KAPITEL 3 / *CHAPTER 3* [3]
RESEARCH OF THE PROCESS OF MAINTENANCE OF SOFTWARE SYSTEMS CODE

DOI: 10.30890/2709-2313.2024-32-00-009

Introduction

The quality of the software code directly affects the ability to scale and maintain the software for the long term. Scalability means the ability of a system to cope with an increase in workload or the integration of new functions without significant changes to its architecture. To achieve this, the software must be well structured and modular, allowing new components to be added or existing ones to be modified without negatively affecting other parts of the system. In terms of maintenance, supporting large-scale software systems can take up significant resources if the code is difficult to understand or difficult to modify. High-quality code reduces the time required to fix bugs or adapt the system to new conditions, which in turn reduces technical debt. Technical debt is the accumulated problems or outdated solutions that make it difficult to further develop the software. Low-quality code can quickly increase technical debt, forcing developers to spend more time and resources on fixing deficiencies, while high-quality code helps to avoid these problems, ensuring the stability and flexibility of the system in the long run.

Thus, the quality of software code is the foundation for effective software maintenance and scaling. Well-designed, tested, and documented code provides flexibility in making changes, increases system stability, and contributes to the long-term operation of the product. In modern software development, maintenance often takes up the majority of a project's time and budget—frequently exceeding the effort dedicated to initial development. This highlights the critical importance of investing in maintainability, as it ensures the system can adapt to evolving requirements, integrate new features, and sustain its performance over time.

Prioritizing code quality during development not only reduces maintenance costs but also minimizes technical debt, which can otherwise slow progress and increase

[3] ***Authors:*** *Zelenko Dariia, Gorbova Alexandra*

risks as the system grows. High-quality code supports more efficient debugging, testing, and enhancement processes, ensuring the software remains reliable and scalable during its whole lifecycle. By focusing on maintainability, organizations can achieve a robust foundation for innovation while maintaining a high standard of quality in their systems.

3.1. Analysis of the current state of the problem

Code maintenance and scalability have long been major challenges in software engineering, evolving as the complexity and size of software systems grows. Historically, early software systems were often developed in isolation, with limited attention to the durability or extensibility of the code base. However, in the second half of the 20th century, as computing needs expanded, the limitations of such approaches became clear. In the past, systems were often highly interdependent, poorly documented, and prone to failures when new features were introduced or when scaling was required to support a larger number of users or bigger datasets. These challenges highlighted the need for more structured approaches to ensure code quality, maintainability, and scalability. As a result, both academic research and industry practice began to place greater emphasis on these aspects, enabling the development of more stable, efficient, and resilient systems capable of handling growing demands and evolving requirements.

In this context, the book “Clean Code” [1] is an important milestone in the evolution of software engineering philosophy. Viewed through a scientific and historical lens, Clean Code contains many lessons learnt from decades of software engineering failures and successes. The book is not a revolutionary text in the sense of introducing completely new concepts, but rather a consolidation of best practices aimed at overcoming common pitfalls in code maintenance and scaling. It reflects the growing recognition, which began in the 1960s and 1970s, that software systems are not static objects. They must evolve, adapt and scale over time, and this requires a disciplined

approach to coding that facilitates long-term maintenance.

From a scientific point of view, the need for maintainable and scalable code has been a subject of increasing study since the early days of computer science. Research in software engineering, especially in the areas of program analysis and code quality metrics, emphasizes that the structure and readability of code have a significant impact on its maintainability. The article [2] examines the limitations of current metrics like cyclomatic complexity and code readability. It argues that these measures often fail to capture nuanced improvements, such as those made during code refactoring. The study suggests enhancing metrics to better align with modern software practices and emphasizes their importance for maintainability and scalability.

