

|  |
| --- |
| 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 English

Sivumäärä 43 + 3 liitettä

Ohjaaja Rayko Toshev

Tämän opinnäytetyön tarkoituksena oli kehittää siirtymisstrategia vanhentuneesta koodikannasta moderniin ratkaisuun. Tutkimuksessa selvitetään syitä, miksi koodikantaa tulee uudistaa, miten se tulisi tehdä ja mitä se tulee maksamaan.

Kohdeyrityksen pääasiallinen ohjelmointikieli on Visual Basic 6 (VB6). Tutkimuksen ensimmäisessä osassa keskitytään VB6:n haasteisiin. Tutkimuksessa selvitetään, millaisia ominaisuuksia VB6:ltä puuttuu ja miten nykyaikaiset työkalut eivät enää tue sitä.

Tutkimuksen toisessa osassa keskitytään siihen, miten kohdeyritys voisi siirtyä nykyaikaiseen koodikantaan. Tutkimuksessa selvitetään ohjelmistoarkkitehtuuria, käytettäviä ohjelmointikieliä ja muita hyödynnettäviä teknologioita. Koko prosessia ohjaa yrityksen tarve. Yrityksen tarpeen selvittämisessä on hyödynnetty myynnin ja markkinoinnin haastattelua ja teknisen puolen apua.

Tutkimuksen kolmas osa keskittyy migraation malliesimerkin tekemiseen. Koodikannan osa siirretään vanhasta koodikannasta nykyaikaiseen REST rajapintaan. Rajapintaa kuluttamaan kehitetään nettisivupohjainen käyttöliittymä Angular.js-kielellä. Lopuksi vanha koodi siirretään käyttämään uutta rajapintaa siirtymäajan helpottamiseksi.

Lopuksi tutkimus sisältää yhteenvedon prosessista. Tutkimuksen lopputuote on kohdeyritykselle siirtymätiekartta.

Avainsanat Vanha koodikanta, Visual Basic 6, Modernisointi, REST, Angular.js

VAASAN AMMATTIKORKEAKOULU

Masters in Cloud-Based Engineering

**ABSTRACT**

Author Tero Ala-Hulkko

Title Modernization of a legacy codebase

Year 2024

Language English

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. This study will investigate what kinds of features VB6 lacks and how modern tools are no longer tailored towards it.

The second part of the research focuses on how the target company could transition to a modern codebase. This involves exploring software architecture, programming languages, and other relevant technologies. The entire process is driven by the company's specific needs. To determine these needs, interviews were conducted with sales and marketing personnel, along with input from the technical team.

The third part of the research centres on creating a migration model example. A portion of the codebase is migrated from the legacy codebase to a modern REST interface. A web-based user interface is developed in Angular.js to consume the interface. Finally, the legacy code is adapted to work with the new interface to facilitate a smooth transition period.

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 8](#_Toc191767530)

[1.1 Background 8](#_Toc191767531)

[1.2 Research methods 9](#_Toc191767532)

[1.3 Structure of the thesis 10](#_Toc191767533)

[2 LegaCy woes 12](#_Toc191767534)

[2.1 Visual Basic 6 deprecation 12](#_Toc191767535)

[2.1.1 VB6 shortcomings 12](#_Toc191767536)

[2.1.2 32-bit environment 12](#_Toc191767537)

[2.1.3 IDE 13](#_Toc191767538)

[2.1.4 Business impact 13](#_Toc191767539)

[2.2 Technical debt 14](#_Toc191767540)

[2.3 Forced to upgrade 17](#_Toc191767541)

[3 Upgrading is difficult 18](#_Toc191767542)

[3.1 High risk of failure 18](#_Toc191767543)

[3.2 Approach 18](#_Toc191767544)

[3.3 Migration case study 20](#_Toc191767545)

[3.4 Economics 22](#_Toc191767546)

[3.4.1 Cost-benefit analysis 22](#_Toc191767547)

[3.4.2 Risk analysis 22](#_Toc191767548)

[4 modernizing the architecture 24](#_Toc191767549)

[4.1 Starting point 24](#_Toc191767550)

[4.2 Ensuring high quality codebase 25](#_Toc191767551)

[4.2.1 Continuous inspection 25](#_Toc191767552)

[4.2.2 Testing the code 26](#_Toc191767553)

[4.2.3 Single-responsibility principle and code isolation 28](#_Toc191767554)

[4.2.4 Code review 29](#_Toc191767555)

[4.2.5 Measuring maintainability 29](#_Toc191767556)

[4.3 Software architectures 31](#_Toc191767557)

[4.3.1 Monolithic architecture 31](#_Toc191767558)

[4.3.2 Separated frontend and backend 32](#_Toc191767559)

[4.3.3 Service-oriented architecture and micro services 33](#_Toc191767560)

[4.4 Migrating the architecture 34](#_Toc191767561)

[4.4.1 Strangler fig pattern 35](#_Toc191767562)

[4.4.2 Transitional architecture 36](#_Toc191767563)

[4.4.3 Targets for displacement 36](#_Toc191767564)

[4.5 Web service architectures 37](#_Toc191767565)

[4.5.1 Representational state transfer 37](#_Toc191767566)

[4.5.2 GraphQL 38](#_Toc191767567)

[4.6 Programming language 38](#_Toc191767568)

[5 Choosing technologies 39](#_Toc191767569)

[5.1 Architecture 39](#_Toc191767570)

[5.1.1 Backend 40](#_Toc191767571)

[5.1.2 Frontend 41](#_Toc191767572)

[5.2 Language and IDE 42](#_Toc191767573)

[5.2.1 Language 42](#_Toc191767574)

[5.2.2 Project type 42](#_Toc191767575)

[5.2.3 Unit test 43](#_Toc191767576)

[5.3 Technologies 43](#_Toc191767577)

[5.3.1 API strategy 43](#_Toc191767578)

[5.3.2 Continuous inspection 44](#_Toc191767579)

[5.4 Methodologies 45](#_Toc191767580)

