МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Харківський національний університет імені В.Н.Каразіна
Факультет математики і інформатики
Кафедра теоретичної та прикладної інформатики

Кваліфікаційна робота

магістр

на тему «Модернізація та убезпечення застарілого коду C++: аналіз і вибір інструментів»

Виконав: студент 2 курсу, групи МФ-61
спеціальність 122 «Комп'ютерні науки»
освітньо-наукова програма
«Інформатика»

Радченко О.Д.

Керівник: Wathne E., Керівник інженерії у Laerdal Medical AS

Рецензент: Jakobsen H.N., Старший розробник у Laerdal Medical AS

Консультант: Зарецька І.Т., доцент, кандидат фізико-математичних наук

Харків – 2024 року

MINISTRY OF EDUCATION AND SCIENCE OF UKRAINE
V.N. Karazin Kharkiv National University
Faculty of Mathematics and Computer Science
Department of Theoretical and Applied Informatics

Qualification Work

Master's Thesis

In the subject of «Modernizing and Securing Legacy C++ Code: An Analysis and Tool Selection»

Completed by: a second-year student, group MF-61 Specialization 122 «Computer Science» Educational and Scientific Program «Informatics»

Radchenko O.D.

Supervisor: Wathne E., Engineering manager at Laerdal Medical AS

Reviewer: Jakobsen H.N., Senior developer at Laerdal Medical AS

Consultant: Zaretska I.T., Associate Professor, PhD

Kharkiv – year 2024

1. INTRODUCTION

1.1. Formulation of the purpose of the work, tasks, and justification of the relevance of the topic

In today's rapidly evolving technological landscape, software development forms an integral part of everyday life, with a considerable number of developers persistently working on code to meet changing demands and innovations. Unfortunately, a substantial amount of this code, particularly in the C++ language, can be classified as legacy. This legacy code often lacks the efficiency, readability, and maintainability of modern code, leading to significant challenges and frustrations for developers tasked with its upkeep and improvement. The issues associated with legacy C++ code pose a severe problem that demands effective solutions. People should strive to modernize this legacy code to not only solve the issues faced by developers but also enhance the overall performance and functionality of the software systems it underpins.

The purpose of this research paper is to delve into the details of modernizing legacy C++ code, a process that has become increasingly crucial today. The main reason for its importance is the rapid development of technologies in the past 20 years and the introduction of generative artificial intelligence (AI). The paper aims to address the substantial security and maintainability risks associated with legacy code, the increased time and resources required to modify such code, and the frustration often experienced by developers when dealing with outdated and obsolete programming structures.

To achieve it, the following tasks were set up:

- Define the parameters within what code can be considered «legacy», a term that often carries varying annotations depending on the context.

- Define the concepts of «modernizing» and «securing» legacy code, outlining the processes and techniques that can be employed to update and safeguard these vulnerable codebases.
- Establish certain criteria that can be used as markers of success to track the progress.
- Suggest a methodology for setting priorities during the modernization process.
- Evaluate the costs associated with modernizing and securing the code and compare it with the costs of ignoring it.
- Explore some best practices and tools that can semi-automate the process, making modernization less daunting and more efficient.

The research begins with defining basic terms, as many developers lack definitions. Not knowing these definitions is a huge problem because it becomes difficult to identify legacy code, and without this knowledge, the process might derail.

The relevance of this topic cannot be overstated. C++, although a robust and powerful programming language, has existed for a lengthy period. This implies that a considerable proportion of C++ code in operation today can be classified as «legacy». In fact, it is increasingly challenging to find a working position related to C++ language that does not involve interacting with and updating legacy code in some form. Therefore, understanding the nuances of modernizing legacy C++ code is not just beneficial but imperative for developers and organizations wishing to keep their software relevant, secure, and efficient.

The paper is written as part of the research performed for Laerdal Medical AS organization. A leading company in the field of healthcare education and resuscitation training. [1] Laerdal's software system is responsible for handling

different medical training manikins and devices used to simulate the human body in medical training scenarios. Over the last 20 years, the codebase that controls these manikins has grown, which unfortunately led to it becoming legacy.

1.2. A brief overview of known results in the research area

One of the known results in this area is the successful adaptation of automated refactoring tools, which can significantly reduce the manual effort involved in code modernization. These tools can be used to automatically identify and refactor areas of code that could benefit from modern C++ features, thereby improving code readability and maintainability.

Another significant result is the increased use of unit testing in legacy C++ code modernization. The application of unit tests ensures that the modernized code maintains its original functionality, reducing the risk of introducing new bugs.

Code review and static analysis tools also play a significant role in modernizing legacy C++ code. These tools can identify potential problems, such as memory leaks and null pointer dereferences, which can then be addressed during the modernization process.

Further, the application of design patterns has also been identified as a key technique for improving the structure and quality of legacy C++ code. By implementing well-established design patterns, developers can make the code more modular, reusable, and easier to understand and maintain.

Whenever a new C++ standard such as C++11, C++17, or C++23 is released, it has been proven to be highly beneficial in modernizing legacy code. These newer standards provide many features that can simplify code and improve performance, making it possible to transform outdated, complex legacy code into cleaner, more efficient modern code.

Despite these advances, the modernization of legacy C++ code remains a challenging task that requires careful planning and execution. The process should be guided by a clear understanding of the legacy code and the goals for its modernization, as well as a thorough knowledge of modern C++ features and best practices.

During the research of known results, only a few publicly available articles were found. Most of the information is scattered around the internet on numerous web pages, and it is difficult to identify useful information from faulty suggestions.

1.3. Information about the obtained results and their novelty

The obtained results in modernizing legacy C++ code have shed light on new methodologies and tools that have significantly improved the efficiency of the modernization process. The integration of modern C++ standards and automated refactoring tools has made it possible to enhance the performance, readability, and maintainability of legacy code while reducing the amount of manual effort required. Innovative strategies and tools have been improving for the last few years. These tools will enhance the quality, performance, and maintainability when working with legacy code.

The following results were obtained:

- The concept of legacy code and what it means to modernize it were defined. Many developers, while understanding the concept, cannot define it.
- Criteria to track the improvements for the code were defined.
- A clear procedure on how to find what to refactor and set the priorities was explained.
- A method to select a proper tool was developed.

The novelty of the paper lies in its comprehensive exploration and presentation of advanced techniques and tools for modernizing legacy C++ code. Most other research focuses only on one of the possible definitions of the «legacy code», which vastly limits both practical solutions and provided guidelines. The grouping of the information was done in addition to developing a clear path for getting from legacy code to a clear and manageable program, thus contributing significantly to the existing body of knowledge in this area.

2. MAIN PART

2.1. Formulation of the problem

2.1.1. What is legacy C++ code?

As a cornerstone of numerous systems and software architectures of yesteryears, legacy C++ code remains a crucial area of study for its practical implications.

Going further into the details of legacy C++ code, it is essential to note that there exist several definitions for «legacy code», each filled with its unique implications and interpretations. This uncertainty in definitions is not exclusive to C++ but persists throughout every programming language.

One of the common definitions describes legacy code as any system or application code that was written in an outdated or obsolete version of a programming language, such as early versions of C++ (especially before the C++11 standard). This includes:

1. The code that was permitted in an earlier version of the C++ standard but has been deprecated or removed in a later version.
2. The code that was formerly allowed by a specific compiler but never conformed to the C++ standard.
3. The code that conforms to all versions of the standard but is no longer considered best practice in modern C++. [2]

Another popular definition views legacy code as any code that lacks adequate documentation or code and test coverage. This perspective underscores the operational challenges in maintaining and refactoring such code, highlighting the significance of proper documentation and robust testing mechanisms in software development. [3, pp. 9-11]

A third perspective defines legacy code as the code inherited from another developer or team. Here, the emphasis is on the transition of code ownership and the associated challenges in understanding and modifying someone else's code. [4]

These definitions encapsulate the diverse perspectives on legacy code, each contributing to a broader, more nuanced understanding of this complex concept in the realm of computer science.

The first perspective, which identifies legacy code as any system or application code written in an outdated or obsolete version of C++, underscores the temporal evolution of the programming language. It allows for a historical examination of the language, tracing its progression, transformations, and enduring influence.

At the same time, the second perspective, which regards legacy code as any code that lacks sufficient documentation or code and test coverage, will show the details and challenges associated with maintaining and refactoring such code. This perspective emphasizes the essential roles of comprehensive documentation and robust testing mechanisms in effective software development.

Simultaneously, the third one, regarding code inherited from the other team, combines both other ones topped by additional challenges of knowledge absence about the codebase and its history. It is noteworthy to underscore that it can be significantly mitigated if the first two perspectives are diligently addressed on all the legacy code bases. If every development team commits to maintaining their code in accordance with the most recent versions of C++, thereby preventing it from becoming outdated and ensuring comprehensive documentation and robust code and test coverage for their code, the challenges associated with inheriting such code are likely to be reduced. The inheriting teams would be receiving code that is up-to-date and well-documented (see 2.1.4) and thoroughly tested (see 2.1.4), thereby

minimizing the difficulties in understanding, maintaining, and refactoring the inherited code.

2.1.2. What is considered by modernizing a program?

This paper discusses the six crucial variables that motivate the need for application modernization. According to Stefan Van Der Zijden [5]: «For many organizations, legacy systems are seen as holding back the business initiatives and business processes that rely on them». [6] Application modernization refers to the process of updating and upgrading applications to ensure their functionality, efficiency, and relevance in today's business environment. The legacy application, due to its outdated technology, architecture, or functionality, presents significant concerns and hindrances in complying with changes that are driven by those six drives.

The first three drivers:

- Business fit.
- Business value.
- Agility.

They stem from a business viewpoint. The term «business fit» shows how well the application aligns with the evolving demands of the business landscape. If the legacy application fails to meet these new specifications, it requires modernization to ensure optimal compatibility. Similarly, «business value» refers to the benefits the application provides to the business. If an application fails to deliver enhanced business value, it warrants modernization. The third drive, «agility», denotes the application's capacity to adapt and respond swiftly to the changing demands of the business. Applications that lack this flexibility may become a financial burden or

pose potential risks, which might include any losses to the project’s financial state, client base, and trust due to a lack of innovation.