Historically, the desire for easy-to-maintain and scalable code aligns with the development of agile methodologies in the late 1990s and early 2000s. Agile's focus on iterative development, customer collaboration, and rapid adaptation to change created a need for codebases that could be modified frequently without compromising code integrity. Clean Code fits neatly into this paradigm, offering guidelines for writing code that can be easily understood and modified by different developers working in short cycles. It reflects a broader movement in software development towards flexibility and adaptability, which was a response to the failures of the rigid, waterfall development processes that dominated previous decades. The article [3] emphasizes the importance of ensuring high code quality in the face of increasing complexity of software systems. The authors emphasize the need to apply the principles of Clean Code, which includes simplicity, readability and maintainability of the code. Even with modern technology and test automation, it is important to maintain clean code, as it promotes better team collaboration, reduces errors, and simplifies further support and scaling of programs. This allows you to create reliable software that can easily adapt to changing requirements and technologies.

The current state of the art of code maintenance and scalability remains a challenging issue in software engineering, despite the progress in development practices and tools. The scientific literature continues to explore this issue deeper, reflecting the growing complexity of software systems and the increasing demands for

flexibility, performance, and reliability in the modern software development life cycle. Research on these topics addresses both the technical and cognitive factors that affect code maintenance and scalability, and there is recognition that despite decades of research, these issues persist in new forms as systems grow in size and complexity. The book [4] discusses the practice of refactoring as a key component of creating quality code. In particular, the analysis of code smells shows that clear identification of problematic code areas and their elimination at early stages can significantly reduce long-term support costs. The study [5] highlights that poor code quality makes it difficult to understand, increases task completion time, and leads to significant time wastage due to technical debt. It focuses on the link between code issues, such as code smells, and development team productivity. It is underlined that investments in maintaining code quality and eliminating technical debt can significantly increase development efficiency and reduce costs in the long run.

Modern research on code quality, writing and maintenance focuses on several key aspects related to the complexity of modern systems and the impact of this complexity on the costs and efficiency of developers. As the scale of systems and functionality requirements grow, the problem of maintaining code support becomes more and more difficult, prompting researchers and practitioners to search for new solutions. Modern researches emphasize that high-quality code not only ensures easy maintainability but also reduces overall development costs.

However, code review remains a challenging task due to its complex nature. Automated tools cannot take into account all contextual aspects [6], such as business logic or specific architectural decisions. This requires the integration of automated methods with manual checks and documentation improvements. An important aspect is the financial component of support.

In general, modern scientific research confirms that ensuring high-quality code writing and support requires a comprehensive approach. This includes improving automated tools, integrating them with human checks, implementing refactoring on a regular basis, and developing standards for complexity assessment. These efforts are critical to reducing costs and increasing team productivity in large projects.

3.2. The complexity of the development process

In the software lifecycle, maintenance is often the most resource-intensive phase, consuming a significant portion of the overall budget. Expanding software systems that become more complex due to interdependencies between components (functions, modules, objects) can result in code that is difficult to understand, modify, and test - driving up maintenance costs.

One of the main problems is that complexity often increases gradually during development. Initial design decisions or quick fixes implemented to meet deadlines or user requirements can lead to confusing structures. Such poorly designed systems, sometimes referred to as 'spaghetti code', make maintenance much more difficult. Developers tasked with maintaining such systems are often faced with the challenge of deciphering complex relationships between code elements, which increases the time and effort required for tasks such as bug fixes, feature updates and testing.

Another key challenge is that traditional engineering methods are not sufficient to deal with the arbitrary complexity natural to software systems. Unlike physical systems, where complexity is often predictable and natural, software complexity is highly abstract and can be unpredictable. This nature of software makes it prone to deterioration in structure as it evolves, unless constant efforts are made to simplify and refactor the code base. However, such efforts are often time-consuming and costly, which further increases maintenance costs.

Insufficient attention to maintainability at the beginning of the software development process results in systems that are difficult to extend or adapt to new requirements. This not only increases costs, but also reduces the overall reliability and performance of the system, as maintaining complex software increases the likelihood of new bugs. Complexity management therefore becomes a critical factor in ensuring the long-term sustainability and quality of software systems.

To study the problem, such concepts of mathematical statistics as finding a trend equation based on data from a study [7] analyzing the cost of developing various operating systems will be used.

