

|  |
| --- |
| Tero Ala-Hulkko |
| Modernization of a legacy codebase |
|  |

|  |
| --- |
| School of Technology |
| 2024 |

VAASAN AMMATTIKORKEAKOULU

Cloud-Based Engineering**TIIVISTELMÄ**

Tekijä Tero Ala-Hulkko

Opinnäytetyön nimi Modernization of a legacy codebase

Vuosi 2024

Kieli suomi

Sivumäärä 43 + 3 liitettä

Ohjaaja Rayko Toshev

Tiivistelmässä käytetään rivinväliä 1. Tiivistelmä kirjoitetaan täydellisinä lauseina. Sähketyyliä ei käytetä, ei liioin alleviivauksia, kursivointeja tai harvennuksia. Tiivistelmän pituus on 75 - 200 sanaa. Teksti on hyvä jakaa kappaleisiin luettavuuden takia, esimerkiksi seuraavan kaavan mukaan:

Ensimmäisessä kappaleessa kuvaillaan lyhyesti tutkimuksen taustaa ja tutkimusongelmaa sekä tutkimuksen tehtävää.

Toisessa kappaleessa kuvaillaan lyhyesti tutkimuksen teoreettista viitekehystä, keskeisiä käsitteitä sekä tutkimuksessa käytettyjä menetelmiä ja tutkimusaineistoa.

Kolmannessa kappaleessa esitetään lyhyesti tutkimuksen keskeisiä havaintoja ja tuloksia sekä keskeisiä johtopäätöksiä ja tulkintoja.

Note: Only Finnish students include an abstract in Finnish if the thesis is writ-ten in English.

Avainsanat Vanha koodikanta, Visual Basic 6, Modernisointis

VAASAN AMMATTIKORKEAKOULU

Masters in Cloud-Based Engineering

**ABSTRACT**

Author Tero Ala-Hulkko

Title Modernization of a legacy codebase

Year 2024

Language Finnish

Pages 43 + 3 Appendices

Name of Supervisor Rayko Toshev

The purpose of this thesis was to develop a transitioning strategy from a legacy codebase to a modern solution. The study will investigate reasons why a codebase should be modernized, how it should be done, and what it will cost.

The main programming language in the target company is Visual Basic 6. The first part of this thesis focuses on the challenges VB6 has today. This study will investigate what kinds of features is it lacking and how modern tools are no longer tailored towards it. This study will also include a risk analysis on what kind of challenges would the target company face if Microsoft dropped VB6 support from its future releases.

The second part focuses on how can marketing and sales leverage modern solutions. At the core of any development should be business need and value generation. It is important to realize market value when beginning large projects. This thesis includes a qualitative interview with the marketing and sales personnel from the target company.

The final part of this thesis focuses on how the target company could achieve transitioning to a modern codebase. This thesis will investigate software architecture, which programming languages should be used, and what other technologies can be employed. The final product of this thesis is a migration roadmap for the target company.

Keywords Legacy codebase, Visual Basic 6, Modernization

**CONTENTS**

Tiivistelmä

ABSTRACT

[1 Introduction 7](#_Toc167061877)

[2 tools used 8](#_Toc167061878)

[3 visual basic 6 deprecation 9](#_Toc167061879)

[3.1 VB6 shortcomings 9](#_Toc167061880)

[3.2 Business impact 10](#_Toc167061881)

[4 Upgrading is difficult 12](#_Toc167061882)

[4.1 High risk of failure 12](#_Toc167061883)

[4.2 Wrong approach 12](#_Toc167061884)

[4.3 Approval from organization 13](#_Toc167061885)

[5 modernizing the architecture 14](#_Toc167061886)

[5.1 Starting point 14](#_Toc167061887)

[5.2 Ensuring high quality codebase 14](#_Toc167061888)

[5.3 Changing the architecture 15](#_Toc167061889)

[5.3.1 Monolithic architecture 16](#_Toc167061890)

[5.3.2 Separated front-end and back-end 17](#_Toc167061891)

[5.3.3 Service-oriented architecture 18](#_Toc167061892)

[5.4 Programming language 19](#_Toc167061893)