[6 proof-of-concept 46](#_Toc191767581)

[6.1 Founding the backend project 46](#_Toc191767582)

[6.2 Migrating a feature 48](#_Toc191767583)

[6.3 Web Frontend 55](#_Toc191767584)

[6.4 Integrating to legacy 58](#_Toc191767585)

[7 roadmap 59](#_Toc191767586)

[7.1 Risk analysis 59](#_Toc191767587)

[References 60](#_Toc191767588)

[Appendices 65](#_Toc191767589)

**LIST OF FIGURES AND Code Snippets**

[**Figure 1** Structure of the thesis 11](#_Toc191245789)

[**Figure 2** Technical debt interest is paid repeatedly until debt is corrected. Debt can be profitable, until tipping point is reached. 15](#_Toc191245790)

[**Figure 3** Continuous integration workflow 22](#_Toc191245791)

[**Figure 4** Unit test result window 23](#_Toc191245792)

[**Figure 5** Test-driven development cycle 24](#_Toc191245793)

[**Figure 6** Refractoring for code isolation 25](#_Toc191245794)

[**Figure 7** Monolithic architecture 28](#_Toc191245795)

[**Figure 8** Architecture with separated front-end and back-end 29](#_Toc191245796)

[**Figure 9** Service oriented architecture 30](#_Toc191245797)

[**Figure 10** Communication between legacy and modernized during transition 31](#_Toc191245798)

[**Figure 11** Modernization over time using strangler fig pattern 32](#_Toc191245799)

[**Figure 12** 3-tier architecture 36](#_Toc191245800)

[**Figure 13** Proof-of-concept communication 42](#_Toc191245801)

[**Figure 14** Backend initial setup 43](#_Toc191245802)

[**Figure 15** Initial project structure including boiler plate classes 44](#_Toc191245803)

[**Figure 16** Bookkeeping accounts maintenance -form 45](#_Toc191245804)

[**Figure 17** Database view of AccountTable 47](#_Toc191245805)

[**Figure 18** Succesfull test detail summary 48](#_Toc191245806)

[**Figure 19** Trying the application with Swagger 49](#_Toc191245807)

[**Figure 20** Server response 49](#_Toc191245808)

[**Figure** 21 Issue discovered during code scan 50](#_Toc191245809)

[**Figure 22** Update API using PUT 50](#_Toc191245810)

[**Figure 23** Navigable side panel 52](#_Toc191245811)

[**Figure 24** Web-based UI 53](#_Toc191245812)

[**Code Snippet 1** Simple test to verify sales basket total is added correctly 23](#_Toc191245813)

[**Code Snippet 2** REST API implementation 46](#_Toc191245814)

[**Code Snippet 3** Unit test for account table validation 47](#_Toc191245815)

[**Code Snippet 4** Test data 48](#_Toc191245816)

[**Code Snippet 5** Defining POST request 53](#_Toc191245817)

# 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 what kinds of issues can occur when sticking with legacy codebases. The focus is on a 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 that VB6 fails to deliver.

The goal of this thesis is to create a plan forward for the target company’s codebase. Literature review will be undertaken to gain information about modern practices, migrating codebase to a different language, and modern software architectures.

## Background

Winpos released its first POS system in the early 1990s. In the beginning the POS machines were large and the software within was concise. Since then, Winpos has grown to be one of the leading POS system providers in the Nordics. Winpos offers solutions to a varied customer base, such as restaurants, ferries, theme parks, and several more sectors, all requiring their own set of features. One of Winpos’s strengths has been the ability to extend the software to serve new clientele.

Over the years Winpos has needed to make technological leaps in order to provide modern solutions. Today Winpos provides highly scalable systems for large restaurant- and ferry chains, mobile applications, and cloud-based solutions. Even though core codebase has been rewritten multiple times to accommodate changing times, developing modern concepts often results in more work needed than it should.

For this thesis, an interview was conducted with the sales- and marketing teams at Winpos. The aim was to identify business needs, potential pain points, and avenues for further growth. Additionally, the technical team was consulted on several occasions. The information gathered was used to design a suggestion for a new architecture for the Winpos software.

## Research methods

The research methods used in this thesis are generally qualitative. Insight is sought from Winpos marketing, using a questionnaire. Information is gathered from books related to the thesis’s subject. In addition to books, blog posts from industry professionals are a major source of information.

Software architecture is very contextual, and decisions made always reflect on the environment. The project for making changes in architecture and codebase will likely take even years to complete. In an agile and iterative process, changes can be made during the project. It is best to apply the best practices and latest methods as a starting point.

This thesis will include a proof-of-concept on how to separate a section of the main program. This separated section will implement many of the methods and strategies discussed in the thesis. To guide the work, following research questions will be answered:

1. How to migrate from legacy software into modern software?
2. What can be done to ensure codebase remains maintainable?
3. What is the business impact of a legacy codebase, and conversely what kinds of benefits could a modern software architecture bring?

## Structure of the thesis

This thesis consists of eight chapters. The introduction-chapter aims to define the focus of the thesis, give an overview of the target company and its needs, and present research methods. Chapter two highlights some of the difficulties of VB6 and what kind of business impact it can entail. In the third chapter, some reasons why businesses get stuck using an outdated system are presented. The fourth chapter evaluates strengths and weaknesses of common architecture patterns, introduces ways to maintain a healthy codebase, and discusses how a software migration could be executed. In the fifth chapter, a practical way forward is established, including choosing the architecture, technologies, and methodologies. The sixth chapter is about making a proof-of-concept using the choices in the previous chapter. Chapter seven displays a road map from beginning of the software migration process to a final modernized codebase. The final chapter is reserved for reflection.

Kuva, joka sisältää kohteen teksti, kuvakaappaus, Fontti

Kuvaus luotu automaattisesti

**Figure 1** Structure of the thesis

# LegaCy woes

The second chapter of this thesis digs into some clear issues and risk factors staying with Visual Basic 6, and legacy systems in general, faces. This chapter will cover both technical and financial challenges that need to be addressed.