The remaining three drivers:

- Cost.
- Complexity.
- Risk.

They originate from the IT perspective. The application’s «cost» is the amount of effort and money required to support it. «Complexity» If the application's total «cost» of ownership exceeds its value or if the technology is overly «complex» in its structure, functionality, or both, it indicates the need for modernization. Additionally, if the application's security, compliance, support, or scalability is at any risk, modernization becomes paramount. [6]

2.1.3. What is considered by securing C++ code?

Among the influx of numerous programming languages that have emerged over the past several years, C++ continues to stand its ground as one of the most potent and prevalent programming languages preferred by developers across the globe. This language's reputation for efficiency and performance is well-earned, as it equips developers with the necessary tools to develop dependable, high-performing applications that meet a broad spectrum of user needs.

The power of the C++ language is also one of its weaknesses. It is easy to introduce severe security and stability issues. A fitting example of this is multiple inheritance, which is not considered a good practice but is an available feature of the language.

It is of high importance for developers to prioritize secure programming during the development process. Sticking to secure programming practices is not merely a

matter of following a protocol but is a commitment to maintain the integrity of the applications being developed. Whether the task involves developing modest utility applications or intricate, complex systems, the security of the code should never be compromised. This is crucial to protect user data and to prevent issues such as unauthorized access, which could lead to a breach in data security and confidentiality.

Being a proficient C++ developer is not only about mastering the language, but it also requires an understanding of the potential security threats that could undermine the code, which requires the knowledge of how parts ranging from memory management to compile process work «under the hood». Such awareness is an invaluable asset as it enables developers to tackle these threats proactively. With a comprehensive understanding of potential threats, developers can address these issues effectively, thereby cleaning their code and enhancing its efficiency. The spectrum of these threats is vast, but it is important to familiarize oneself with some of the most common threats, which include: [7]

- **Buffer Overflow**: This security issue is a prevalent concern associated with C++ applications. A buffer overflow transpires when a program attempts to write data that exceeds the designated boundaries of a buffer, resulting in corruption of the adjacent memory. The exploitation of this vulnerability can lead to several unwanted outcomes, such as the execution of arbitrary code, overwriting of vital data, or even causing the application to crash. Employing adequate safeguards against this threat, such as bounds checking and carefully managing memory, is crucial to maintaining the security of the application.
- **Integer Overflow and Underflow**: These issues typically occur when the value that a program intends to store in an integer either surpasses

the maximum value that the integer can represent (overflow) or falls below its minimum representable value (underflow). The consequences of such situations can lead to unpredictable memory behavior, potentially leading to corruption and security breaches. It underscores the importance of rigorous checks to prevent values that are too large or too small from causing system instability.

- **Injection Attacks**: This form of attack involves the introduction of malicious code into a program, potentially culminating in unintended execution. Within the context of C++, common types of injection include code injection and command injection. Regular input validation and sanitization are key practices to prevent such attacks, thereby preserving the integrity of the application.
- **Pointer Initialization**: If a pointer that has not been correctly initialized is used, it could potentially expose a considerable amount of sensitive data. Moreover, the use of this uninitialized pointer could cause the program to read from or write to an unexpected memory location, leading to potential security breaches. Included in this category are issues such as use-after-free, double-free, and uninitialized memory access, any of which could lead to the exploitation of the program. It underscores the importance of diligent pointer management in application security.
- **Incorrect Type Conversion**: Also recognized as type punning, this issue arises when a program erroneously treats a variable of one data type as if it were a different data type. This can result in significant data loss and can potentially cause the program to behave unpredictably or erratically. Ensuring accurate type conversions through rigorous type-checking measures can help mitigate this risk.

- **Undefined behavior**: Undefined behavior in C++ code, where the standard does not specify the outcome, leads to unpredictable and potentially harmful results, making it crucial to avoid. It can compromise program correctness, create security vulnerabilities, impede debugging, and reduce portability. Exploitation by attackers can lead to arbitrary code execution, data overwriting, or program crashes. Its unpredictable nature can make error reproduction and diagnosis difficult, thereby slowing development and increasing the risk of new bugs. Furthermore, undefined behavior can differ between compiler implementations or hardware platforms, affecting program functionality in different environments.

2.1.4. What are criteria for improvements?

Establishing criteria for improvements is important when it comes to modernizing and securing any code. This process often involves significant refactoring and rewriting, and without a clear set of improvement criteria, it can become directionless and inefficient. The criteria provide a roadmap for the modernization process, identifying key areas of focus and enabling the tracking of progress toward the modernization goals.

From the perspective of security, these criteria are even more vital. Legacy C++ code often contains vulnerabilities that have been discovered and exploited over the years. A fitting example of such exploitation is an application called «Cheat Engine», which can scan and modify the memory of the program. The improvement criteria serve as a guide, helping to ensure that these security issues are addressed during the modernization process. This not only helps to secure the legacy code but also contributes to its robustness and reliability.

The criteria for improvements form the backbone of any effort to modernize and secure legacy code. They provide direction, facilitate tracking and evaluation, and ensure that the modernized code is secure, robust, and dependable. Hence, their importance cannot be overstated.

The first perspective to consider is the increase in code and test coverage percentages. Before diving deeper, the boundary between code coverage and test coverage should be established.

Test coverage represents the amount of testing performed. It is not based on the code itself but focuses on the numerous documents related to requirements specifications. This includes functional requirements specification (FRS), user requirements specifications (URS), and other project-specific documents that contain requirements. With all the requirements, there is a list of functionalities that should be tested, and test coverage indicates how many are covered. On the other hand, code coverage is the number of validated lines of code when tests are run. Both are crucial metrics in assessing the quality of the software. [8]

For legacy C++ code, it is imperative to ensure that code and test coverage percentages continue to grow, as it provides a quantitative measure of how much of the codebase is tested. It is worth mentioning that tests range all the way from unit and integrational tests to «end-to-end» and system tests. This includes both automated and manual tests, as both can verify a certain number of lines of code. It enables developers to identify areas of the code that have not been adequately tested, thereby allowing them to focus their testing efforts on these areas and reducing the likelihood of undetected bugs or issues. A steady increase in test coverage percentage is indicative of an improvement in the quality of the C++ code.

In theory, it is convincible to suggest that every team should have 100% coverage, but research and experience have shown that the cost of having 100%

coverage is higher than the benefits. The reason is that the last 20-30% are way too difficult and expensive to achieve and maintain. It is worth underlining that it heavily depends on the project type, and a lot of factors arise from the specifics of the subject area. In practice, the team usually settles with a number between 70% and 85%.

Documentation is another crucial factor in the improvement of legacy C++ code. The documentation of a codebase is a significant resource for understanding the functionality of the code and the rationale behind its design and implementation decisions. In the context of C++ code, documentation can be assessed using automated tools to determine the percentage of scopes that it covers. The scope here is defined as a function, a class, or a file. [9] That means that one file with one class and five functions will have seven scopes. An increase in this percentage signifies an improvement in the code, as it ensures that the knowledge about the code is preserved, thereby aiding in its maintenance and future development.

In cases where clear percentage criteria are not readily available, it becomes necessary to conduct initial research (as defined in 2.3) to identify potential issues in the legacy C++ code. These issues could range from undefined behavior and buffer overflow to outdated usage of the C++ standard. By cataloging these issues, developers can gain a better understanding of the code's current state and identify areas for improvement. Arbitrary numbers can be assigned to these issues to provide a quantitative measure of their severity, which can then be used as a baseline for improvement efforts. It is crucial to recognize that there is no «silver bullet» solution in this realm. Each case necessitates a unique approach tailored to its specific characteristics and challenges.

In conclusion, the criteria for improvement in the context of legacy C++ code are multi-dimensional and encompass test coverage, documentation, and the identification of potential issues through initial research. These criteria provide a

comprehensive framework for assessing and improving the quality of legacy C++ code, thereby ensuring its long-term viability and maintainability.

2.2. A developed overview of the current situation in the research field

The prevailing situation in the research field paints a vivid picture of the challenges inherent in legacy C++ projects. The persistence of legacy code can be attributed to a variety of reasons, each of which contributes to the creation and continuation of such code.

Many of the projects were initiated several years ago. Consequently, a considerable proportion of the codebase does not conform to the more recent C++20/23 standards. This misalignment with modern standards often translates into inefficiencies and potential vulnerabilities in the code.

However, one of the primary reasons is management team decisions. In the fast-paced world of software development, there are often tight deadlines to meet. In the interest of delivering products on time, management may not allocate sufficient time for developers to craft optimal solutions, deal with technical debt, or write comprehensive tests. This is the case as these tasks offer no immediate gains, but their absence can lead to the creation of legacy code. According to Professor Peter Welch [10] «good code» is the code that «has never had to live in the wild or answer to a sales team».

The availability of resources also plays a significant role. Part of it is that tools that could improve programming efficiency and code quality are sometimes available only in paid versions, which may be beyond the budget of the project or organization. As a result, developers write code in a manner that may introduce technical debt, thereby leading to the creation of legacy code.

Team culture is another contributing factor. In some development teams, there is often a pervasive culture of ignoring compiler warnings, not writing tests, or providing proper documentation. This could be due to a lack of awareness about their importance or a perception that they are not necessary. Once part of the code base is written in this manner, the technical debt may accumulate and be ignored, further perpetuating the existence of legacy code. This tendency not only hampers identifying potential code issues but also undermines the software's overall quality. These elements are fundamental to the clarity, maintainability, and robustness of the code, yet they are frequently neglected in the development process.

Additionally, the complexity and size of the codebase can lead to legacy code. In large and complex projects, understanding and updating the entire codebase can be a daunting task. This is particularly true when there is a lack of proper documentation. As a result, developers may choose to work around the existing code instead of updating it, leading to more legacy code in the already existing code base. Most of the time, the legacy code will not be dealt with, but rather, it will accumulate.

Finally, legacy code can exist due to the skill level and experience of the development team. Inexperienced developers or those not familiar with best practices might write code that works but is not up to current standards. Over time, this code becomes legacy code.

