviewed were mostly huge and that the failure factors were not just management factors but also included technical factors.
6	An analysis of software project failure [ABE et al., 1979]	The main aim of this paper is to indicate how various losses may be reduced or avoided when the development of software does not proceed according to its schedule; i.e., if what we call “bankruptcy” occurs. Data were collected from twenty three projects in various types of applications, the projects together containing a million lines of code. The causes of failure in developing software were obtained by interviewing the managers of the projects under observation. Having analysed these two aspects, this paper points out under what circumstances managers are likely to fail and proposes a method of detecting failures in the software development.

7	What factors lead to software project failure? [VERNER and CERPA, 2008]	It has been suggested that there is more than one reason for a software development project to fail. However, most of the literature that discusses project failure tends to be rather general, supplying us with lists of risk and failure factors, and focusing on the negative business effects of the failure. Very little research has attempted an in-depth investigation of a number of failed projects to identify exactly what are the factors behind the failure. In this research we analyze data from 70 failed projects. This data provides us with practitionerspsila perspectives on 57 development and management factors for projects they considered were failures. Our results show that all projects we investigated suffered from numerous failure factors. For a single project the number of such factors ranges from 5 to 47. While there does not appear to be any overarching set of failure factors we discovered that all of the projects suffered from poor project management. Most projects additionally suffered from organizational factors outside the project managerpsilas control. We conclude with suggestions for minimizing the four most common failure factors.
8	Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics [CURTIS et al., 1979]	Three software complexity measures (Halstead's E, McCabe's u(G), and the length as measured by number of statements) were compared to programmer performance on two software maintenance tasks. In an experiment on understanding, length and u(G) correlated with the percent of statements correctly recalled. In an experiment on modification, most significant correlations were obtained with metrics computed on modified rather than unmodified code. All three metrics correlated with both the accuracy of the modification and the time to completion. Relationships in both experiments occurred primarily in unstructured rather than structured code, and in code with no comments. The metrics were also most predictive of performance for less experienced programmers. Thus, these metrics appear to assess psychological complexity primarily where programming practices do not provide assistance in understanding the code.
9	The Use of Software Complexity Metrics in Software Maintenance [KAFURA et al., 1987]	This paper reports on a modest study which relates seven different software complexity metrics to the experience of maintenance activities performed on a medium size software system. Three different versions of the system that evolved over a period of three years were analyzed in this study. A major revision of the system, while still in its design phase, was also analyzed.

10	A Controlled Experiment on the Impact of Software Structure on Maintainability [ROMBACH, 1987]	This paper describes a study on the impact of software structure on maintainability aspects such as comprehensibility, locality, modifiability, and reusability in a distributed system environment. The study was part of a project at the University of Kaiserslautern, West Germany, to design and implement LADY, a LAnguage for Distributed systems. The study addressed the impact of software structure from two perspectives. The language designer's perspective was to evaluate the general impact of the set of structural concepts chosen for LADY on the maintainability of software systems implemented in LADY. The language user's perspective was to derive structural criteria (metrics), measurable from LADY systems, that allow the explanation or prediction of the software maintenance behavior. A controlled maintenance experiment was conducted involving twelve medium-size distributed software systems; six of these systems were implemented in LADY, the other six systems in an extended version of sequential Pascal. The benefits of the structural LADY concepts were judged based on a comparison of the average maintenance behavior of the LADY systems and the Pascal systems; the maintenance metrics were derived by analyzing the interdependence between structure and maintenance behavior of each individual LADY system.
11	Some Stability Measures for Software Maintainers [YAU and COLLOFELLO, 1980]	Software maintenance is the dominant factor contributing to the high cost of software. In this paper, the software maintenance process and the important software quality attributes that affect the maintenance effort are discussed. One of the most important quality attributes of software maintainability is the stability of a program, which indicates the resistance to the potential ripple effect that the program would have when it is modified. Measures for estimating the stability of a program and the modules of which the program is composed are presented, and an algorithm for computing these stability measures is given. An algorithm for normalizing these measures is also given. Applications of these measures during the maintenance phase are discussed along with an example. An indirect validation of these stability measures is also given. Future research efforts involving application of these measures during the design phase, program restructuring based on these measures, and the development of an overall maintainability measure are also discussed.