## 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. For example, multi-threading is not supported by VB6. Multi-threading is an important feature to fully make use of a computer’s resources, or to build listeners that launch tasks as they get requested. In software development, nothing is impossible, but poor starting point increases the costs.

### 32-bit environment

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)

### IDE

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.

### 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 within 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)

## Technical debt

Technical debt is friction in software development. It is often accrued by making quick changes at the expense of future development being more costly or even impossible. Technical debt is not about defects in the system, but how maintainable and evolvable the system is. Technical debt mounts over time and may have severe consequences. As such, managing it should be a core part of software development. Technical debt is a business level problem that needs to be understood. Scheduling pressures and product pivots are the drivers of technical debt accumulation. Technical debt presents itself in the code, and the responsibility of the developer is to recognize, make a record of and manage it. Typical symptoms of technical debt are developers spending too much time fixing defects in the code, development time of new features being longer, incompatibilities between different systems, having to fix the same defect multiple times, having to make major workarounds when upgrading technology, and cumbersome and lengthy manual testing phases before each release. (Kruchen, Nord, Ozkaya)

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

Tekoälyn generoima sisältö voi olla virheellistä.

**Figure 2** Technical debt interest is paid repeatedly until debt is corrected. Debt can be profitable, until tipping point is reached.

Technical debt is accrued wholistically in software. it can be found in the architecture and structure of the system. This is usually the most hindering for development, as the system becomes rigid for new functionality, scaling and other extensions. Architectural technical debt is also a symptom of a successful company whose scope increases, and the original architecture no longer serves its purpose. Technical debt is found in related software such as deployment infrastructure. Time also plays a role in technical debt as technical gap between the system and modern solutions grows too large. At the source code level technical debt is the most visible. Tooling and inspection make it possible to identify debt as it is incorporated into the system. (Kruchen, Nord, Ozkaya)

Case studies show that there is no silver bullet or one-size-fits-all solution for managing technical debt. One company may have limited resources for regression testing and the overall development methods are fast and loose. Another company may have locked in architectural patterns. In the next, company aging technology stack and high turnover may be the source of technical debt. The first step to paying off technical debt is recognizing where and what it is. Ideally all technical debt should be back-logged as it incurs. Back-logging requires writing down what kind of technical debt has been created, where it resides, and crucially the consequences of leaving it unpaid. Having informative descriptions allows for all parties from management and product owners to developers and testers to be aware of these largely otherwise invisible issues. Technical debt can then be treated as any other software development need and added to development roadmap. The task of a developer is to write high quality code, assisted by static code analysers and code reviewed by peers to not implement any unintended technical debt. (Kruchen, Nord, Ozkaya)

Deciding whether to pay up technical debt or not, is an economical one. Paying the proverbial interest, happens upon further development reduced business. If no further interest payments are expected, paying a large sum in money and effort may not be worthwhile. If debt is in key areas of the product, interest payments may happen at every step of every new feature, and in great cost to business opportunity. In such a case, the benefit of remedying the code will be great, however the cost may also be substantial. From a business perspective, technical debt can also be used to gauge market interest. A quick and dirty application to see if a new business approach is viable is much faster and cheaper than a well architected solution. If market interest was not found, little development time was lost. (Kruchen, Nord, Ozkaya)

To identify technical debt in the system, a technical debt credit check can be performed. Credit check is an interview performer by management interviewing developers and architects. It is a discussion focusing on business vision, architecture, development, and organization about common causes for technical debt. The consolidated list of issues is presented to stakeholders and decision makers. Together, a ranked list based on severity is created. This ranked list serves as a foundation for remediation of technical debt. (Kruchen, Nord, Ozkaya)

## Forced to upgrade

There are several reasons why companies may want to start a modernization project. It is important to identify which needs the business has and work accordingly to remedy the issue. It is important to realize that a modernization project cannot be just about choosing new technologies. Leaving everything else the same eventually leads to the same outcomes.

Developing legacy systems may be costly. Developing minor business needs may end up costing a lot of money. This can lead to situations where small changes are not worth doing on their own. Instead, small changes are lumped together with larger projects. This complicates larger projects further, increasing their risk and cost. (Cartwright, Horn, Lewis)

Mounting costs and difficulties in supporting old systems can be a reason for modernization. System built on older technology may not be problematic because it’s older, it’s a problem when it’s expensive or out of support. Conversely, adoption of newer technologies is not beneficial on its own. Offering competitive advantage compared to competitors, keeping up with competitors, lower running costs, or developmental benefits, such as speed and cost, can be worthwhile reasons for modernization. External pressure from competitors’ ability to offer modern systems can be a crisis for a stale company. Another source of external pressure can come from regulators. (Cartwright, Horn, Lewis)

Modernization in these situations is best done early before it reaches a crisis point. When a crisis point is reached, modernization efforts need to be low hanging fruits and need to cause as little disruption as possible. (Cartwright, Horn, Lewis)

# 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. This chapter will delve into reasons why companies find it difficult to upgrade architectural designs and some pitfalls to avoid when deciding to upgrade.

## 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.)

## 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. There are two ways to modernize software; big bang development, in which software is rewritten entirely, and incremental which rebuilds smaller pieces at a time.