[5.4.1 Leveraging Visual Studio features 19](#_Toc167061894)

[References 20](#_Toc167061895)

[Appendices 23](#_Toc167061896)

**LIST OF FIGURES AND TABLES**

[Figure 1 Continuous integration workflow 15](#_Toc167061864)

[Figure 2 Monolithic architecture 17](#_Toc167061865)

[Figure 3 Architecture with separated front-end and back-end 18](#_Toc167061866)

[Figure 4 Service oriented architecture 19](#_Toc167061867)

[**Table 1.** Examples of contents of the thesis. 3](#_Toc138775303)

**APPENDICES**

**APPENDIX 1.** What to Include to Appendices

**APPENDIX 2.**

# Introduction

The term “Legacy code” is not clearly defined. Some say it is code without tests. Some extreme definitions say code becomes legacy code as soon as it is written. The definition I like the best is “Legacy code is valuable code that you’re afraid to change” (Carlo, 2024). This definition leaves room for improvement.

Legacy code comes with several hindrances that have business impact. Legacy code cause longer development times, longer QA times, and difficulties implementing continuous integration practises. The development team can simply do less in a given amount of time. This in turn can have customers feeling ignored and unhappy. (Magalhães, 2020)

This thesis explores some of the issues and unrealized gains the target company is experiencing. The focus is on the codebase written in Visual Basic 6. VB6 is a programming language that was developed by Microsoft in 1998. This thesis will cover what kind of support is expected from Microsoft, how modern tools apply for VB6, and some modern programming concepts VB6 fails to deliver.

In the second part of this thesis, a qualitative interview will be presented. This interview focuses on the business benefits that a modern software can have from the perspective of marketing and sales business units.

The final part of this thesis will be about modernizing the software architecture of the target company. This section explores ways to achieve migration from VB6 code to C# code. A proof-of-concept migration of a smaller application that is in production use today is presented.

# tools used

# visual basic 6 deprecation

Microsoft has reduced the amount of support they are providing for VB6. In an article Microsoft published, they say they are committed to “It just works” compatibility for VB6 Windows systems up to Windows 11. The article has been revised several times to include the latest Windows version. (Microsoft, 1.4.2024)

Despite the promising name of the support scheme, the support only extends to the VB6 runtime files. Runtime files will work for a minimum of 5 years after the release of Windows 11. After 5 years, Microsoft promises 5 years of extended support. The support offered by Microsoft is limited to serious regressions and critical security issues. Development platform for Visual Basic 6 has been out of support since 2008. (Microsoft, 1.4.2024)

## VB6 shortcomings

Due to the lack of further development, several modern concepts and architectures are not supported on VB6. These shortcomings can make developing applications more difficult and costly.

Visual Basic 6 and any application written on it, can only be run as a 32-bit application (Microsoft, 1.4.2024). 32-bit applications have several disadvantages compared to the newer 64-bit architecture. One disadvantage a developer may run into is the limited memory capacity of the 32-bit architecture. 32-bit applications can only reserve up to 4GB RAM. This can be a limiting factor and a liability if applications cannot handle the limited environment. (GeeksForGeeks, 5.2.2024)

Several modern IDEs support direct integration with the version control system GIT. Since git was developed later than the end of support for VB6 IDE, they cannot be integrated directly. This is the case for several other modern third-party technologies as well. Several third-party extensions exist for VB6. These are often used to enable some very rudimentary features that the VB6 IDE is missing. These features include the functionality of mouse scroll wheel and the inclusion of tabs in the IDE (VBForums).

## Business impact

Finding skilled professionals becomes more difficult as the popularity of a technology decreases. This can make finding candidates for hire take longer and increase candidate’s expectations of salary. Additionally, the total cost of ownership increases as legacy systems are more difficult to maintain.