The current landscape of legacy C++ projects is fraught with challenges, including outdated code standards, ignored compiler warnings, and neglected testing and documentation. There are a lot of factors that lead to the increasing amount of legacy code. These issues underscore the pressing need for concerted modernization efforts in the field, and understanding them is the first step in addressing and mitigating their impact.

2.3. Research methods

The research methods for securing and modernizing legacy C++ code involve a systematic and multifaceted approach that utilizes various techniques, such as:

- **Code Analysis:** This step in the process involves performing a thorough analysis of the existing code. This stage may involve using static analysis tools like «Klocwork» or «SonarQube» that examine the code without executing it to identify potential vulnerabilities, bugs, or areas that need to be improved.
- **Reverse Engineering:** This process is used to understand the system's architecture, design, and functionality. It involves deconstructing the code to its basic components to understand its structure and behavior and might require the use of disassembler tools.
- **Automated Refactoring:** The use of automated refactoring tools can help modernize the codebase, making it more compliant with current coding standards and guidelines. This process can involve the systematic modification of the code by using automated tools (e.g., «ReSharper») to improve its structure and readability while preserving its original functionality.
- **Code Documentation:** Proper documentation is essential when dealing with legacy systems. It helps to understand the original intent of the code, which is crucial during modernization.
- **Testing:** Rigorous testing is performed to ensure that changes made during the modernization process do not break the existing functionality of the code. Unit tests, integration tests, and system tests are all utilized to validate the code and ensure its robustness.

- **Security Audits:** Security audits involve the detailed inspection of the code to identify any potential security flaws or vulnerabilities. These might include buffer overflows, memory leaks, or other potential points of exploitation.
- **Training and Education:** This involves educating the development team about modern C++ practices, secure coding principles, and the specific challenges associated with modernizing legacy systems.
- **Continuous Integration and Continuous Delivery (CI/CD):** This practice allows for frequent code changes to be tested and merged, ensuring that the modernized code is always in a deployable state.
- **Literature Review:** This method is crucial in identifying, evaluating, and interpreting all relevant research findings on the topic. The literature review involves a comprehensive search for research papers, articles, and books on legacy code analysis, refactoring, modernization techniques, and best practices.
- **Case Studies:** This method involves detailed, in-depth analysis of specific instances of successful legacy code transformations. Case studies offer rich contextual data and practical insights into the real-world challenges and solutions associated with code modernization.
- **Analyzing behavioral data:** This method specifically focuses on analyzing Version Control Systems like «Git». It enables the identification and prioritization of code files that are frequently altered, which serves as a strategic measure to streamline the refactoring process. The rationale behind this approach is that files subjected to frequent modifications often tend to have additional bugs and issues in the code.

By leveraging these methods, researchers can effectively secure and modernize legacy C++ code, ensuring its relevance and efficiency in the modern landscape.

It is also worth noting the remarkable technological advancements in Artificial Intelligence (AI) in recent years. Particularly, AI models have proven to be effective in comprehending the complex nature of the code, identifying problematic patterns, and suggesting appropriate solutions. The integration of these AI systems can further enhance the efficiency of maintaining and modernizing an existing C++ codebase. This represents a significant addition to the arsenal of tools available for tackling issues within legacy code, demonstrating the far-reaching potential of artificial intelligence applications in software development.

2.4. Description and justification of algorithms and research results

2.4.1. Changing software

Before diving deep into the solutions for the problem, there must be a solid explanation of the motivation to change the code. The process of modifying software is typically driven by four principal reasons, and each serves a distinct purpose in the software development lifecycle.

- **Adding a feature**: As software evolves to meet new business requirements or user needs, new features are often introduced. This could range from implementing a new functionality to enhance the application's capabilities to integrating with other systems for increased compatibility. The addition of features is a critical aspect of software development that drives its evolution and helps it stay relevant and useful.
- **Fixing a bug**: Software, regardless of how meticulously it has been designed and implemented, can contain bugs. These are flaws or errors

in the code that cause the software to behave unexpectedly or incorrectly. Fixing bugs is a central part of software maintenance and is key to ensuring the software's reliability and performance. It also enhances user experience by eliminating errors that may hinder the software's functionality.

- **Improving the code structure (Refactoring)**: Over time, software design may need to be improved to accommodate changes in technology, business requirements, or coding standards. This could involve refactoring the code to make it more readable and maintainable or restructuring the software architecture to make it more robust and scalable. Improving the design not only enhances the software's quality and maintainability but also makes it easier to add new features or fix bugs.
- **Optimizing resource usage**: Software often needs to be optimized to make better use of resources such as memory, CPU, and disk space. This could involve improving the algorithms used, reducing memory leaks, or minimizing disk usage. Optimizing resource usage is crucial for enhancing the software's performance and scalability and for ensuring that it runs efficiently on the target hardware.

These four reasons - adding a feature, fixing a bug, improving the design, and optimizing resource usage - form the cornerstone of software change. They drive the evolution of software, ensuring it remains dependable, efficient, relevant, and valuable to its users.

In the context of a completed system, the possibilities for alteration are confined to three primary categories: structure, functionality, and resource usage. [3, pp. 3-7]

The structure of a system, which refers to how its components are organized and interrelated, can be modified to enhance maintainability and scalability or accommodate new requirements. Such changes might involve refactoring code for better readability, reorganizing modules for improved coherence, or altering the system architecture for increased robustness and flexibility.

Functionality denotes the capabilities of a system, the tasks it can perform, or the services it provides. Changes to functionality are typically driven by the need to introduce new features or improve existing ones. This might involve adding new capabilities to meet user needs or business requirements, modifying existing features to work more effectively, or removing outdated or redundant features.

Resource usage pertains to the system's consumption of computational resources, such as memory, CPU cycles, or disk space. Changes in this area are aimed at optimization, making the system more efficient. This could involve enhancing algorithms to reduce computational complexity, fixing memory leaks to minimize memory usage, or streamlining data storage to conserve disk space.

	Adding a feature	Fixing a bug	Refactoring	Optimizing
Structure	*Changes*	*Changes*	*Changes*	–
New Functionality	*Changes*	–	–	–
Old Functionality	–	*Changes*	–	–
Resources	–	–	–	*Changes*

The table above shows what the developer should concentrate on when making a change. For example, when the reason for the change is refactoring, then the structure and only the structure of the code should be changed, hence, it is the

developers’ focus. The table also shows what should be preserved. As only the structure should change during refactoring, the functionality and the resource usage should not be affected. The main challenge is that it is not easy to figure out if something that should be preserved was changed. If it were easy, the team could concentrate on the selected behavior and ignore the rest.

2.4.2. Prioritizing components

Exploring more, it is important to set priorities when changing components. Admittedly, an instantaneous and all-encompassing transformation of the entire system is an impracticable aim. Prioritization provides a systematic and efficient approach to code modification and an effective allocation of resources and effort. In the current context, the «hotspot» is a file or a component that is of high priority based on the selected prioritization strategy.

One strategy to prioritize components is the utilization of behavioral data extracted from the existing repository. It provides information on how the work with the code was done. This approach, primarily data-driven, serves as a mirror reflecting the recent history of code modifications. By discerning what components have been frequently altered in the preceding period, the likelihood of those components requiring further modification can be deduced. The frequency of past changes can be an accurate indicator of the code component's instability, thereby aiding in the identification of potential areas requiring attention.

However, the complexity of the component also plays a pivotal role in determining its priority for change. A complex component, due to its intricate structure and the potential for high impact on system operation, often necessitates modification. An array of tools exists designed to quantify «code complexity» through various metrics, including cyclomatic complexity, depth of inheritance, and

coupling. [11] An additional metric to accommodate for the inherited code can also be a relative number of changes made by the person outside of the current development team. If it is more than 50% of all changes done to the component, then it might indicate that it is inherited, which is one of the legacy code definitions.

The so-called «Maintainability index» metric can also be calculated based on all the other metrics to produce an arbitrary score that will unify other metrics with certain coefficients. However, these metrics often fall short of providing a comprehensive and precise measure of complexity that aligns with the intuitive understanding of the term. Consequently, it can be wise to resort to a simpler criterion, such as the number of lines in a file. This metric might not correlate with functionalities as more skilled developers might use fewer lines to deliver the same functionality. Despite this, the number of lines in the code is effective in estimating the effort put into the component, file, or program and the effort required to maintain it. [12] The amount of effort required to maintain the component is one of the key factors in setting the priority for each component (the other one is the cost of modernizing, analyzed in 2.4.3). In addition, having those metrics allows for easy automation that verifies that after a refactoring, the «code complexity», as it was defined, has not increased.

This leads to a case where the «Maintainability index», which will define the priority, can be calculated as a combination of four factors with different influences. These factors are:

1. Number of commits relative to commits for the whole repository.
2. Code age, i.e., the amount of time passed since the last modification.
3. Number of lines relative to the largest file in the whole repository.

4. Code ownership, i.e., if the code is owned by the team or not. The code is not owned by the team in case more than 50% of all changes are made by people outside the development team.

Consequently, the prioritization of components for change is a delicate balance between the frequency of past modifications and the inherent complexity of the component. By employing these two parameters, a systematic and efficient approach to managing legacy C++ code can be achieved. This approach, while respecting the historical development of the code, also allows for its continuous improvement and adaptation to meet evolving requirements and standards. [13]

When prioritizing components based on behavioral data, an open-source tool named «GitCommitsAnalysis» [14] proves to be instrumental. This tool facilitates the extraction of information from a Git repository and presents it in a user-friendly HTML format. Moreover, it also provides the data in Excel or JSON formats, thereby enabling automatic analysis and further manipulation of the data, hence streamlining the process of prioritizing code components for refactoring or rewriting. While the provided data is useful, this tool has problems showing the information vital for the research. Fortunately, as it is an open-source software, it is possible to modify it to include all the additions mentioned above.