One of the drawbacks of big bang development is that 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 still need to be implemented in the old system. This creates the need to fix the same bugs and develop the same features on 2 systems: the production system and the modernized 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)

If incremental approach is chosen, instead of fully rewriting a system in one go, the project can 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)

Big bang development is often seen as completing faster than incremental approach. This however can be deceiving as first user feedback comes only once the project is finished and in production. Using incremental approach feedback is received constantly throughout the migration process as the software is in customer use from the first finished part. Big bang development is also simpler to visualize, as you only go from a working application to another application. Using the incremental approach, releases happen as new parts finish. Each release needs to be planned and evaluated. (Richardson)

## Migration case study

A case study by De Marco, Iancu, and Asinofsky presents a software migration project where a codebase written in COBOL was migrated to Java. The project also changed the operating system from mainframe to Linux and included a data migration component. The project aimed to modernize the system to increase interoperability between newer systems and reduce costs. The paper details the process needed to test functional equivalence, testing challenges faced, and lessons learned during and after the project. The paper reveals that a migration and a full rewrite of source code was attempted a few years earlier. The initial project failed despite being in development for three years. (De Marco, Iancu, Asinofsky)

The case study team began by comparing migrating options. They ended up choosing to translate the COBOL code using a 3rd party translation tool. The tool translated code was deemed functional but more costly to maintain than a rewritten application. The code also retained COBOL idioms and mainframe concepts that may be unfamiliar to Java developers. On the upside, code translation approach was estimated to cost far less than a rewrite, and the timeline was estimated to be much faster. As the migration progressed, support utilities and services were needed to interface with the existing infrastructure. New components were also developed to maintain functionality and make use of modern tooling. Parts of the code translation also failed and needed to be reimplemented. (De Marco, Iancu, Asinofsky)

Testing the modernized application had the goal of achieving functional equivalence with the legacy application. Test cases were first built at the component level, building up to the whole application. The new and old systems were tested side-by-side. Testing components was done through existing interfaces, such as database tables, SOAP services, and files. When the outputs of the two systems were equal, given the same inputs, the test were considered successful. Testing the full system was done in a private data center. Development, staging and production – environments were set up. Development environment was used to test static data, and staging environment was used to test current data that was transferred from the legacy system. The testing was highly automated and tested the infrastructure performance in addition to data integrity. (De Marco, Iancu, Asinofsky)

The project ended up taking twice the amount of time initially projected. Testing ended up being 70%-80% of the time spent. Valuable lessons were learned during the testing process. The legacy application had poor test coverage, which made it difficult to determine what was the applications intended function. Analysing inputs, outputs, and internal behaviour caused months of unaccounted work. Some of the code in the legacy application was obsolete, which needed removing to save time on translation and testing. Previously unaccounted ways to configure the application were also discovered and the use of insecure protocols both caused more work. The legacy application also was lacked support for significant batch automation, which needed to be implemented. Several issues were discovered that were caused by switching operating systems. Some files supported by the mainframe were not supported by Linux. They needed to be converted to another form, and some new file processing tools needed to be developed. (De Marco, Iancu, Asinofsky)

In the end the migration team states that lacking profound understanding of the legacy application was the primary cause of the delays in delivery. They discovered that improving the initial state of the application would have provided such familiarity that planning, estimates and management could have been handled better. After some time in production, the modernized application has proven to be functional. New feature development however remains challenging and recruiting Java developers has been difficult due to the translated code, which if off putting to many. Additionally, the existing developers are having difficulties with the changed language. (De Marco, Iancu, Asinofsky)

## Economics

Before embarking on a project, it must be established that the potential benefits outweigh the risks.

### Cost-benefit analysis

At its core, cost-benefit analysis, or CBA, is about cataloguing all benefits and costs in monetary terms. Its purpose is to help make decisions by calculating net benefit by deducting the costs from benefits. This gives yields a simple formula Net benefit = Benefits – Costs. CBA can be conducted to evaluate past projects, the maintenance of ongoing processes, and make decisions about future projects. (Boardman et al., 2018)

### Risk analysis

The international standard for risk management, ISO 31000, states that risk management is fundamental to an organizations management at all levels. The goal of risk management is to assist in creating and protecting value, achieve objectives, and help make informed decisions together with stakeholders, or interested parties. By its principles, ISO 31000 promotes structured and comprehensive risk management in all organization’s activities. It also notes that risks are contextual and may emerge, change or disappear based on internal or external factors. Information and lack of information is key at managing risks. Information should be available to all stakeholders in a timely manner. Final principle is that the humane factor has a significant impact on risk management. (ISO)

ISO 31000 is a large framework that promotes leadership responsibility to customize the framework for the organization, ensure application of the framework, delegate risk management to appropriate bodies, and manage resources among other things. The risk management process itself involves systematically applying procedures and activities such as communication, establishing context for the risk, and monitoring, reviewing and reporting of risks. One key component of a risk assessment is conducting risk analysis for individual projects or the like. Risk analysis is used to understand individual risks and their severity. Risk analysis can be comprehensive or a lighter overview depending on the resources and information available. Risk analysis should evaluate factors like likelihood and severity of consequences, complexity, and time-related questions. Multiple methodologies are encouraged to be used for assessing risks related to a project. The standard notes that biases, opinions, and perceptions, as well as information available, assumptions, and exclusions influence the risk analysis and should be taken into consideration when making decisions. (ISO)

# modernizing the architecture

This chapter introduces today’s modern concepts in programming. It takes a look at how codebase can be kept clean, compares different software architectures, and discusses how software migration can be achieved. Choosing technologies for a modern application requires well-reasoned information. The aim of this chapter is to lay a foundation upon which the migration work can rely on.

Software architecture is designed with a purpose in mind, as such it is a reflection of its time. As time passes technology advances and adjustments need to be made to the codebase. Adding to an older codebase creates complexities that might not exist in a more recent codebase. Additionally, the outcomes may not be ideal for modern times. It can be justified to change the underlying architecture during a rewriting process.

## 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

Managing large codebases can be difficult. Technical debt is a measurement of how much bug fixing and refactoring a codebase needs to be considered optimal. Technical debt naturally increases over time as new features are implemented. This section of the thesis explores some of the ways to reduce the amount of technical debt is added over time.

### Continuous inspection

Tools can be integrated into the IDE that flags 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, Tarralappu, Suorakaide, kuvakaappaus

Kuvaus luotu automaattisesti

**Figure 3** Continuous integration workflow

### Testing the code

The most important purpose of testing is to find bugs before they make it into a production environment. In the context of this thesis, tests are snippets of code that test the outcome of a function in code. Several different types of tests are available.