12	An economic model to estimate software rewriting and replacement times [CHAN et al, 1996]	The effort required to service maintenance requests on a software system increases as the software system ages and deteriorates. Thus, it may be economical to replace an aged software system with a freshly written one to contain the escalating cost of maintenance. We develop a normative model of software maintenance and replacement effort that enables us to study the optimal policies for software replacement. Based on both analytical and simulation solutions, we determine the timings of software rewriting and replacement, and hence the schedule of rewriting, as well as the size of the rewriting team as functions of the: user environment, effectiveness of rewriting, technology platform, development quality, software familiarity, and maintenance quality of the existing and the new software systems. Among other things, we show that a volatile user environment often leads to a delayed rewriting and an early replacement (i.e., a compressed development schedule). On the other hand, a greater familiarity with either the existing or the new software system allows for a less-compressed development schedule. In addition, we also show that potential savings from rewriting will be higher if the new software system is developed with a superior technology platform, if programmers' familiarity with the new software system is greater, and if the software system is rewritten with a higher initial quality.
13	ON THE ECONOMICS OF THE-SOFTWARE REPLACEMENT PROBLEM [GODE et al., 1990]	Software maintenance constitutes a significant fraction of the software budget. The cost of maintaining old applications has been escalating and this trend is likely to continue in the foreseeable future. The study of software maintenance strategies has become important to both researchers and practitioners in Information Systems. While there is a rich literature on the technical aspects of software maintenance, research on the economics of maintenance is in its infancy. In particular, the tradeoffs between maintaining and rewriting old software have not been investigated from a theoretical standpoint. In this paper, we present an economic model of the software replacement problem. Based on available empirical evidence, we hypothesize that, with frequent modifications and enhancements, the complexity of software increases rapidly. This deterioration of the code leads to a sharp increase in the maintenance cost. Thus, there may exist a time when it is optimal (in an economic sense) to rewrite the system, which reduces the system complexity and the subsequent maintenance cost. The proposed model allows us to compare the economics of various rewriting strategies and to determine the optimal rewriting point (s). Some interesting results with implications for the systems manager are obtained from the analysis. These include the impacts of system size, structuredness of the underlying technology, and the availability of superior technologies upon the rewriting point (s) and life cycle costs. A numerical example is provided to demonstrate the applicability

		of the model.
14	Legacy systems: Coping with success [BENNETT, 1995]	Legacy systems may be defined informally as “large software systems that we don't know how to cope with but that are vital to our organization”. Legacy software was written years ago using outdated techniques, yet it continues to do useful work. Migrating and updating this baggage from our past has technical and nontechnical challenges, ranging from justifying the expense in dealing with outside contractors to using program understanding and visualization techniques.
15	Software risk management: principles and practices [BOEHM, 1991]	The emerging discipline of software risk management is described. It is defined as an attempt to formalize the risk-oriented correlates of success into a readily applicable set of principles and practices. Its objectives are to identify, address, and eliminate risk items before they become either threats to successful software operation or major sources of software rework. The basic concepts are set forth, and the major steps and techniques involved in software risk management are explained. Suggestions for implementing risk management are provided.
16	The causes of project failure [PINTO et al., 1990]	A study was conducted of 97 projects identified as failures by the projects' managers or parent organizations. Using the project implementation profile, a set of managerially controllable factors is identified as associated with project failure. The factors differed according to three contingency variables: the precise way in which failure was defined; the type of project, and the stage of the project in its life cycle. Implications for project management and for future research on failed projects are discussed. The results demonstrated empirical justification for a multidimensional construct of project failure, encompassing both internal efficiency and external effectiveness aspects. The fact that the critical factors associated with failure depended on the way in which failure is defined suggests that it is necessary to know considerably more about how project managers define failure (and success) and, indeed how the parent organization makes judgments on the matter.
17	Fear of trying: the plight of rookie project managers [PRESSMAN, 1998]	Most software engineers do not want the hassle of project management. Poor project management is the number one cause of software project failure. How do we grow good project managers? What do we teach the rookies who have just been appointed to lead their first software project? Regardless of the training or mentoring approach you use, I suggest focusing on four major attributes, which I describe in their order of importance: communication, negotiation, organization and facilitation.