Codebase with a lot of technical debt makes it more difficult to implement new features. Repairing technical issues with a codebase itself has risks involved. Common issues caused by repairing technical debt include accidentally removing features still in use and regression. Inability to quickly implement features runs the risk of being left behind competitors. (Birchal, chapter 1)

TIOBE is an organization that tracks and analyses the popularity of programming languages. TIOBE’s data is based on availability of skilled engineers, courses, and third-party vendors. According to their statistics, Classic Visual Basic has fallen from 2% popularity in 2015, to 1% popularity in 2024. They define Classic Visual Basic to include both VB6 and VBA, a programming language used withing applications such as Excel. Additionally, they make it known that due to ambiguity between all the versions of Visual Basic, there is only 50% confidence in assigning which Visual Basic should be credited. (TIOBE, 5/2024)

The number of job postings for VB6 has decreased. Meanwhile, salaries for VB6 developers have increased faster than some its peers. In the UK, the proportion of job postings in IT sector citing VB6 has decreased from roughly 1.8% to less than 0.1%. In the same period, salaries have increased from roughly 30 000£ to 60 000£. In the same period, C# developer salaries have increased from 40 000£ to 60 000£. (ITJobsWatch, 10.5.2024)

Total cost of ownership adds up as inefficiencies increase. Nearly half of UK employees say they waste more than 3 hours a day due to inefficient systems. This amounts to nearly half of those worker’s working hours. Employees also report dissatisfaction about the tools they use. Maintenance of legacy systems alone can account for 10-15% of a company's budget. These costs arise from cross-platform interfaces, ongoing management, and complex integrations among other things. (Audacia)

# Upgrading is difficult

Previous chapters established the shortcomings of using dated technologies and how they increase costs for the organization. Even with all the good upgrading would do, risks and costs have deterred many organizations from moving forward with upgrades.

## High risk of failure

Complex codebases are difficult to migrate to a newer standard. Older codebases often lack documentation. Details about the system, its integrations, and interactions can only be read from the code. Manually migrating code to a newer language is prone to errors, often leads to missing features, and in general takes a significant amount of effort. This significant amount of effort could instead be going towards new marketable features. This can make the whole project seem like a high-risk low-reward situation. (Mathijs T. W.)

Automating the migration process can help reduce change of failure. Automating the migration can be done iteratively until the correct result is achieved. This can be much faster than learning each part of the code and the manually programming it again. For large projects automation can be the only feasible way to accomplish a migration project. (Mathijs T. W.)

## Wrong approach

Code migration projects can fail for many reasons. Sometimes the failure is a project that get cancelled due to problems. Even a completed migration project may not always provide the expected benefits.

It is difficult to commit to a full rewrite of the current system. During the rewrite, developing of new features may need to be halted, especially for monolithic code architectures. The time that development needs to be halted can be long for large applications. During this time critical bugs may be discovered and some features that have been sold to customers may have to be implemented. This creates the need to fix the same bugs and develop the same features on 2 systems: the production system and the new one, still under construction. Extra development and other delays can stretch the timetable to the point that development of the new system is simply stopped. At worst the company can be left in a situation where they will need to maintain two separate systems and consider building a third system to replace them both. (Carlo)

Instead of fully rewriting a system in one go, the project can sometimes be split into multiple smaller phases. Smaller rewrites can reduce the amount of risk. Each phase should provide value and if the project is discontinued, some benefits will still have been gained. One way to divide the work is to split the codebase into logical components such as billing, till, and article management. (Birchal, chapter 3)

## Approval from organization

Large refactoring of codebase takes time and resources.

# modernizing the architecture

## Starting point

Any modernization effort must begin by getting to know the system that is getting modernised. It is important to gather information about the current state of the system, where to focus efforts first, and set up monitoring to verify progress.

Automated tools can provide information about various aspects of the codebase. Automated tools are particularly useful in detecting code that violates styling rules and certain types of bugs in the code. If tools flag a certain area of the code, it can be a good indication to focus efforts there. Tools should not be fully trusted to provide all the information needed and can flag code that is working well as a bug. (Birchal, chapter 2)

Performance of code should be monitored and measured. Performance tests should exist in both production systems and in systems before and after refactoring. Performance tests should be written into code and batches of data can be processed to measure the time it takes for the program to complete. Results of the experiment can be written into a log file for human verification. (Birchal, chapter 2)