Unit testing is about testing the inputs of a function and verifying their outputs. Unit tests can be set up in a way that tests both regular cases and edge cases. Edge cases are typically more laborious to test in other types of testing. To make effective tests, each part of the code should only address one concern. Unit test’s data can be mock data. Mock data is intended to simulate a real database connection. Mock data is data that uses the classes and functions of a real-life scenario, but the data is engineered for testing purposes. (Schneider) In the case of Winpos, instead of testing whether adding payment and ending the receipt, one should test all individual parts of the process, such as adding payment to the basket, sending a receipt to a receipt control unit, and creating a new task in the kitchen display unit.

Code Snippet 1 and Figure 2 showcase unit testing in a simple way. The code tests another piece of code that adds items to sales basket. Figure 2 shows if the tests were successfully completed. If a programmer makes changes to the sales basket logic and gets the test passed, they should be confident that no logic was broken.

Kuva, joka sisältää kohteen teksti, kuvakaappaus, Fontti

Kuvaus luotu automaattisesti

**Code Snippet 1** Simple test to verify sales basket total is added correctly.

Kuva, joka sisältää kohteen teksti, kuvakaappaus, ohjelmisto, Tietokonekuvake

Kuvaus luotu automaattisesti

**Figure 4** Unit test result window

One way to ensure tests are implemented systematically, and are effective, is the method test-driven development. In TDD a list of tests is written before the actual function is written. A test is then chosen, and the function is implemented until the test is passed successfully. Both the code written and the existing code is then refactored to form a cohesive unit. (Fowler)

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

Kuvaus luotu automaattisesti

**Figure 5** Test-driven development cycle

### Single-responsibility principle and code isolation

Single-responsibility principle is about reducing the effect changes into software have on the codebase. SRP advocates that a class should only have one responsibility, a function, and only one reason for changing it. The benefits of SRP are ease of implementation and help with preventing unexpected results caused by changes. Single-responsibility principle can make the codebase easier to understand. Easily understandable code can help reduce bugs, makes development faster, and improves the developer experience. (Thorben)

Single-responsibility principle can be used as a guide rather than a strict rule. Too many classes can create a feeling where reading the code becomes more difficult as the developer can only see a small part of the code at once. Defining what a responsibility is difficult and subjective.

Breaking code into smaller pieces improves the testability of software. Some types of functions are easier to test using unit testing. By isolating unit testable code from a larger function, unit tests can be made more easily. This also creates a clear boundary between unite testable code and code that needs to be integration tested. (Wagner)

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

Kuvaus luotu automaattisesti

**Figure 6** Refractoring for code isolation

### Code review

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)

### Measuring maintainability

Several models exist for measuring software maintainability. One widely adopted model is the ISO/IEC 25010:2023 -standard. It defines nine categories pertaining to software product quality. One category is software maintainability, which is further divided into five subcategories.

1. Modularity, which measures how modular a program is and how much modules interfere one another.
2. Reusability, which measures how well a program can be used in multiple systems or as a building block for one.
3. Analysability, which measures how well changes to a system can be measured.
4. Modifiability, which measures how well a system can be modified efficiently without causing defects.
5. Testability, which measures how effectively test criteria can be made and how effectively the criteria can be tested.

Subcategories provide a focal point for observation. Exact ways of measuring and acceptable values are more open for interpretation. (ISO).

ISO/IEC 25010:2023 – standard is a part or a series of standards, known as SQuaRE. Case study by Mena and Santorum used metrics and equations defined in ISO/IEC 25022:2016 to evaluate software maintainability. Equations such as modiﬁcation complexity X =A/T, where A is the number of modifications, and B is time worked on said modifications, can help identify problem areas if X is too small. Some metrics are more related to code quality such as cyclomatic complexity X=A+1, in which A is the number of conditional statements in a function. In the case of cyclomatic complexity, X should be as close to 1 as possible (Mena & Santorum).

Maintainability index is another, rather simpler, way to measure software maintainability. After it was adopted and altered by Microsoft, the maintainability index measures five values in software code and gives a maintainability index score between 0-100. It measures cyclomatic complexity, depth of inheritance, class coupling, lines of source code and based on these values, calculates the maintainability index. Maintainability index can be calculated automatically in some IDEs, such as Visual Studio. This automatization makes it an appealing option for it to be used as one component of measuring maintainability. (Microsoft)

## Software architectures

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 frontend and backend 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 the 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 7** Monolithic architecture

### Separated frontend and backend

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)

This approach could be the best middle ground solution for the target company. POS systems require several different user interfaces for mobile, browser, and local installations. One of the recent challenges the target company has had is that the customers expect to be able to access their POS system anywhere. This style of architecture supports running the backend as a service and accessing it through a browser in a customer’s home, for example.

Backend running as a service with APIs used for communication supports all types of UIs without the need for modification. Local installations would suffer very little as running the frontend as well as the backend on the same computer offline is still an option.

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

Kuvaus luotu automaattisesti

**Figure 8** Architecture with separated frontend and backend

### Service-oriented architecture and micro services

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)

Service-oriented architecture is the best choice for many applications deployed in a cloud platform. Software split into multiple services can be cost optimized better than a backend that is built out of one part. Offering software as a service in a cloud platform, however, is about much more than mere software. SaaS model is a decision that affects the way a business operates, and the revenue streams would be significantly different. Multiple services would also want to make use of multiple databases. Other options listed before make use of a single database. Generally, developers should avoid using the microservice pattern, unless they have a really good reason to. Good reasons include things like having more than a million users, or the system is too large and complex to be a monolith, both in a cloud environment. Neither case applies so the use of microservices is not advised.

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

Kuvaus luotu automaattisesti

**Figure 9** Service oriented architecture

## Migrating the architecture

Large-scale modernization projects are complex and in many cases should not be attempted in a single project. Instead, modernization effort should be considered a process that takes place over a long period of time.