In terms of changes made to the utility, there are the following:

1. A Calculation of the maintainability index using the criteria described above is done for each file. It is also presented in a graph view.
2. An ability to provide an input file containing the list of all users that are not part of the current development team but may be part of the Git repository and have any commits there. This data is used for the maintainability index. It is also used to establish and display if the major part of the changes to the file were made by people outside of the team.

3. An ability to provide an input file containing the list of synonyms for any name, as the user can change the username (and it usually happens accidentally) to not lose the grouping of changes made by a specific person.
4. On the view where each folder is displayed, in addition to the absolute number of commits to the folder, there is a percentage number that is independent of the overall commits count.

To genuinely appreciate the «GitCommitsAnalysis» utility, let us delve into a practical example of its operation. This will offer a concrete demonstration of how the tool processes and presents the data, thereby aiding in the prioritization of components within a legacy C++ codebase. The data shown below is based on a repository at Laerdal Medical AS Organization. The repository has existed for more than 20 years and now consists of more than 270000 lines. The repository history spans more than 20500 commits.

Some of the data provided by «GitCommitsAnalysis» show that the last comparatively large update was done to the repository around three years ago. (see Figure 2.4.2.1) The figure shows the number of changes in the repository, where each column represents a month (new to the left and old to the right), and its height is the number of changes.

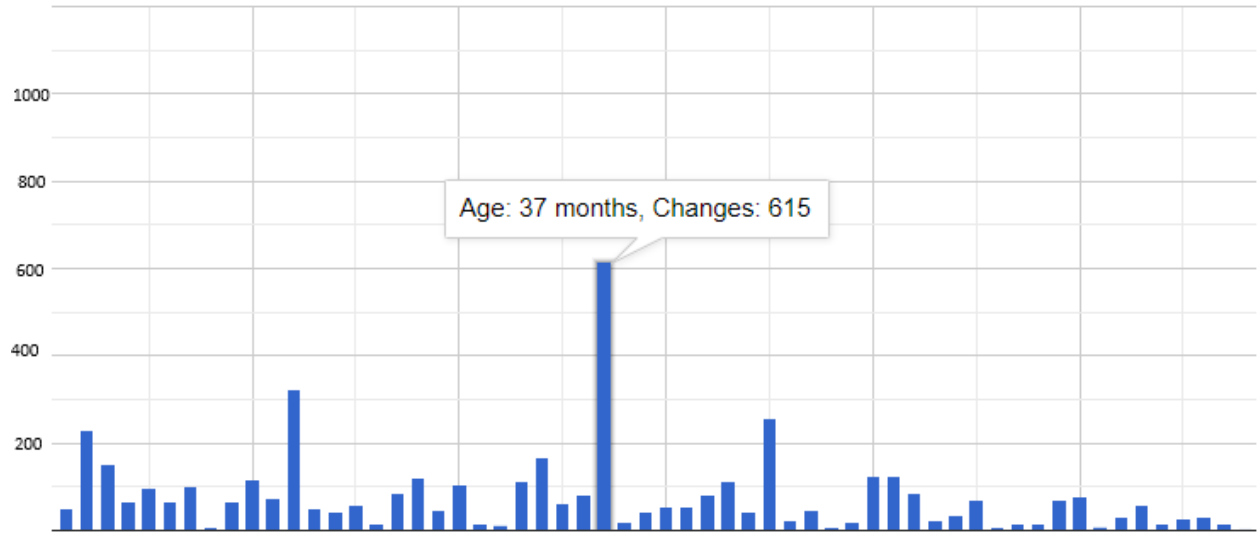

Figure 2.4.2.1 Code age in a repository.

This information serves as a brief introduction, as even without knowing anything additional about the codebase, the bold assumption is that it can be called legacy. If this data is extended with low test and code coverage and the absence of documentation, that fact becomes apparent.

The next problem to solve is what should be done and in what order to make sure that the problem is resolved.

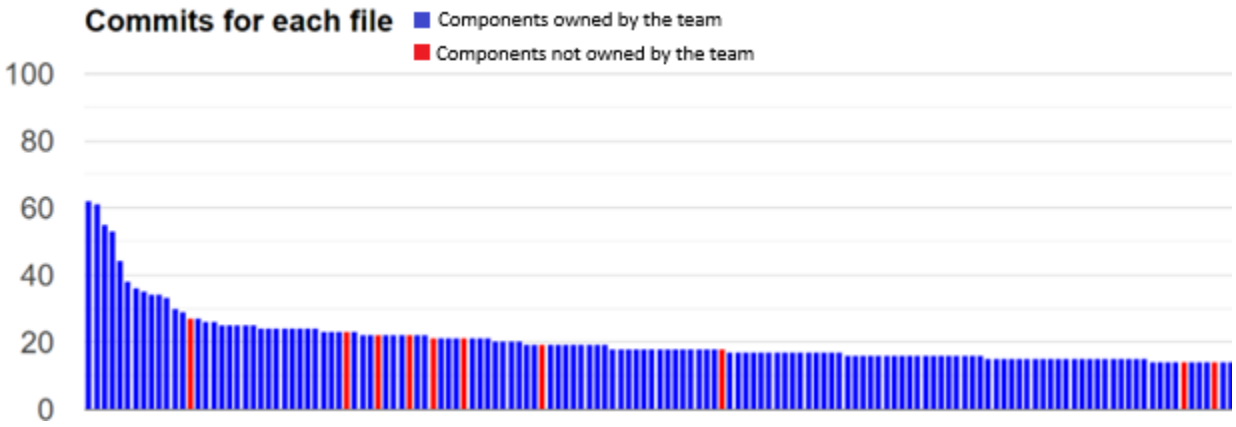

Figure 2.4.2.2 Maintainability index for files in a repository. (each bar is a file)

Figure 2.4.2.2 shows the maintainability index for each file in the repository and, as a result, provides an overview of the repository's activity, enabling it to grasp

the frequency of updates and the complexity of the files based on their line count. As stated above, ownership contributes to the overall score, but it is also marked separately on the diagram. It is done due to its importance, as it indicates that the team does not have enough experience working on those files. This, in turn, aids in the evaluation of the maintainability of the Legacy C++ code, as files with a higher commit frequency and line count may require more attention in terms of testing and refactoring.

Intriguingly, an analysis of Figure 2.4.2.2 reveals that the distribution of commits across the repository adheres to the power law distribution, indicating that only five files are subjected to a disproportionately high number of commits, while most other files have few commits relative to other files. While this may initially appear coincidental, it is worth noting that a similar analysis of repositories with completely different lifespans and programming languages used consistently exhibits the same pattern, underscoring the prevalence of this phenomenon. [13]

In addition to the graph showing commits for each file, there is a table showing the relative number of commits per folder and file, which helps to identify which folders have changed the most. It is worth underlining that if folder separation exists, it will usually indicate the separation between different components.

A part of the repository is displayed in Figure 2.4.2.3. As a side note, it is worth pointing out that the percentage shown is local per each folder, which helps to identify hotspots for each individual component and sub-component.

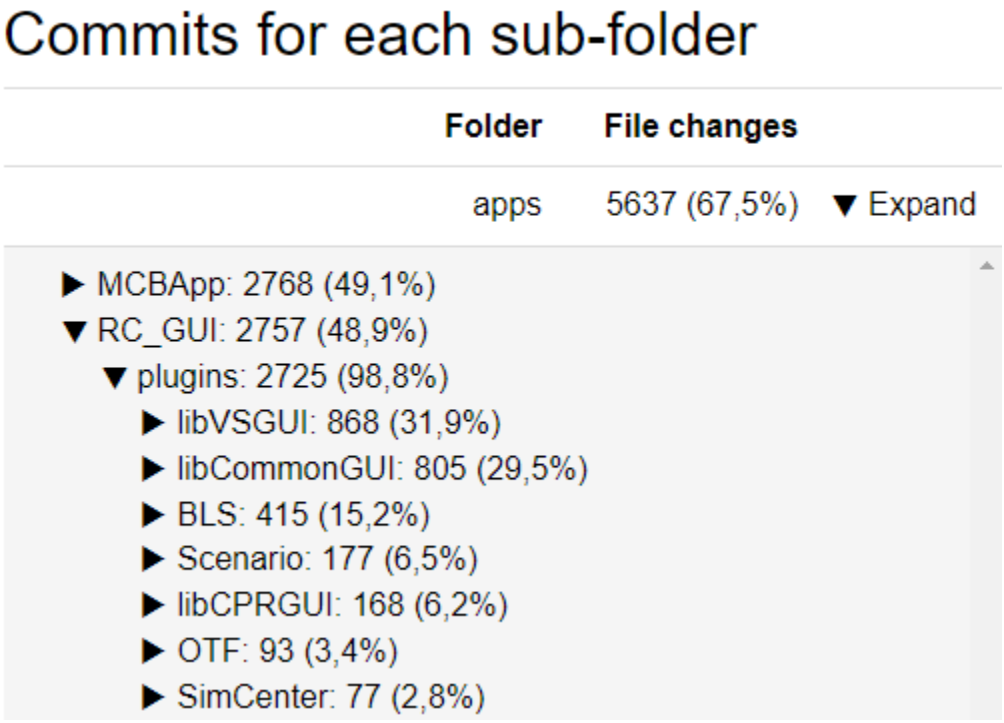

Figure 2.4.2.3 Number of commits for files in a repository.

2.4.3. Analyzing the cost

After components that need modernization are found, the following four steps of identifying the cost of doing and ignoring it should be performed. [15]

2.4.3.1. Estimate the cost of not modernizing

The first step in identifying the cost is to estimate the impact of leaving the code without any changes. This cannot always be done reliably as it is difficult to gather the exact data on what time was spent on analyzing, investigating, fixing, and validating defects. Sometimes, the history of issue tracing systems like «Jira» can provide the time spent on certain tasks to analyze, develop, and test them. Unfortunately, this data shows not only the time spent on outdated components but also the effort put into new features, which makes it less dependable. Additionally, the feedback from the whole development team can be used, as most of the time,

every developer will be able to estimate how difficult it was to work with legacy code.

The general rule here can be to take an educated guess if the data is unavailable. It should also be conservative to not overstate each case in favor of modernizing.