A trend equation is a mathematical model that describes how costs change over time (or with versions). All graphs are based on polynomial equations, and this model allows to take into account the acceleration of cost growth, not only linear or quadratic growth, but also more complex changes.

This will allow to qualitatively assess the costs of system development and its upgrades. Because the OS code consists of millions of lines that are responsible for the operation of various subsystems - from the kernel to drivers, file systems, network protocols and graphical interfaces. This multi-layered nature requires clear planning, especially in terms of support, modification, and functionality updates. Large projects of this type involve the development and use of sophisticated mechanisms for version control, testing, change management, and code metrics. In addition, such projects have large budgets due to their high criticality and long historical development.

3.3. Evolution of the growth in the cost of developing and supporting different versions of operating systems

Based on the research data, were built graphs of the ratio of estimated development costs and estimated support costs with an average value and trend lines for different OSes (Figures 1, 2, 3), trend calculated equations and final analytics performed.

Analyzing the graph, the following can be observed:

• Estimated development costs (billion USD), costs have been steadily increasing from Windows NT to Windows 8.

• Estimated maintenance costs (USD billion), with an even faster growth rate than development costs.

• Trend equation for development: $y = -0.0095x^6 + 0.2723x^5 - 3.0382x^4 + 16.806x^3 - 47.615x^2 + 70.725x - 27.111$. This function indicates a complex cost growth dynamic with certain periods of slowdown.

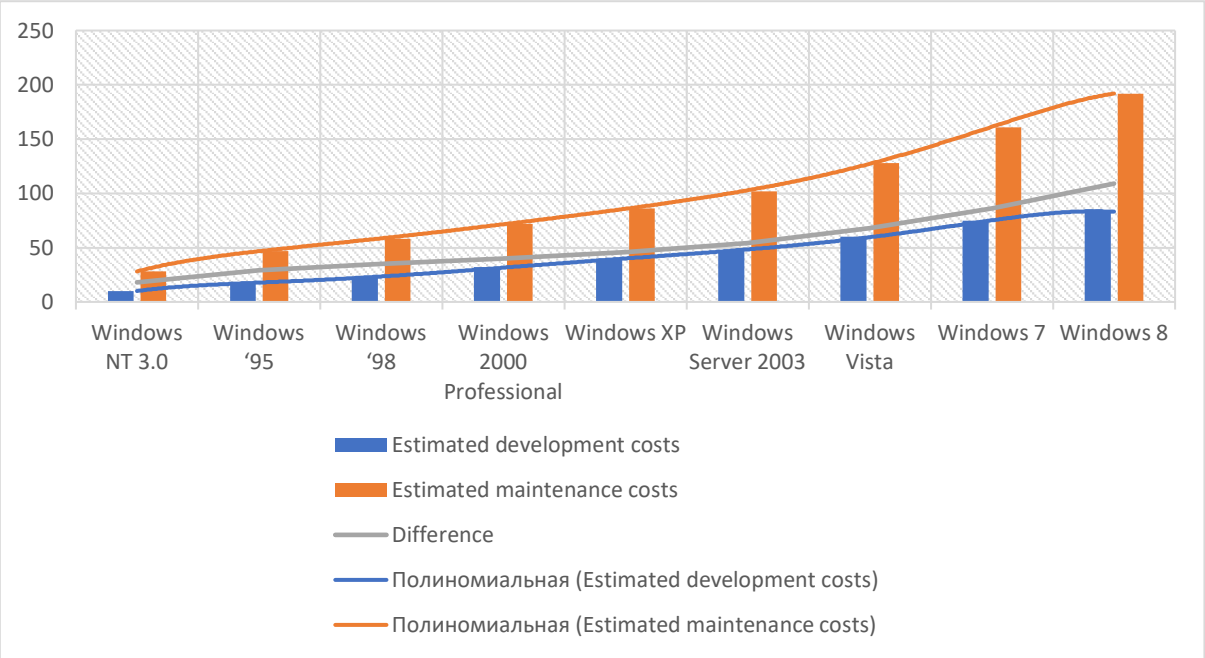

Figure 1 – Graph of the ratio of estimated development and support costs for different versions of Windows

• The trend equation for maintenance: $y = -0.0102x^6 + 0.2902x^5 - 3.2536x^4 + 18.575x^3 - 56.603x^2 + 98.934x - 29.889$. Maintenance costs show a similar but even greater growth rate.