## Ensuring high quality codebase

Tools can be integrated into the IDE that flag code that does not abide by rules set by the organisation or code that may introduce bugs and vulnerabilities. By themselves they rely on programmers frequently checking the output of the tools. Continuous inspection helps ensure that no problem flagged by the tools goes unnoticed. Continuous inspection can be a part of continuous integration. Continuous inspection makes use of an automated workflow that gives feedback to the developer upon completion. (Birchal, chapter 2)

The automated workflow (Figure 1) for continuous integration makes use of several different components. When the developer is ready with a feature, they can commit the code to a version control system. Committing code to version control system triggers an automatic build in the build server. Any problems with the build get noted and feedback is sent to the developer. The committed code is optionally rejected, and developer can fix any issues and recommit the code. (Birchal, chapter 2)

Kuva, joka sisältää kohteen teksti, diagrammi, kuvakaappaus, Suorakaide

Kuvaus luotu automaattisesti

Figure 1 Continuous integration workflow

Unit tests are another paramount feature in any high quality.

Code review is the practice of reviewing another programmer’s code before it can be merged into the master branch of version control system. Code review is useful for increasing cohesion in the codebase. If a piece of that is being reviewed does not follow the patterns agreed by the team, the reviewer can reject it. Suggestions can also be given if the reviewed code has mistakes in it, this can be particularly useful when onboarding junior programmers. When putting code up for review, the programmer is also sharing information about changes they are making to the codebase. (Birchal, chapter 3)

## Changing the architecture

Making changes to architecture is a large project. It is paramount to establish clear goals before beginning the work. The goals should be tangible and actionable properties such as splitting the codebase into modules or improvements to the build process. To have a cohesive end product, it is important to agree within the team how the codebase needs to look like in the end. (Birchal, chapter 5)

The target company of this thesis is more or less built as a monolithic application. Architectures most worth considering are keeping the current architecture, a monolithic architecture. Second option is to change the architecture to an architecture with front-end and back-end separated into their own projects. Third option would be to opt for a service-oriented architecture.

Different software architectures bring with different benefits and challenges. Any benefit obtained by changing the architecture needs to outweigh the workload changing architecture brings. (Birchal, chapter 5)

### Monolithic architecture

Monolithic architecture (Figure 2) is the simplest architecture to develop. This is due to functions being readily available in the same codebase. Monolithic architecture should result in fast applications as there is reduced need to talk to other services or applications. (Birchal, chapter 5)

Downsides of monolithic architecture include unexpected interactions in the codebase. Because codebase is a single entity, making changes to code has more opportunities to unintentionally affect other code. Increasing isolation in the code is one reason to avoid monolithic architecture. Another byproduct of low isolation is catastrophic failures when one part of the code is failing. If the application is run as a server, monolithic architecture is the least efficient to scale up. (Birchal, chapter 5)

Kuva, joka sisältää kohteen teksti, kuvakaappaus, diagrammi, Fontti

Kuvaus luotu automaattisesti

Figure 2 Monolithic architecture

### Separated front-end and back-end

Presenting user interface and business logic can be separated into their own application tiers (Figure 3). This type of architecture is typically used in web applications. Communication between tiers happens through APIs. Separating the two tiers makes the code easier to understand. User interface and business logic existing in separate applications promotes separation of concerns. Often there are two developing teams focused on user interface and business logic. The technology stack can vary significantly between the two tiers. (Birchal, chapter 5)

Compared to a monolithic architecture, communication though APIs is more complicated. Communication must be carefully engineered for it to work as expected. Additionally, error handling must be much more thorough as communicating over network APIs can cause issues that are out of hands of the programmer. (Birchal, chapter 5)

**Kuva, joka sisältää kohteen teksti, kuvakaappaus, diagrammi, muotoilu

Kuvaus luotu automaattisesti**

Figure 3 Architecture with separated front-end and back-end

### Service-oriented architecture

Service-oriented architecture (Figure 4) is composed of several services that are isolated from the rest of the architecture. Developing services in this way makes it harder to accidentally introduce regression in the rest of the ecosystem. Scaling the application is also much improved as you only need to scale the services that need more resources. (Birchal, chapter 5)