2.4.3.2. Estimate the effort of modernizing

After the cost of ignoring the problem was calculated, the cost of fixing it should be approximated. Usually, it can be done the same way any estimation is done on the project, as this task is no different from fixing a bug or implementing a new feature, so it can be treated as such. It is worth noting that not only the component itself should be reviewed, but all its dependencies should be accounted for.

2.4.3.3. Estimate what time will be saved

There is only one last estimation required to grasp the whole cost of the operation and decide if it is worth doing. It is the rough calculation of what time will be saved after the refactoring is done. This includes both the defects that will be fixed by itself and the decrease in time to extend or fix the component in the future. It is worth underlining that changes in a component will affect all its dependencies. This affects both the increase in the time taken for modernization as other components might require changes and the time saved in the future when other parts of the code using that component are changed. The same approach is used when estimating the cost of not modernizing. That is, use the data available and have educated guesses to finalize the conclusion. The baseline can be to assume that it will decrease the time by half for all defects in that component.

2.4.3.4. Estimate the payback time

The last step in the procedure is to gather all the data and calculate the ratios. The time spent on modernization and the time that will be saved after it is estimated. That allows to get a ratio and calculate how long it will take for modernization of that component to provide benefit for the project in terms of saved development time.

The exact period can be brought down to iterations like releases and/or sprints, depending on the methodology used and the audience. This can be used to get a different picture of how long it will take. For example, «3 months» might give more information for the developers and «2 releases» for the management team.

After all the estimation is done, the team should consider modernizing options with the shortest payback time/period. Certain components with longer payback periods might be discarded due to project-specific causes. An example of such a cause might be a planned discontinuation of a product or a component in a shorter term than its payback time. Unfortunately, no strict guidelines exist as the threshold depends on many factors of the project, such as available resources or delivery deadlines.

While analyzing the cost, the whole project's picture should be considered. If the component has a list of dependencies, then changing that component will influence them. That might increase or decrease the payback time as it might become easier or more difficult to continue the development in a different part of the program that depends on a certain change. That will always depend on the nature of the changes as sometimes it might also affect all the dependencies across the entire system. All pros and cons should not be discarded as it allows one to have a sharp vision of the effect of the work being done.

2.4.3.5. Example of estimating costs and setting priorities

To demonstrate the process of estimating the cost, it will be performed based on the codebase at Laerdal Medical AS organization. Based on the data obtained from the «GitCommitsAnalysis» shown in Figure 2.4.3.5.1, two components, namely «libCommonGUI» and «TextClient», will be discussed in detail. They are selected to show both cases of accepting and rejecting modernization.

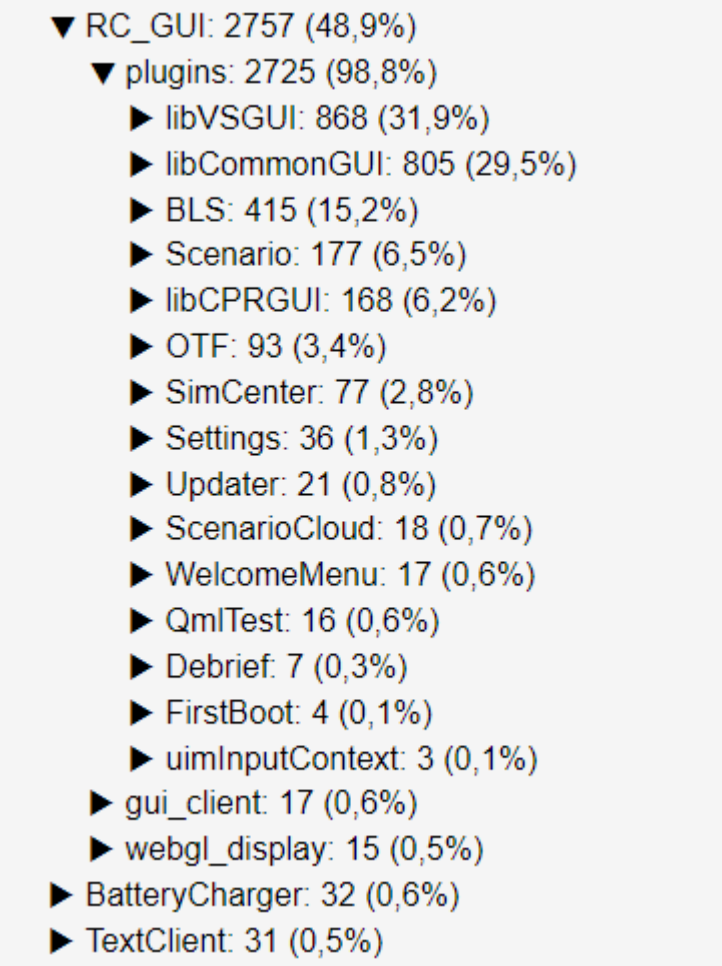

Figure 2.4.3.5.1 Distribution of commits per folder

«TextClient» is a separate application that allows one to control the manikin based on text input. Based on «Jira», which is used to track all the issues on the project, there were no issues with it for the past four releases. The cost of not modernizing it is slim to none because there are no issues with it. It is a separate

application that does not have any dependencies, and the latest change to it was also done three years ago. Theoretically, it is enough to conclude that modernization is not needed right now, as even if it becomes modern and secure, it is quite easy to make a mistake during the process and introduce new bugs. To formally continue the research, the estimation of effort to modernize it should be done. It contains eleven files, each of which is not covered with tests and uses outdated C++ features, so it might take two weeks to completely modernize it. Based on the fact that nobody has worked with this code for several years, the saved time is also almost equal to zero. That means that payback time for the modernization of that component is almost infinite, as time will be spent on it, but no benefit is visible.

On the other hand, «libCommonGUI» is a library used in more than half of the projects related to graphical user interfaces. During the past release only, there were 48 issues tracked in Jira related to it, where each issue took at least four days to develop and at least one day to test. Judging by feedback from developers, most of the time was spent working around existing legacy code, as even if the task is not related to the library, the dependency on it shows the effort. The basic estimate is that developers spend around 78 days as a team just because of the legacy code, in addition to 19 days spent by testers. The numbers are approximate here, and the way to calculate them was to divide the whole time by 2.5, where the assumption is that 40% of the time is spent on legacy code. There are 271 files in that component, which would take around 300 days to refactor completely. The same estimation process was done for the «TextClient» but on a larger scale. Based on the assumptions made when estimating the cost of not modernizing, similar numbers are applicable for the estimate of how much time will be saved. The same numbers can be used, so 97 days per release will be saved. The payback time for that component

is three releases, as each release on the project takes ten weeks, and the payback time is 30 weeks.

Parameter I am running a few minutes late; my previous meeting is running over. Component	«TextClient»	«libCommonGUI»
The cost of not modernizing	Almost 0	78 + 19 days per release
Effort of modernizing	2 weeks (14 days)	300 days
Saved time	Almost 0	97 days per release
Payback time	Almost infinite	3 releases (30 weeks)

Based on two analyzed projects, as the payback time for «libCommonGUI» is 30 weeks and is almost infinite for «TextClient», it is recommended to prioritize «libCommonGUI». The important note here is that only two components were selected for the sake of the example, and every component of the project should be considered to make sure that the process is efficient and that none of the important components of the project are ignored.

2.4.4. Tools selection

In the previous chapters, the aspects of legacy C++ code, its limitations, vulnerabilities, and potential areas for modernization were identified. Certain practices to prioritize and analyze what should be done were described to show the basic direction for the whole procedure. Going deeper into this research, the crucial part of the process is tool selection. Certain things can be completely automated, while others can be approached with more information based on a certain tool's output. Through a comparison of viable options, a list of available options is

provided to ensure informed decision-making for each project. While certain tools might be flexible enough to be used by any project, unique requirements, constraints, limitations, and goals of each project should be considered. Even though certain tools mentioned can be used with any programming language, this paper is about the C++ programming language. So, the benefit of a tool being able to work with multiple languages and the drawback of it being focused on a small list are not considered. Certain tools that are specifically designed for C++ language are also highlighted.

Given the vast number of similar tools on the market, determining which ones will most effectively meet the project-specific needs can pose a significant challenge. Some of the questions to answer are the following:

- **What problem is being solved?** The first step is to identify what features of a certain tool will be used.
- **Who will use the tool?** How many users and what teams will use it? This is mostly needed to evaluate the cost of the tools, as sometimes the pricing depends on the number of users.
- **What is the goal of using the tool?** Is the goal of the tool to improve the readability of the code? Is it to look for performance or securing issues? Or should it create certain files within the fixed format?
- **How can it be integrated with the existing tools?** Which tools will be replaced? How will the tool work with the issue tracking system, a code repository, and any other system on the project?
- **How can it be integrated into the organization?** The software selection should also depend on the workflows and delivery methodology used in the company. What is working well, and in which areas an improvement would be beneficial?

To summarize what tools will be required, they can be divided into the following categories:

- Static code analysis tools (see 2.4.4.1).
- Dynamic code analysis tools (see 2.4.4.2).
- Test and code coverage tools (see 2.4.4.3).
- Code documentation tools (see 2.4.4.4).
- Repository analysis tools (see 2.4.4.5).
- Artificial intelligence (see 2.4.4.6).

It is worth noting that the lists of tools are not extensive and might not contain all the required details to decide which tool to use. It is intended to serve as a baseline for choosing which instrument to use. A lot of the project-specific details might affect what tool to select, and additional research per each project is required to decide.

2.4.4.1. Static code analysis tools

Static code analysis is the analysis of the code performed without executing the program. Most of the time, it is performed based on the program's source code.

Undeniably, static code analysis provides numerous advantages, including automation, enhanced security, and increased speed of development. The greatest benefit that these tools bring is that they can be automated, checks will be done independently from the development team, and results will be produced faster than any manual review.

The most basic tool that every C++ developer cannot avoid is the C++ compiler. While the main purpose of it is to compile code, some analysis of the code is done without executing it. The result is the list of compiler warnings and compiler errors.