Conclusion: Windows development and maintenance costs are significant, and maintenance takes up the bulk of the budget. The trend equations for development and maintenance costs show a complex nonlinear dynamic that includes periods of growth, deceleration and possible decline. Both equations have a polynomial structure of the 6th degree, which indicates significant variability over time. The development costs equation is characterized by an acceleration in the mid-term, but with a possible decline in the long run due to the negative impact of the high-degree coefficients. For maintenance, a similar pattern is observed, but with more intense growth, indicating a faster increase in expenditure in the medium term. Both processes indicate a complex variability in costs, with growth gradually slowing down in the later stages.

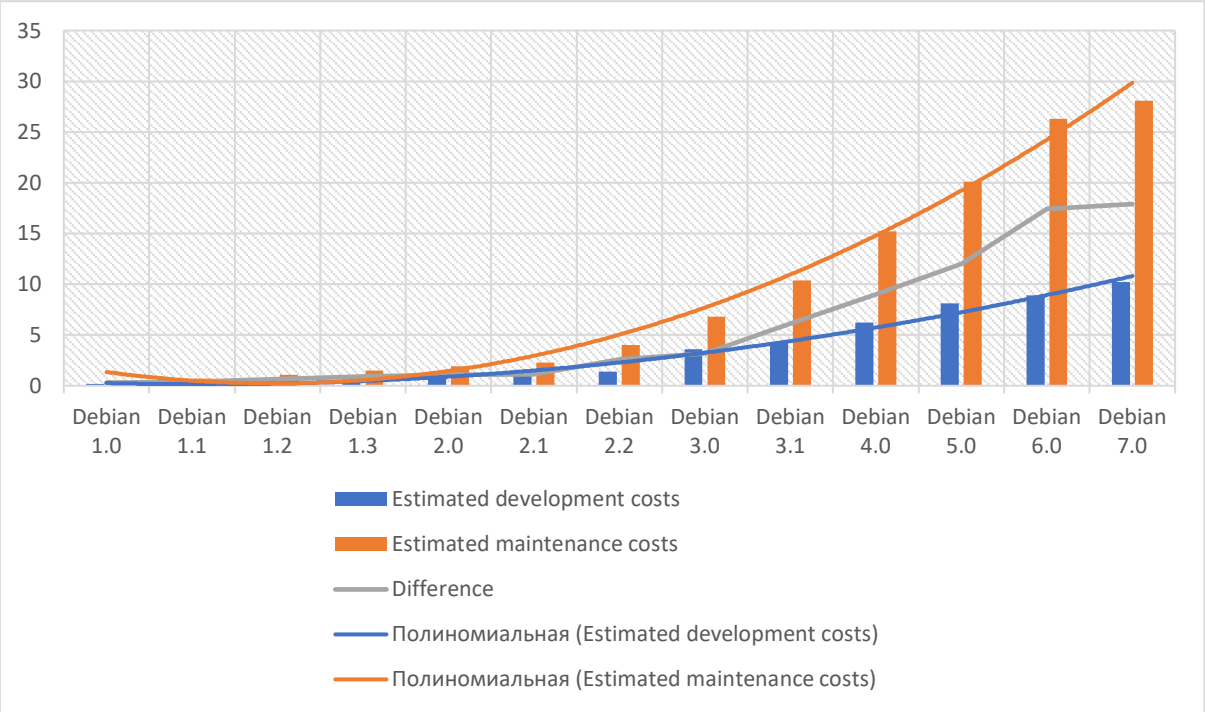

Figure 2 – Graph of the ratio of estimated development and support costs for different versions of Debian Linux

Analyzing the graph, the following can be observed:

• Estimated development costs (USD billion), with a gradual increase in these costs over time.