### Strangler fig pattern

Strangler fig pattern is an idea in which a portion of a software is “strangled” and migrated to a new architecture. This act reduces the size of the legacy codebase, while the modernized codebase grows. The pattern is repeated until the entire software is migrated. Both new and legacy software will be in use until the migration is fully completed. This will require a mechanism to allow communication to be directed to where it is needed. (Microsoft)

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

Kuvaus luotu automaattisesti

**Figure 10** Communication between legacy and modernized during transition

Strangler fig pattern is a pattern that helps minimize the risks involved in legacy modernization projects. Timeline of the modernization process can be extended without negative business impact. The legacy software and modernized parts can exist side-by-side, routing more and more traffic to the modern platform as it builds. One goal of the strangler fig pattern is that the migration process is not visible to the end user. (Microsoft)

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

Kuvaus luotu automaattisesti

**Figure 11** Modernization over time using strangler fig pattern

### Transitional architecture

During the migration process, legacy applications will need to work together with the new applications. For this to be possible, transitional architecture will need to be in place to enable the communication between the applications. The key feature of transitional architecture is that it is temporary and only serves to enable the migration process.

What transitional architecture offers, is speed of delivery. New features can be developed for the new platform, while it is still in an unfinished state. New features can include an interpreter for the legacy system that is designed for removal once it’s no longer needed. Another benefit of the transitional architecture is further derisking of the migration.

### Targets for displacement

Identifying which code can be migrated is often called finding seams or introducing seams into the system. These seams can be used to direct the flow to the new system without altering the old system. Seams were originally intended to make the legacy system more testable, but their most valuable use was found in legacy migration. Seams are a great way to gradually migrate behaviour away from the legacy system, by decoupling functionality and migrating it partially to a new system. Even without modernization of the codebase, introducing seams make the old system more testable. (Fowler)

Function calls are common locations of seams. A function can often be migrated to a new system without modifying the original code that calls for it. Issues arise when seams are not present, and a code block’s logic is too large to be migrated. While migrating the code as is, is possible, benefits of migration will suffer when improvements are not made.

## Web service architectures

Modern applications often rely on communication over the internet. The method in which communication is established defines the structure, and protocols of the application. Several approaches have been developed that each come with their own strengths.

### Representational state transfer

Representational state transfer, also known as REST, is a stateless way of achieving server-client communication. Like many strategies, REST allows client-server communication where the two systems don’t need to know each other’s state. Clients call endpoints of individual resources it needs, and the server will fulfil the client's request. (Codeacademy)

In REST, web requests are sent to a specific URL. The URL includes where the server lives and any information it needs to handle the request. Requests are made using HTTP verbs; special keywords that describe the intention of the call, such as GET, and POST. Additionally, headers are used to make the request more accurate. Headers may include pointers like content type that the client accepts as a response. When the request is properly formed and handled, the server will response with a code indicating whether the request was completed successfully or not, and the contents of the response. (Codeacademy)

### GraphQL

GraphQL was initially develop to remedy some of the shortcomings of REST. It has been gaining popularity since its inception in 2012. Its benefits are best realized as a part of a complex system where data is dispersed. Additionally, its benefits become greater as traffic increases. Overall, the implementation is more complex than the one of REST. (Kong)

The main idea behind GraphQL is that it retrieves the exact information a client requests. In REST there is no way for a client to limit what data an API returns, the client may have to make multiple requests for data to get the dataset it needs, in GraphQL one request is sufficient. This results an efficient system as traffic over the internet is reduced, amount of API calls is reduced, and the server workload is reduced. (Kong)

## 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.

# Choosing technologies

This chapter introduces all the technologies that will be used to make a proof-of-concept about the modernized system. The technologies are intended to reflect upon what a true production system would look like. The decisions were made based on three main points; theory laid out in the thesis, integrability at Winpos, and feedback received from the business and technical departments at Winpos.

## Architecture

When designing software architecture, business need defines the product. At Winpos the system has moulded to its current form as a result of business needs and opportunities. Still the current state makes implementing some modern concepts more difficult than they should be.

Winpos POS system is a large software product with numerous components that need to be supported. The most basic use case for customers is locally making sales using a POS pc, and being able to manage products, run reports, and perform other general maintenance. Additionally, customers would like to be able to do maintenance from home, make sales using mobile devices, and general functionality available in an offline environment. While all the aforementioned is possible it is not without significant development overhead.

What this thesis proposes is conversion of the existing structure to a 3-tier architecture. 3-tier architecture consists of web-, business logic-, and data-tiers. Each tier is physically separated, albeit there is nothing stopping one from running them on a single machine. 3-tier architecture benefits over a lesser tier architecture include faster development, improved reliability, scalability, and security. (IBM)

In figure 11, parts of the POS system have been divided into tiers and layers. Layers represent a logical component of the application. In the presentation layer, a shop manager can, for example, connect through a web service to the Backoffice application to edit product information. Firewalls are present to deny access from unauthorized traffic to the internal network. Changes made to products are forwarded to the Business logic layer. Business logic layer contains the backend software in one or more individual applications. Data is formatted and saved in the data layer, which houses databases.

Kuva, joka sisältää kohteen teksti, Tarralappu, diagrammi, Suunnitelma

Kuvaus luotu automaattisesti

**Figure 12** 3-tier architecture

### Backend

Winpos POS system today is primarily written using two languages. The core components are written in Visual Basic 6. Significant portion of integrations and other components are written in C#. These lay a foundation where either language can be used. Two options rise above the rest for modernizing the Winpos POS system: adopting Twinbasic or migrating fully to C#.

Twinbasic is a new basic language that aims to be fully backwards compatible with VB6 projects. Twinbasic aims to implement several features that VB6 is missing, such as a modern IDE, 64-bit support, and multi-threading. Twinbasic is still in development, with version 1 anticipated in 2024. (Twinbasic)

Twinbasic is an option because of the backwards compatibility with VB6. In a perfect scenario, upgrading to a modern language and IDE would be a simple conversion of the existing codebase. Twinbasics small development team and its early release status brings some risks as the main development language. Some features might not be fully functional, and it might lack wide third-party support tools and libraries.

Another option would be to use Microsoft’s Visual Studio IDE and C#. Visual Studio is currently in use at Winpos, as several applications have been written in C#. There is no reliable way to translate VB6 code C# code. To maintain full functionality code must be rewritten to the new language. Any rewrite-project is bound to be resource intensive and time consuming.