While sometimes the compiler is not considered a static code analysis tool, it still produces a list of issues with the code without executing it. Every C++ compiler has a set of settings to configure the severity of the warning to be displayed. However, it is important to note that while all compilers perform some level of static code analysis, not all static code analysis tools are compilers.

The compiler is a mandatory tool to use during development, but because of its focus on compiling the program itself, it falls short of providing extensive code analysis results. More sophisticated static code analysis tools can detect more complex issues that compilers may overlook, such as potential runtime errors, code quality issues, or deviations from coding standards.

The following comparative analysis has been created based on the performed research about the eight different static code analysis tools:

1. «Codacy» is an excellent choice for gaining insight into your code quality. It offers customization and security checks. However, its drawbacks include a higher price point and a complicated setup process.
2. «Veracode Static Analysis» stands out for identifying and rectifying vulnerabilities with real-time feedback. It offers rapid real-time results with high accuracy, but it can be slow when performing the whole codebase analysis and can be challenging to set up.
3. Both «ReSharper» and «CLion», developed by the same company, are superior static code analysis tools integrated within the IDE used only for C++. They facilitate easy navigation, speedier development, and code refactoring but may produce false positives and can be expensive.
4. «SonarQube» is an exceptional self-managed static analysis tool for continuous codebase monitoring. It is flexible, extensible, customizable, and

easy to integrate. However, it may provide false positives, and its interface can be incomprehensive.

5. «Klocwork» is ideal for helping with decreasing the development time and increasing the quality of the code when delivering. It is easy to automate and customize, but the user experience could be enhanced, and the setup process can be difficult.
6. «Helix QAC» is the best static code analyzer for C and C++ programming languages. It complies with ISO/MISRA/AUTOSAT standards, but it may not have a large community for support.
7. «Codiga» excels for customizable static analysis that operates within your IDE and CI/CD pipelines. It has a straightforward setup process, but its interface can be incomprehensive.
8. «Kiuwan» is excellent for automatic code scanning and defect remediation. It offers customization and a good user experience. However, setting it up can be challenging.

2.4.4.2. Dynamic code analysis tools

Dynamic code analysis, in contrast with static code analysis, primarily focuses on analyzing the program at runtime instead of reading the code, so it is performed during the program execution. That results in a set of key issues that they can detect. That set contains but is not limited to, issues related to memory management, like memory leaks or corruptions, and thread synchronization issues, like data races or deadlocks. Dynamic code analysis tools are pivotal for understanding and refining the complex structure of any code.

The following comparative analysis has been created based on the performed research about the five different dynamic code analysis tools:

1. «CHAP» is a powerful tool that specializes in analyzing un-instrumented ELF (Executable and Linkable Format) core files to identify leaks, memory growth, and corruption. Additionally, it provides valuable assistance to a debugger by revealing the status of various memory locations. However, it is worth noting that «CHAP» may present a steep learning curve due to its complexity, and its effectiveness may be limited when it comes to analyzing files other than ELF core files.
2. «KLEE» is a symbolic virtual machine developed on top of the «LLVM» compiler infrastructure. Its major advantage is the ability to generate tests that achieve high coverage on intricate real-world programs. On the downside, «KLEE» may demand considerable computational resources and may struggle with handling external libraries.
3. The «LDRA» tool suite, which encompasses dynamic analysis and testing to various standards. However, potential users should be aware that «LDRA» may be complex to configure and use. Additionally, the licensing cost may be a deterrent for some organizations.
4. «LLVM/Clang Sanitizers» is a set of tools for dynamic analysis, including «AddressSanitizer» for detecting memory errors, «MemorySanitizer» for identifying uninitialized memory reads, and «ThreadSanitizer» for detecting data races in C/C++ programs. These tools excel in spotting a variety of errors and issues in C/C++ programs. However, they may increase both the compilation and execution time of programs and may not identify all types of bugs.
5. «Valgrind» is a versatile instrumentation framework for building dynamic analysis tools. It boasts multiple debugging and profiling tools that help identify a wide range of issues in programs. However, it is worth noting that

«Valgrind» may significantly slow down program execution and may not support all hardware architectures or types of program binaries.

2.4.4.3. Test and code coverage tools

The difference between test coverage and code coverage was explained before, which results in a separate set of tools required for each. For the code coverage, the metric is clear, which is the number of tested lines of code divided by the number of lines in the whole program. Sadly, the same cannot be done for the test coverage. Ideally, test coverage is the number of requirements attained divided by the total number of requirements. The catch is that most of the time, those requirements are stored in a document or any other format that is not accessible to any of the analysis tools. For it to be manageable, the program should follow the test-driven development (TDD) process, where the tests correspond to the requirements, which is not true for most legacy projects.

For those reasons, there are no practical tools applicable for every situation related to the test coverage, but there is an abundance of tools that allow checking the code coverage. The focus will remain on the C++ language, so all the tools below will support C++.

When selecting a code coverage tool, there are a lot of things that should be considered. The first one is the programming language used. The scope of the project is also an important part. For small projects, it does not make sense to buy the enterprise edition of an expensive tool. It also should integrate with the tools currently in use in the company. The main tools for the C++ language are:

1. «BullseyeCoverage» is a code coverage analyzer that provides in-depth and precise coverage metrics. It is excellent for large-scale systems due

to its compatibility with a wide range of compilers and platforms. However, its main drawback is that its interface can be incomprehensive.

2. «Microsoft Visual Studio» is a comprehensive suite that not only offers C++ code coverage but an entire range of tools for development. Its integration with other Microsoft tools is seamless, making it ideal for teams already using the Microsoft ecosystem. However, it may be overkill for smaller projects or teams only needing code coverage tools. It also supports only the Windows platform, which makes it useless if other platforms must be supported.
3. «FrogLogic Coco» stands out for its excellent report generation and the ability to work well with Qt applications. For over one and a half million developers [16] Qt is the main framework used, so a tool that is designed to work with it proves to be highly effective. However, it can be challenging to set up and is not as good as other alternatives outside of the scope of Qt.
4. «Testwell CTC++» is a powerful tool that provides coverage for a wide array of testing levels, from unit to system testing. It is known for its portability and efficiency, and it is able to manage large codebases without a significant impact on performance. However, it lacks a graphical user interface as it is mainly text-based, which can make it less accessible to less experienced developers.

2.4.4.4. Code documentation tools

Code documentation usually boils down to a text that describes the code. It should answer questions like «how», «why», and «what». It can be a separate document in any format that can represent text, preferably rich text, for readability

reasons. However, this approach does not prove to be highly effective. The main pitfall with it is that after the code is updated, the documentation is often forgotten and becomes outdated extremely fast. New people on the project are also not aware of any documentation, so they cannot use or extend it.

That leads to the idea that the documentation should be in the code itself. There are some tools that allow the use of C++ language comments format:

1. «Doxygen» has been a default and a baseline tool for annotating code and generating documentation for a long time. It is easy to use and has widespread adoption in the industry. However, its main downside is that it can struggle with more complex codebases, and the quality of the documentation is highly dependent on the annotations in the source code.
2. «Sphinx» is a powerful, feature-rich tool for creating high-quality documentation. It supports multiple output formats and is highly extensible. One of its drawbacks is that it may be overkill for smaller, simpler projects, and it has a steep learning curve for beginners.
3. «Breathe» is an extension for Sphinx that allows it to read «Doxygen» XML output files. It is useful for projects that want to leverage the power of Sphinx while also benefiting from the «Doxygen» ability to parse C++ code. However, it requires a two-step process to generate the documentation (first generating «Doxygen» XML files and then processing them with «Sphinx»), which may be cumbersome for some projects.
4. «Exhale» is another extension for «Sphinx», designed to create a class and file hierarchies from C++ source code, which can be an extremely useful feature for large codebases. However, it is still relatively new and not as mature or robust as «Doxygen» or «Sphinx», and it also relies on

«Doxygen» to parse the source code, which can be a disadvantage for projects that want to avoid using «Doxygen».

2.4.4.5. Repositories analysis tools

An additional source of valuable information a team can use to set up a direction is the repository itself. Many teams use a version control system (VCS) different from «Git», but it is still statistically one of the most popular, taking around 40% of the enterprise market. [17] and is the easiest to use. Extracting behavioral data will be a completely different procedure for every VCS. As a result of «Git» popularity, it is less sensible to investigate other VCS within the scope of a single research study. The assumption will be that the team works with the «Git» repository, and thus, there is a need to extract behavioral data from it.

The main tools to extract that data are the following:

1. The «GitCommitAnalysis» tool was already described above. It works for any «Git» repository and does not depend on any other tools used by the team.
2. «GitHub» is a developer platform that uses Git, providing distributed version control. While using «Git» development teams use a similar system to store and share code online. While GitHub is not mandatory, it is also used by over 100 million developers. [18] It provides an analysis of the most active user for the repository and the amount of code over time.
3. «Apache Kibble» is an open-source activity reporting platform that can track a project’s code, discussions, issues, and individuals. Its strength is its ability to manage substantial amounts of data and its flexibility due to

its open-source nature. However, it has a steeper learning curve and may require more initial setup compared to other tools.

4. «Pluralsight Flow» is a comprehensive tool that provides detailed insights into your coding workflows, helping you to improve productivity and efficiency. Its strength lies in its data visualization, which allows for easy understanding of complex data. However, it might not be ideal for smaller teams due to its pricing structure, and its interface can be incomprehensive.

Unfortunately, many tools available apart from modified «GitCommitAnalysis» only provide the activity data, which is different from the behavioral data that is the most valuable in the current context.

2.4.4.6. Artificial intelligence (AI) tools

AI-based tools are becoming increasingly prevalent, providing innovative solutions to traditional challenges. These AI tools can assist in a variety of tasks, such as code optimization, bug detection, automatic code refactoring, and even the prediction of future coding errors. Tools like «Copilot» leverage machine learning algorithms to analyze code and provide real-time suggestions. On the other hand, «ChatGPT» is proven to be useful when writing separate chunks of code or changing the input code based on a prompt.