• Estimated maintenance costs (USD billion), these costs are significantly higher than development costs and have a significant growth rate.

• Trend equation for development:$y = 0.0903x^2 - 0.3857x + 0.5657$. This is a quadratic function, indicating an accelerated increase in costs in later versions of Debian.

• The trend equation for maintenance: $y = 0.2936x^2 - 1.7358x + 2.7993$. The quadratic function indicates a similar growth rate, but maintenance costs remain significantly higher.

Conclusion: the growth of development and maintenance costs is accelerating, with maintenance taking the dominant share. The trend equations for development and maintenance costs show a quadratic pattern of changes, indicating that costs are increasing at an accelerating rate with each new version of Debian. For development, the growth rate is moderate, with initial costs relatively low, but acceleration becomes

more noticeable with increasing versions. In contrast, maintenance costs show a much higher baseline and faster growth rate, which highlights the greater complexity and amount of work required to ensure the stability and functionality of the system. This indicates that support becomes a major cost item in the long term, far outstripping development costs.

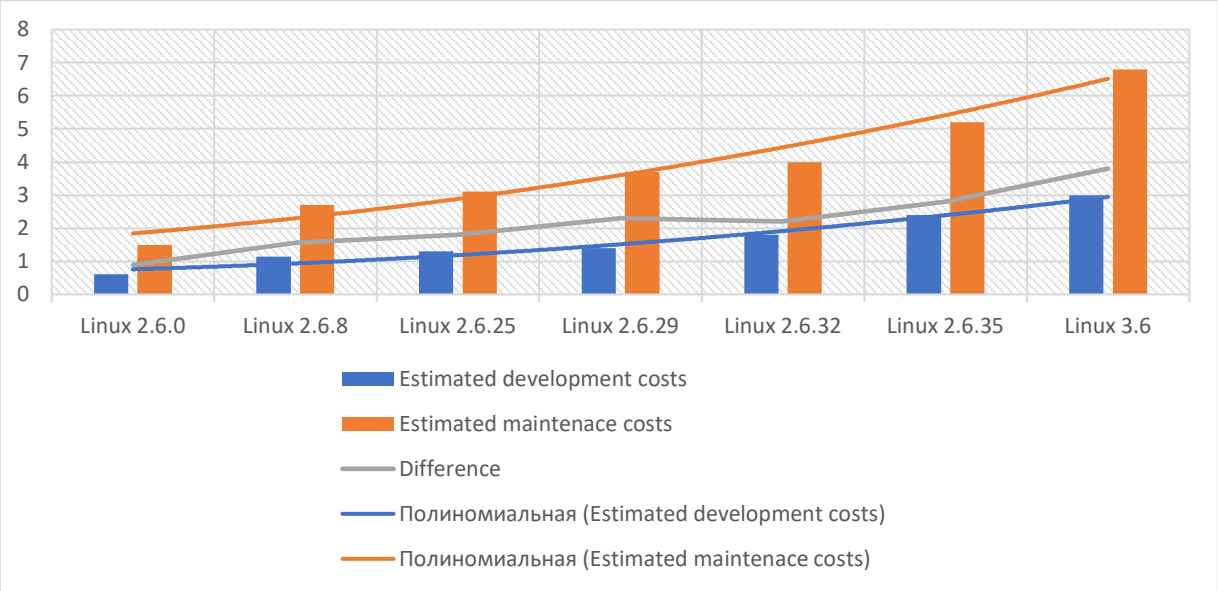

Figure 3 – Graph of the ratio of estimated development and support costs for different versions of the Linux kernel

Analyzing the graph, the following can be observed:

• Estimated development costs (billion USD). kernel development is gradually becoming more expensive with each version.

• Estimated maintenance costs (USD billion), although maintenance costs are increasing, they remain relatively low.

• Trend equation for development:$y = 0.0375x^2 + 0.0639x + 0.6586$. The quadratic function indicates a moderate but steady increase in costs.