Service-oriented architecture adds operational overhead. Developing and testing all the different services becomes more complicated. Additionally, tracking metrics and bugs is more difficult. Because of the isolation in SOA, code is often duplicated in the different services. This is due to multiple teams working in different services have limited knowledge of each other’s work. (Birchal, chapter 5)

Kuva, joka sisältää kohteen teksti, diagrammi, viiva, kuvakaappaus

Kuvaus luotu automaattisesti

Figure 4 Service oriented architecture

## Programming language

When choosing a programming language, several technical and non-technical factors need to be considered. Non-technical factors are all about the team working on a project. Team’s existing knowledge plays a big role in which language is the best choice. Every new technology has a learning curve, pre-existing expertise reduces friction at the start of the project. Additionally, not all programming languages are equally difficult to get started with. When considering new hires, lower barrier of entry increases the number of candidates that can be considered. (Fauerbach)

From a technical standpoint, language selection depends on the solution that is being built. Different languages are specialized in different types of applications, such as desktop or web applications. (Fauerbach) Another factor that should be considered is how well the language integrates with existing systems.

### Leveraging Visual Studio features

# References

Carlo, N. What is Legacy Code? Is it code without tests?. Understand Legacy Code. Retrieved 2024-05-05. https://understandlegacycode.com/blog/what-is-legacy-code-is-it-code-without-tests/

Magalhães, G. Legacy Code: a dead end for your project?. Medium. Retrieved 2024-05-05. https://medium.com/what-really-matters/legacy-a-dead-end-for-your-project-dc466d24a3a5

Support statement for Visual Basic 6.0 on Windows. Microsoft. Retrieved 2025-05-09. https://learn.microsoft.com/en-us/previous-versions/visualstudio/visual-basic-6/visual-basic-6-support-policy

Difference Between 32-bit and 64-bit Operating Systems. GeeksForGeeks. Retrieved 2025-05-10. https://www.geeksforgeeks.org/difference-32-bit-64-bit-operating-systems/

TIOBE Index for May 2024. TIOBE. Retrieved 2025-05-10. https://www.tiobe.com/tiobe-index/

The Classic Visual Basic Programming Language. TIOBE. Retrieved 2025-05-10. https://www.tiobe.com/tiobe-index/classic-visual-basic/

Programming Language Statistics. ITJobsWatch. Retrieved 2025-05-10. https://www.itjobswatch.co.uk/IT-Job-Market/UK/Programming-Languages

VB6 IDE AddIns - What's essential?. VBForums. Retrieved 2025-05-12. <https://www.vbforums.com/showthread.php?804911-VB6-IDE-AddIns-What-s-essential>

Audacia. 2019. The cost of legacy IT. Medium. Retrieved 2025-05-12. https://medium.com/@audaciatech/the-cost-of-legacy-it-1c1fa6217e59

Schuts MTW, Aarssen RTA, Tielemans PM, Vinju JJ. Large-scale semi-automatedmigration of legacy C/C++test code.Softw Pract Exper. Retrieved: 2025-05-12. <https://onlinelibrary.wiley.com/doi/epdf/10.1002/spe.3082>

Carlo, N. The Ship of Theseus to NOT rewrite a legacy system from scratch. Understand Legacy Code. Retrieved 2024-05-13. https://understandlegacycode.com/blog/ship-of-theseus-avoid-rewrite-legacy-system/

Birchal, C. 2016. Re-Engineering Legacy Software. Manning Publications.

Fauerbach, C. 2020. How Do I Choose the Right Programming Language for My Team?. Retrieved 2025-05-13. https://medium.com/capital-one-tech/lessons-from-the-leadership-trenches-how-do-i-choose-the-right-programming-language-for-my-team-36222aa78b38

# Appendices

APPENDIX 1

**WHAT TO INCLUDE TO APPENDICES**

You can enclose as appendices for example a questionnaire used in the study or other material that is related to the study.

Material that the client wished to classify can be enclosed as an appendix; in this case, the appendix is not included in the published version submitted to Theseus or in the hardbound version.

**THE TITLES AND NUMBERING OF APPENDICES**

The appendices are numbered and given a title. When you refer to an appendix in the text, use the correct referencing practice. Remember to mention the number of the appendix.

APPENDIX 2