A «prompt» is a text in a natural language (a language that occurs naturally in human speech) that describes to an AI what should be done. There are a lot of different techniques for how the prompt should be composed. The process of creating a prompt that can effectively be processed and understood by an AI model is called prompt engineering. Most of the time, the prompt structure heavily depends on the AI model used and the nature of the task.

However, while these AI-based tools offer promising capabilities, they are not without their limitations and challenges. The effectiveness of these tools is highly dependent on the prompts used to access their capabilities, which often requires additional training for the development team.

Therefore, when selecting tools for handling legacy C++, it is crucial to consider a balance of traditional and AI-based tools, leveraging the strengths of each to enhance the overall development process.

2.4.5. Setting goals and how to get to them

The process of modernizing and securing legacy code, like any other significant task, requires a clear roadmap to ensure success. One way to define it is to use the Objectives and Key Results (or OKRs) goal-setting framework. OKRs are crucial in setting ambitious yet realistic goals (Objectives) and defining measurable ways to track the progress towards these goals (Key Results). Grouping all that was stated above is still not easy, but it is possible to define basic OKRs for the process. It still should be looked at as more of a template of things to look for. Every project has different specifics and needs that will increase or decrease the importance of certain OKRs for the company.

The list of OKRs can have the following points:

1. Objective: Select and configure appropriate tools for the modernization process.

   Key results:

   a. Identify and select a set of tools suitable for the modernization process, considering factors such as compatibility, cost-effectiveness, and ease of use.

   b. Configure these tools to automate as much of the process as possible, aiming for a certain percentage of automation.

2. Objective: Identify components of the legacy code that require modernization.

   Key results:

   a. Conduct a thorough assessment of the legacy code to identify which components require modernization.
   b. Document and categorize these components based on the severity of their need for modernization.

3. Objective: Prioritize the modernization of components based on cost and necessity.

   Key results:

   a. Develop a priority list for modernization tasks, considering the cost and importance of each component.

4. Objective: Identify potential security issues in the legacy code.

   Key results:

   a. Use the selected tools to scan the legacy code, aiming to identify a certain number of potential issues.
   b. Document and categorize these issues based on their severity and potential impact on the system.

5. Objective: Fix identified issues and solve defined tasks in the legacy code.

   Side note: this step should be separated into a list depending on specific tasks identified during the latter research. It might be more specific, like «split the application into independent modules» or «perform refactoring

to allow unit and integration testing to be implemented». Key results:

a. Define and implement solutions for the identified security issues and modernization tasks, aiming to resolve a certain percentage of them within a specific period.
b. Validate the fixes to ensure they have effectively resolved the issues and have not introduced new ones.

Many of the elements in that list might be more specific or less specific as they might be needed for a certain project, or some important steps for certain projects might be omitted. It still serves as a baseline from which to start the work. While the key results are not specific here, as there are no clear numbers, criteria for improvements were described above and should also be used to specify the target goals in a measurable way.

While objectives and key results goal-setting framework is a wonderful way to define goals, it is not the only one. Another powerful tool to use is S.M.A.R.T. (Specific, Measurable, Achievable, Relevant, Time-bound). They are different in their approach and focus. S.M.A.R.T. goals are designed to be highly specific, with an emphasis on defining goals that are attainable, strictly relevant, and bound by a definitive timeline. This approach ensures clarity and encourages realistic goal setting. However, OKRs aim to set ambitious and aspirational objectives, with a focus on measurable key results that track progress towards them. S.M.A.R.T. goals can provide clear, actionable steps, while OKRs can inspire the team to strive for significant improvements and innovations.

While setting any goals, it is important to understand the team and current processes in the company. For example, if the team is frustrated dealing with legacy code, it would be more effective to use OKRs to increase morale. If the company

already uses a certain goal-setting framework, it would be better to use it as the team is already used to it.

2.5. Analysis of results

In this paper, a comprehensive analysis was conducted to understand the reasons for software modification, prioritize components for modernization, and estimate the associated costs. This chapter delves into the findings and interpretations of the analysis conducted.

A short instruction on how the procedure should flow was created to serve as a short reference when modernizing the code using the paper (see Addition A. Instructions on how the paper can be applied).

A repository used directly by a leading company in the field of healthcare education was analyzed, which has been written in C++ for over 20 years. Other open-source repositories were looked at as

1. «pygwalker» has been written in Python for over a year. It contains ~12500 lines and has ~800 commits. [19]
2. «gogs» written in Go over ten years. It contains ~65500 lines and has ~13500 commits. [20]
3. «Rocket.Chat» was written in TypeScript for over nine years. It contains ~109500 lines and has ~31500 commits. [21]

The output diagrams from «GitCommitsAnalysis» showing the maintainability index for each of the repositories in the same order they are mentioned above are shown below in figures 2.5.1-2.5.3.

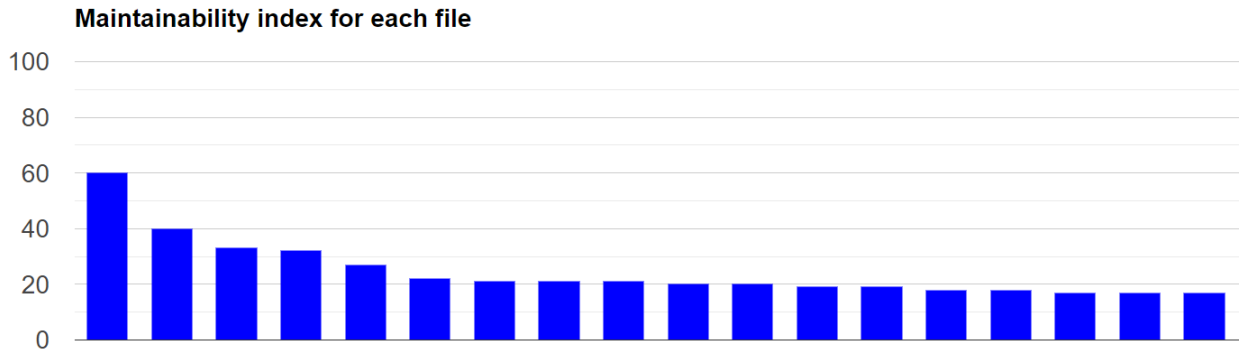

Figure 2.5.1 «pygwalker» maintainability indexes for each file

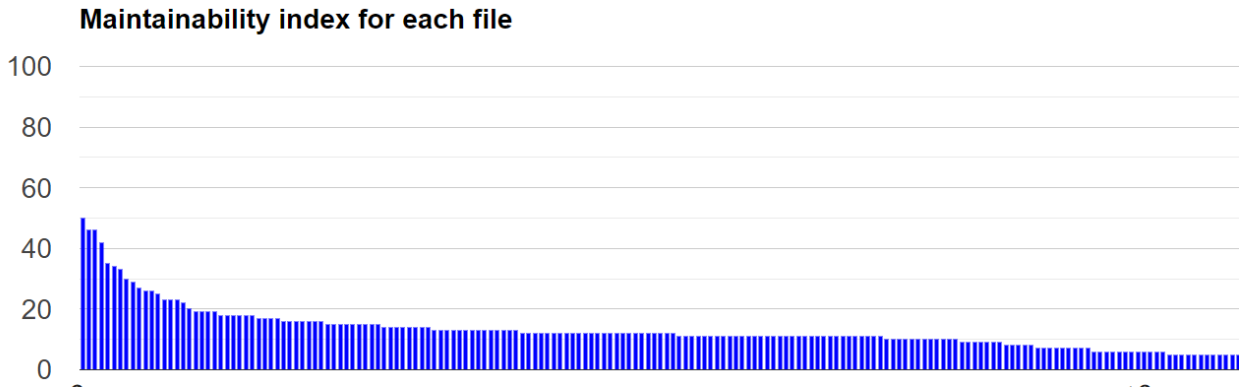

Figure 2.5.2 «gogs» maintainability indexes for each file

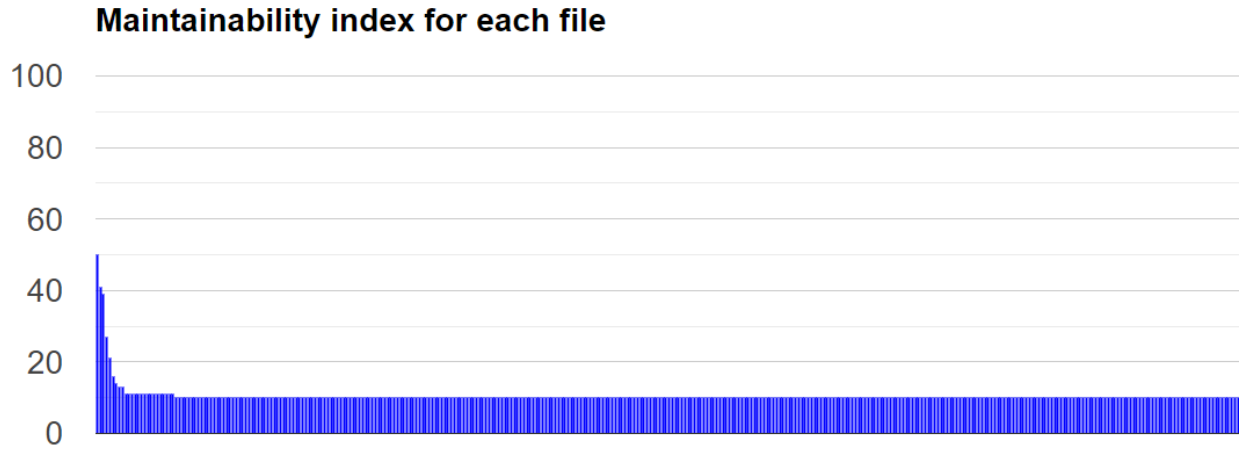

Figure 2.5.2 «Rocket.Chat» maintainability indexes for each file

For all those repositories, it was confirmed using found tools (see 2.4.4) that the maintainability index distribution (which is mostly based on a number of commits) is close to the power law distribution. It is worth noting that that fact

becomes increasingly apparent the more commits there are for the repository, which represents the time it is being developed. That confirms that it is not affected by the scope of the project, programming language, or production company, as all those projects are independent of each other. The complexities are also different, as the number of commits and lines might vary a lot. For example, «Rocket.Chat» has 8.8 times more lines than «pygwalker».