4.7 REVIEW REPORT

This review report was created after investigating the various 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.

Study	Finding	Description
1	1	When problems appear on a software, and they are small enough, one usually have three options: doing nothing (ignore), making incremental changes (refactor), or writing a new code from scratch (rewrite). However, in some cases, one of the options might not be available.
1	2	Engineers have a bias toward fixing things (refactoring).
1	3	The decision making has a lot of guessings, but it often includes: 1) estimating how long it will take to both fixing and rebuilding; 2) assessing if rewriting can be afforded (cost and time); 3) deciding if the problems can be exaclty addressed; 4) analysing if the requirements are fully known (otherwise rewriting will have to mimic the old software).
1	4	In general, refactoring is less risky and easier to integrate, but rewriting is cheaper and faster.
1	5	It favors rewriting when: 1) the code is entangled; 2) the domain modal is wrong or outdated; 3) major language, framework or architechtural changes are needed.
1	6	If the code is messy and you decide to ignore, it communicates the wrong message to the team which leads to further problems (broken windows theory).
1	7	When the problem is small enough, deciding what to do is easier. However, on a large scale, there's less advice and the decision is always a trade-off between the necessity of fixing the problems against the resources, schedule and commitments to deliver new features. Hence, there's no clear right choice, one can only maximize chances of making a good decision.
2	8	The increase in system size, control flows and inter-module interactions results in higher complexity, which makes system maintenance

		progressively more difficult.
2	9	When the system is rewritten completely, the resulting system is expected to be much more modular with fewer inter-module interactions. The restructuring of the system is therefore expected to lower complexity, and hence its maintenance costs.
2	10	Software complexity (measured as lack of structurednes) increases in time, due to a deterioration from its initial state, which increases maintenance costs.
2	11	The lack of software documentation and personnel turnover make it even more difficult to enhance the system. Thus, it is expected that over time, it takes more and more effort to maintain the system.
2	12	The operational life of a system increases with an increase in modification/enhancement requirements and decreases with increased structuredness of the technology.
2	13	Advanced technologies like 4GLs make programming significantly less time consuming.
2	14	Switching to a superior technology occurs earlier than rewriting with the same technology.
2	15	Very often, organizations continue to maintain old inefficient systems much longer than the optimal time to rebuild them.
2	16	Development and acquisition costs, cost of integration with existing applications and learning/training cost are usually neglected.
3	17	Although a major replacement of the system may require significant investment, such a rework will also improve consistency and increase familiarity of the code to the developers, which lowers further maintenance costs.
3	18	Because software evolves over time as changes are made to it, the complexity of the system and inherent difficulties related to software maintenance tend to increase unless preventive measures are taken to improve the state of the system. For example, preventive measures such as improving and reworking the design, updating documents, and establishing change control to software files may enhance maintainability of the software code.
3	19	In practice, the decision to perform a major rewrite to the software is undertaken when the maintenance has become too expensive, software reliability low, change responsiveness sluggish, system performance not acceptable, or system functionality outdated.
3	20	The software code loses its structure the more times it is modified for maintenance work. The faster the code is changed, the faster it deteriorates. As the software complexity is a function of both the size and the pace of change, it grows exponentially if the rate of change and the additions of new functionalities are high.

3	21	Stochastic models can be derived to predict the optimal moment to replace the code based on the state of the system and the code changes history.
4	22	The most common factors for project failures are: - 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.
4	23	IT project usually fails when the rework exceeds the value-added work that's been budgeted for.
4	24	Organizations are often seduced by the technological imperative: the urge to use the latest technology in hopes of gaining a competitive edge. But using immature or untested technology is a sure route to failure.
4	25	A project's sheer size is a fountainhead of failure. Large-scale projects fail three to five times more often than small ones. The larger the project, the more complexity there is in both its static elements (the discrete pieces of software, hardware, and so on) and its dynamic elements (the couplings and interactions among hardware, software, users and connections to other systems).
4	26	Poor project management is probably the single greatest cause of software failures today.
5	27	Several management related factors were found in software project failures like: - 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.
6	28	Most software bankruptcies become apparent during the test period.
6	29	In the studied cases, the main factor leading to bankruptcy was `misjudgment`, splited between inadequate case study, insufficient fact finding and judgment based on over-optimistic reports; while inexperience of the team members and interferences from other projects did not seem to correlate
7	30	Software projects usually fail because of a combination of technical, project management and business decisions factors. High staff turnover is also associated.
7	31	Other software project failure factors have been described: - 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.
7	32	Projects may fail because: - rework costs exceeded the value-added work; - inadequate planning and specifications led to numerous change requests; - cost and schedule overruns. These problems all probably stem from inadequate requirements. Rework is usually caused by requirements changes, so excessive rework costs are likely due to problems with the initial requirements. Inadequate planning and specification leading to numerous change requests, is also probably caused by inadequate requirements at the start of the project.