• The trend equation for maintenance: $y = 0.0643x^2 + 0.2643x + 1.5143$. The growth in maintenance costs follows a similar trend.

Conclusion: Linux kernel development is showing a steady increase in costs, but in general, these costs remain lower than Debian or Windows. The trend equations for development and support costs show a quadratic relationship, indicating a moderate

but steady increase in costs over time. Development costs are growing more slowly, indicating a relatively small increase in the volume or complexity of work as new versions are released. At the same time, support costs are growing faster, starting from a higher baseline, demonstrating their important role and the need for more resources. Both trends confirm the sustainability of costs with a gradual acceleration in the long term.

The equations confirm that development and maintenance costs increase not only proportionally with each version, but also with an increasing rate.

1.1.Dynamics of development and maintenance costs

- Development covers the initial investment in creating each version of a system or kernel. It is usually less expensive than maintenance, but increases over time.
- Maintenance is associated with the costs of support, upgrades, bug fixes and adaptation to changes in technology.
- For Windows, maintenance is growing faster than for Linux.
- In Debian Linux, maintenance is also growing rapidly, but a little slower than in Windows.
- With Linux Kernel, maintenance remains the least expensive.

Maintenance is a major cost element in the long term, especially for large operating systems (Windows and Debian).

1.2.Cost ratio

The difference between development and maintenance costs becomes more significant with each successive version:

- In Windows, this difference is particularly strong, reflecting the high cost of supporting a large ecosystem (with millions of users).
- In Debian Linux, the ratio is closer, but maintenance costs still exceed development costs.
- In Linux Kernel, development and maintenance costs remain close to each other, which indicates an efficient architecture.

Windows requires more investment at each stage than Debian or Linux Kernel, which indicates the complexity of its support.

1.3.Dynamics of polynomial trends

Polynomial equations show not just stable growth, but growth with acceleration.

- In Windows, you can see a sharp increase in costs in later versions (e.g. Windows 7 and 8).
- For Debian Linux and Linux Kernel, costs also increase with acceleration, but at a slower rate.

The most rapid growth in costs is observed in Windows, slightly less in Debian, and the most moderate in Linux Kernel.

1.4.Conclusions on cost-effectiveness

- Windows: Maximum costs due to the complexity of the system, its popularity and duration of support (Legacy systems).
- Debian Linux: Costs are rising, but the system remains cheaper than Windows due to the open source and community working on it.
- Linux Kernel: The most cost-effective, as costs grow the slowest, due to its narrow specialization and open approach.

Maintenance is a major expense for all systems. With each version, the costs increase rapidly, especially for Windows and Debian Linux.

The cost-effectiveness of the Linux Kernel is due to its openness and modular architecture, which reduces support costs.

For Windows and Debian, costs are increasing, but Windows leads the way in terms of total cost due to its higher support requirements.

Maintenance costs increase significantly with the size and complexity of the OSes in question. Also, as the complexity and size of the system increases, the proportion of effort devoted to maintenance steadily increases. While at the initial stages, about 60% of total effort is spent on support, at later stages of software development, this share can increase to 80% or more. This reflects the fact that maintaining large, complex systems requires more resources and time compared to initial development.

Conclusions

The analysis of development and maintenance costs shows that maintenance is always more expensive than the initial development costs. This can be seen in all three projects: Windows, Debian Linux and Linux Kernel.

The main reasons are:

1. Duration of maintenance: Support for each version of the system lasts much longer than the development phase, which accumulates costs.

2. Expanding the user base: Increasing the number of users requires more resources for bug fixes, updates and adaptations.

3. Integration and compatibility: Over time, the number of external dependencies (software, hardware) increases, which increases the cost of maintaining stability and security.

The most prominent example is Windows, where maintenance costs dramatically exceed development costs due to the need for long-term support and the complexity of the ecosystem. For Debian Linux and Linux Kernel, the trend is the same, although maintenance costs grow more slowly, especially for Linux Kernel due to its modular structure.

Thus, maintenance costs are a determining factor in the overall cost of a system, and their growth is inevitable in the long run.