Before committing to any change in the code base, there should be apparent reasons for it. The analysis started with understanding those reasons and identified four main drives: adding a feature, fixing a bug, improving code structure, and optimizing resource usage. The main goal of the modernization process is to minimize the time required to do every one of those tasks. As a result, during that process, both new and old functionalities should not be changed at all. The main task is to improve code structure to allow easy modification and test coverage, thus drastically improving the maintainability of the code.

After the reasoning is established, the object of modernizing should be identified. While it might be obvious that certain components should be changed, it is not always easy to be sure that doing so will have the expected impact. For that reason, the components should be prioritized based on certain data, which can be done using the behavioral data of the code base. While it allows for the setting of priorities, before diving in, the cost analysis should be performed as well. Some changes might not bring enough benefit in a specified amount of time and thus are not worth doing.

While the process might sound easy on paper, it often poses several challenges that can sometimes be dealt with automatically. For that, a thorough analysis of available tools should be done. The main tools to work directly with the code identified in the research are static and dynamic code analysis tools, test and code

coverage tools, code documentation tools, repository analysis tools, and AI tools. Some tools to collaborate with a team are also important, including goal-setting frameworks, which allow one to clearly communicate with a team to establish alignment within what tasks should be performed in what order and how to measure if the work is done correctly. A list of popular tools was broken down and explained in detail.

In conclusion, this analysis provided profound insights into the various aspects of modernizing legacy C++ code. It highlighted the necessity of modernization, the strategy for prioritizing components, the cost implications, the importance of using the right tools, and the effectiveness of different goal-setting frameworks in the modernization process.

3. CONCLUSIONS

As a result of the research conducted, the main goal of the work was achieved. A large volume of references and literature was analyzed, and a vast number of different tools were looked upon and compared. Methods and recommendations on how to work with legacy C++ code were developed, and fundamental issues when modernizing and securing it were identified.

The results obtained from this analysis provide critical insights into the process of modernizing legacy C++ code. They reveal the necessity of such modernization to maintain compatibility with contemporary systems, optimize performance, and enhance security. Moreover, they offer a strategic approach to prioritize components for modernization, emphasizing resource optimization by focusing on high-risk and frequently modified components. The analysis also sheds light on the cost implications, emphasizing the need to weigh these against the long-term benefits of modernization. The invaluable role of tools for static/dynamic code analysis, code documentation, and repository analysis is highlighted, demonstrating their effectiveness in improving code quality and facilitating the modernization process.

The research on modernizing and securing legacy C++ code opens several promising perspectives for further investigation and development. One potential area of future research could be the development of more sophisticated tools and methodologies for code analysis and documentation. This could enhance the efficiency of the modernization process and provide deeper insights into the legacy code. Furthermore, the study of organizational and human factors in the modernization process could be expanded to include team topologies.

To conclude, this research is an important contribution to product development, based on the necessity of updating outdated systems to ensure the relevance of existing software in the rapidly evolving technological landscape.

4. REFERENCES

[1] Laerdal Medical AS, "About Laerdal," [Online]. Available: https://laerdal.com/us/about-us/. [Accessed Apr 2024].

[2] T. Whitney, T. and D. Coulter, "Tools for upgrading C++ code," Microsoft, 17 Aug 2021. [Online]. Available: https://learn.microsoft.com/en-us/cpp/porting/ide-tools-for-upgrading-code?view=msvc-170.

[3] M. C. Feathers, Working Effectively with legacy code, New Jersey: John Wait, 2004.

[4] R. Bellairs, "8 Tips for Working with Legacy Code," Perforce, 13 Nov 2018. [Online]. Available: https://www.perforce.com/blog/qac/8-tips-working-legacy-code.

[5] Gartner, "Stefan Van Der Zijden," 2024. [Online]. Available: https://www.gartner.com/analyst/52795.

[6] Gartner, "7 Options to Modernize Legacy Systems," 05 Nov 2019. [Online]. Available: https://www.gartner.com/smarterwithgartner/7-options-to-modernize-legacy-systems.

[7] H. L. Nyakundi, "Best Practices for Secure Programming in C++," Mayhem, 29 Jun 2023. [Online]. Available: https://www.mayhem.security/blog/best-practices-for-secure-programming-in-c.

[8] ADHITHI, "Code Coverage vs Test Coverage," 3 May 2023. [Online]. Available: https://testsigma.com/blog/code-coverage-vs-test-coverage/.

[9] Sourya, "DeepSource documentation coverage," DeepSource, Apr 2020. [Online]. Available: https://discuss.deepsource.com/t/how-is-documentation-coverage-calculated/65/2.

[10] Wikipedia, "Peter D. Welch," 7 Mar 2024. [Online]. Available: https://en.wikipedia.org/wiki/Peter_D._Welch.

[11] R. Timbó, "Code Complexity: What It Is & How to Measure It," Revelo, 18 Aug 2023. [Online]. Available: https://www.revelo.com/blog/code-complexity.

[12] Wikipedia, "Source lines of code," 6 Apr 2024. [Online]. Available: https://en.wikipedia.org/wiki/Source_lines_of_code.

[13] A. Tornhill, "Seven Secrets of Maintainable Codebases," Dev Day, 3 Oct 2016. [Online]. Available: https://www.youtube.com/watch?v=a74UkJxKWVM.

[14] A. Simonsen, "GitCommitsAnalysis repository," CoderAllan, 24 Oct 2021. [Online]. Available: https://github.com/CoderAllan/GitCommitsAnalysis.

[15] M. Cohn, "4 Steps to Persuade a Product Owner to Prioritize Refactoring," Mountain Goat, 4 Feb 2020. [Online]. Available: https://www.mountaingoatsoftware.com/blog/4-steps-to-persuade-a-product-owner-to-prioritize-refactoring.

[16] Qt, "Qt Group story," 2023. [Online]. Available: https://www.qt.io/group.

[17] G2, "Best Version Control Software," Mar 2024. [Online]. Available: https://www.g2.com/categories/version-control-software.

[18] T. Dohmke, "100 million developers and counting," GitHub, 25 Jan 2023. [Online]. Available: https://github.blog/2023-01-25-100-million-developers-and-counting/.

[19] Open-source, "pygwalker," [Online]. Available: https://github.com/Kanaries/pygwalker.

[20] Open-source, "gogs," [Online]. Available: https://github.com/gogs/gogs.

[21] Open-source, "RocketChat," [Online]. Available: https://github.com/RocketChat/Rocket.Chat.

5. ADDITIONS

Addition A. Instructions on how the paper can be applied

1. Select automated tools to do static/dynamic code analysis, code coverage, documentation. (see 2.4.4)
2. Set criteria for improvement (see 2.1.4). After goals are defined, e.g., «write tests to have coverage», «make sure that there are no compiler warnings».
    a. Set test coverage. 70%-85% as a baseline.
    b. Do initial research to define what is outdated and current issues (see 2.3).
        i. Apply static/dynamic code analysis («SonarQube», «Valgrind», etc.)
        ii. Configure CI/CD (apply those tools automatically)
        iii. Hold security audits. What are potential security incidents?
        iv. Analyze behavioral data. Which components have worked without changes for years? («GitCommitAnalysis»)
3. Set priorities and understand what needs to be changed.
    a. Use behavioral data to identify complex components that are changed frequently (2.4.2)
    b. For those components, identify the cost (2.4.3) How much does it cost to change the component, and what will be paid if we ignore it (i.e., +1 hour for each change when there are usually five changes in 1 release)?
    c. Based on the cost estimate, payback time.
    d. Adjust the priority based on the payback time.

   i. If 1 component is on the 2nd priority, but the payback time is ten times less than move it forward)

4. Fix it. (and make sure to use all the tools you found to do as much as possible)

АНОТАЦІЯ

Дана дипломна робота складається зі вступу, основної частини, висновка, списку використаних джерел та додатків. Обсяг роботи становить 59 сторінок. Основна частина займає 45 сторінок та містить 7 рисунки і 2 таблиці. Документ містить 21 посилань на використані джерела.

Дане дослідження присвячено вивченню та узагальненню стратегій модернізації та захисту застарілого коду C++. Метою цього дослідження є розробка чіткого шляху модернізації, який слугує міцною базою та є придатним для будь-якого проекту та будь-яких команд, щоб уникнути його розробку щоразу. Крім того, його мета полягає в розгляді проблеми з різних кутів, щоб поєднати кілька критеріїв, які зазвичай розділені. Для досягнення цієї мети було досліджено багато методів та інструментів.

Результатом дослідження є короткий список кроків, які слід виконати для досягнення мети, представленої в Додатку А. Він написаний у простій формі та може бути переданий команді розробників для навігації по даній роботі. Надається велика кількість настанов для кожного кроку, щоб забезпечити врахування всіх факторів проекту. Застосування цих настанов дозволяє командам розробників уникнути виконання роботи, яка може бути пропущена або виконана двічі.

Ключові слова: застарілий код, модернізація, захист, автоматизовані інструменти.

ANNOTATION

This diploma thesis consists of an introduction, the main part, a conclusion, a list of used sources, and additions. The volume of work is 59 pages. The main part occupies 45 pages and contains 7 figures and 2 tables. The document contains 21 references to used sources.

This study is devoted to the research and generalization of strategies for modernizing and securing legacy C++ code. The purpose of this research is to develop a clear modernization path that serves as a solid baseline and is applicable for any project for all the teams to avoid developing it every time. In addition, it aims to look at the issue from different angles to combine several criteria which are usually separated. To achieve this goal, many methods and tools were investigated.

The result of the study is the short list of steps that should be performed to achieve the goal presented in Addition A. Instructions on how the paper can be applied It is written in a simple form and can be handled to a development team to navigate the paper. An extensive number of guidelines for every step are provided to ensure all project factors are accounted for. Applying these guidelines allows the development teams to avoid doing work that may be omitted or doing something twice.

Keywords: legacy code, modernization, securing, automation tools.