Effort that is required to fully migrate to C# will likely be greater. Despite this, rewriting to C# would likely be a better choice due to its potential to yield superior results. Visual Studio has garnered a well-established, robust ecosystem. Most modern tools and libraries are readily available to be integrated into Visual Studio. For Winpos specifically C# is the superior choice as there is little learning curve for new tools and environments. Additionally, for a developer, a backend written in one language makes it easier to switch between different projects.

### Frontend

Backend built using restful APIs supports any number of frontends. The focus of this thesis is on the core shop interface. In a shop environment, a browser-based user interface is suitable, as the same interface can be used remotely over the internet. Crucially, in the interim period, the existing user interface needs to be connected to the new backend. In the proof-of-concept section of this thesis, connectivity of a browser-based GUI and the connectivity of existing GUI is presented.

Web-based interfaces at Winpos today are written in Angular. Angular is a modern and widely adopted framework of TypeScript. It is designed to make building user-interfaces both over the web and natively. Due to strong existing knowledge, angular is a good choice to move forward with. Another option could be to use Blazor. Blazor is developed by Microsoft and utilizes the .NET Core framework. A shared codebase with the backend application could be seen as a benefit.

## Language and IDE

Changing the architecture to a 3-tier system requires that the existing software needs to be split into front- and backend applications. Each side needs their own developer tools. Learning curve for developers was one of the most important factors when choosing languages, as long as the language was able to provide the necessary features.

### Language

For proof-of-concept, C# will be used to create the backend. C# was chosen due to its history withing Winpos, long-term support, and proven track record. Another factor why C# was selected is its Visual Studio integrated development environment, or IDE. Visual Studio offers easy access to many of the topics discussed in this thesis, such as support for unit testing, and integrated continuous inspection tools.

For proof-of-concept, Angular will be used due to its suitability for the job and history within Winpos. Angular is more flexible as to which tools a developer would like to use. One good option is to use Visual Studio Code, or a text editor such as VIM or NotePad++.

### Project type

ASP.NET Core is an extension to the .NET Core platform. It is specifically intended to be used for building web applications.

### Unit test

Choosing the right unit testing framework is a decision that will have an impact on the software for years. Lots of code will need to be written into the unit test project so choosing the correct project type is impactful. Today there are three popular unit testing frameworks for C# and .NET; MSTest, xUnit, and nUnit. Although they are all unit testing frameworks, they all have their strengths over one another.

The main benefits of MSTest are its easy integration to Visual Studio environment, support from Microsoft by regularly offering updates, and support for data-driven tests. Main drawbacks are its lack of extensibility options, and slowness in large projects. NUnit is highly customizable for an organization’s specific needs. Test execution can be customized, and custom extensions can be self-written. XUnit is the most performant option and has an active open-source community for updates. XUnit is also the most extensible option using plugins and other extensions. (Ghinaiya)

For proof-of-concept, xUnit will be used to create tests. Winpos POS system is a large project with many functions that need to be tested. Fast tests, combined with ready extensions for many needs, and are compelling reasons to choose xUnit over the other options. It also synergises well with integration testing applications.

## Technologies

When choosing technologies, longevity, easy integrability to existing systems, and external systems, and time to market were of the highest importance.

### API strategy

The backend will be written as REST APIs. Compared to alternatives, REST APIs offer an easier learning curve and a proven track record. REST APIs are endpoints and resources that can be called from over the internet. These endpoints and resources are intended to allow easy information swapping between different systems.

Today, a lot of developer’s time is spent making integrations to other systems. Winpos has integrations to many services in different financial sector operators. Integrating other systems to Winpos requires a significant effort. Easy integrability may bring new business opportunities. This is where REST APIs can shine. REST APIs allow easy integrability for both in-house frontends such as, browser and mobile, as well as external integrations.

### Continuous inspection

Code quality analysis is best achieved by using an integrated extension in an IDE, and automated inspection server in conjunction with other CI/DC tools. A popular continuous inspection tool is SonarQube. SonarQube can be run, as an automated inspection tool, on its own server that analyses code as it is committed to version control. Analytics can also be directly integrated to an IDE so the developer can self-monitor their work.

One of the key principles using SonarQube is the principle of clean as you code. Clean as you go refers to writing new code, and altering existing related code, that is verified to be up to standards. Automated tooling uses quality gates that ensure that code that does not pass certain quality criteria cannot be merged to the larger codebase. Quality gates can be configured by the organization with minimum passing quality.

For proof-of-concept, the integrated components of SonarQube will be used. The generic settings are a good starting point as it will report about poor code, unsafe methods and keep track of consistent naming of variables.

## Methodologies

For the longevity of any codebase, it is vital to adopt practises that promote well written code. Starting a new project is an opportune time to apply methodologies and tools that help ensure the code is written well and stays high quality throughout the development process.

In a rewriting project, the end result of the new code is particularly well known. This makes it a good opportunity to practise test-driven-development.

# proof-of-concept

This chapter presents a proof-of-concept way of splicing a piece of code from the legacy monolith and migrating it to an API-based backend. Connectivity to the backend is demonstrated from both a new Angular.js-based browser frontend and how to use the new API from the existing solution during the transitioning period. Many of the topics discussed in previous chapters are also presented in practise, such as test-driven-development, creating the project files and must have infrastructure, as well as IDE-based code quality validation. The end result is a backend server that can be reached through the existing VB6-based executable, though a middleware app, and a web browser (Figure 12).

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

Tekoälyn generoima sisältö voi olla virheellistä.

**Figure 13** Proof-of-concept communication

## Founding the backend project

A solution needs to be created that will house two projects. One project contains the new backend code, and the other contains unit tests for the code.