7	33	Common causes of failure are: - underestimated project scope; - poor understanding of scope; - unclear requirements; - politics; - management style; - complexity; - conflict with team members and outside managers; - noisy environment; - poor tool support; - plans developed by managers with no software development experience, and no PM involvement; - lack of senior management support; - unrealistic schedule and budget; - lack of resources; - low project priority.
7	34	Overall the most frequent factors for failure were: - the delivery date impacted the development process (93% of the failed projects); - the project was underestimated (81%); - risks were not re-assessed, controlled, or managed through the project (76%); - staff were not rewarded for working long hours (73%); - delivery decision was made without adequate requirements information (73%); - staff had an unpleasant experience working on the project (73%).
8	35	There are countless proposed metrics to asses the complexity of a program, such as: - number of distinct, or total frequencies, of operators and operands; - total programming times; - length of the decision tree of a program; - number of linearly independent control paths comprising a program.
8	36	The number of statements in the program proved to be strongly related to performance on the experimental tasks (understanding and modifying code).
9	37	The repairs caused increases in both the code metrics and the structure metrics. The combined effects of numerous changes is not localized. In particular, the growth in structural is consistent with other studies and the general perception that systems become more difficult to maintain over time because they become increasingly complex.
10	38	The average effort (in staff-hours) per maintenance task and module is best explained or predicted by internal complexity metrics either length or structure and less influenced by implicit information flows. Internal complexity aspects are more dominant than external complexity aspects.
11	39	The stability measures indicate that the stability of programs utilizing parameter passing is generally better than that of programs utilizing Global

		Variables.
11	40	The stability measures indicate that the stability of programs that use data abstractions is generally better than that of programs which do not.
11	41	Program complexity directly affects the understandability of the program and, consequently, its maintainability. Thus, the stability of programs with less complexity is generally better than that of programs with more complexity.
12	42	Avoid complete rewrite when the application concerned is large. It may not be economical to rewrite a large application because much of the effort will be expended on redeveloping the initial software functionality.
13	43	There is substantial support in favor of the argument that structured code is easier to maintain.
14	44	Top 10 risk items found in software projects along with risk managements techniques: - Personnel shortfalls: staffing with top talent, job matching, team building, key personnel agreements, cross training. - Unrealistic schedules and budgets: detailed multisource cost and schedule estimation, design to cost, incremental development, software reuse, requirements scrubbing. - Developing the wrong functions and properties: organization analysis, mission analysis, operations-concept formulation, user surveys and user participation, prototyping, early users' manuals, off-nominal performance analysis, quality-factor analysis. - Developing the wrong user interface: prototyping, scenarios, task analysis, user participation. - Gold-plating: requirements scrubbing, prototyping, cost-benefit analysis, designing to cost. - Continuing stream of requirements changes: high change threshold, information hiding, incremental development (deferring changes to later increments). - Shortfalls in externally furnished components: benchmarking, inspections, reference checking, compatibility analysis. - Shortfalls in extemally performed tasks: reference checking, preaward audits, award-fee contracts, competitive design or prototyping, team-building. - Real-time performance shortfalls: simulation, benchmarking, modeling, prototyping, instrumentation, tuning. - Straining computer-science capabilities: technical analysis, cost-benefit analysis, prototyping, reference checking.
15	45	Identifying and dealing with risks early in development lessens long-tem costs and helps preven disasters.

16	46	Critical factors for project success: - 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.

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]:

- Maintainability: the effort in staff-hours per maintenance task;
- Comprehensibility: 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;
- Locality: 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[1] 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;

[1] 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 eficiente; 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).