The backend project type is ASP.NET Core Web API. When creating such a project, several options are given. In this PoC, the latest framework, .NET 8.0, is selected. OpenAPI support is enabled to leverage Swagger in development. Swagger is designed to simplify API development by allowing easy testing of APIs without a ready UI. Another option that is enabled is the support for containers. Containers are virtual operating systems that run the application being developed. The main benefits of using containers are that the program will run on the same operating system regardless of where it is installed, resource efficiency when deployed in a cloud environment, and security by isolation.

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

Kuvaus luotu automaattisesti

**Figure 14** Backend initial setup

Secondly project with type xUnit Test Project is added to the same solution. This project will contain all tests needed for the backend application. The backend project is added as a dependency to the unit test project, so its code is visible to the unit test project.

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

Kuvaus luotu automaattisesti

**Figure 15** Initial project structure including boiler plate classes

## Migrating a feature

Migrating a feature can be seen as a three-step process. Firstly, a feature that can be migrated is selected. Secondly, code is created in the new software, and thirdly, old code is removed, and the flow of information is directed to the new software. For the purposes of this thesis, a smaller feature is selected, and parts of its basic functionality is migrated.

Bookkeeping accounts are managed in a form called “Bookkeeping accounts maintenance” of the VB6 codebase (Figure 14). The legacy codebase has a rich set of features and functions that enable meticulous managing. The goal of this proof-of-concept is to migrate the addition and updating of a bookkeeping account in Winpos database.



**Figure 16** Bookkeeping accounts maintenance -form

To add a new record into the bookkeeping account table it is beneficial to divide the task into steps. The backend needs to have:

1. A REST API to receive information from the frontend
2. Capability to deserialize the information into a manageable class
3. Validation of received data
4. Unit tests for validation
5. Ability to store new information into a database
6. Capability to send a response to frontend

The REST API implementation begins with selecting a http method and route. Http method POST is used for addition. Using the appropriate http methods allows for the same route to be used multiple times as each method is handled differently.

Next, the function that is executed upon request is defined. IActionResult is defined as the return type, which allows responding with descriptive response types and messages back to the frontend that made the request. Response types and messages offer insight to the calling frontend application about whether or not the request was correctly formed. If the message went through successfully, response type 200 and a message including the new record is sent back. If the message was not properly formed, a message including information about how to rectify issue is sent back. The body of the API calls for the data to be validated and calls for a new bookkeeping account -record to be created into the database.  
Kuva, joka sisältää kohteen teksti, kuvakaappaus, Fontti

Kuvaus luotu automaattisesti

**Code Snippet 2** REST API implementation

Validating the account table data is a part of business logic. Business logic is best placed in an easily testable function. Unit testing business logic functions reduces the change of accidentally changing the way the system should work. Test driven development advocates for tests to be created before the actual implementation of a function. The tests created will include test cases for scenarios where validation must be rejected, successful, and edge cases that need to be successful.

A new unit test class is added to the testing project. Implementing a unit test begins by introducing which type of test it is. In this case, Theory-type is used. Theory signifies that the test uses predefined data for testing. The test data is introduced and generated before the test is run. The function that is being tested is executed against the test data.

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

Kuvaus luotu automaattisesti

**Code Snippet 3** Unit test for account table validation

Database fields can be used to define restrictions and structure that the input data should have. Database fields give insight both for the creation of an account table -class and unit testing the data. From the database it is observable what kinds of variable names there are and their data type, size, and whether it is mandatory (Figure 15).

Kuva, joka sisältää kohteen teksti, Fontti, viiva, numero

Kuvaus luotu automaattisesti

**Figure 17** Database view of AccountTable

Test data is created as new objects. These objects imitate real world data and include a value that the validation function should return for each data node. Objects created represent some edge cases such as maximum length values for description-, and code-fields. They also include objects that have invalid codes or dates. The tests are designed to fail until the function that is being tested is correctly implemented.

Kuva, joka sisältää kohteen teksti, kuvakaappaus, Fontti

Kuvaus luotu automaattisesti

**Code Snippet 4** Test data

With the restrictions and requirements written into a test, the function becomes clearer to write. The function becomes more robust as what it needs to accomplish is tested every time tests are run. This reduces the chance of accidentally making changes that break business logic of other features. To verify functionality, tests can be run in the IDE (Figure 16).

Kuva, joka sisältää kohteen teksti, kuvakaappaus, Fontti

Kuvaus luotu automaattisesti

**Figure 18** Succesfull test detail summary

When the API is fully implemented, it can be executed by running the application. Because OpenAPI support was selected upon project creation, the project comes with a built in Swagger functionality. Swagger is a software development tool that allows for sending web requests to the application. Swagger automatically generates a request body that matches what is defined in the receiving API. The request body can be modified to test real world data or edge cases (Figure 17).

Kuva, joka sisältää kohteen teksti, kuvakaappaus, ohjelmisto, näyttö

Kuvaus luotu automaattisesti

**Figure 19** Trying the application with Swagger

Pressing execute will send the request to the running backend web application. The web application processes the request and successfully sends back an appropriate response (Figure 18). In this case the response includes the code 200, which indicates a successful operation, and a success response defined in the API.

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

Kuvaus luotu automaattisesti

**Figure 20** Server response

The code for the API is now nearly complete. To verify that the code is high quality, a tool that scans the code for bugs and vulnerabilities needs to be applied. One such tool is the SonarAnalyzer.CSharp, which can be installed via the nuget package manager in Visual Studio. The tool highlights errors and gives suggestions how to fix the error, it also has links for each error it finds that gives detailed descriptions of why the code might need to be changed (Figure 19). Scanning the code revealed an issue with the API code needed to be changed. Ideally code would be scanned and rejected upon merging into the codebase. Scanning while merging would enforce rules set on the codebase automatically.

Kuva, joka sisältää kohteen teksti, Fontti, viiva, numero

Kuvaus luotu automaattisesti

**Figure** 21 Issue discovered during code scan

Implementing an update method can be done using the same endpoint. If the same endpoint is used, using a different HTTP verb can be used to call a different function in the backend. HTTP verb PUT is often used to update fields. This allows cohesive naming patterns for easier use on the frontend side. The implementation is similar to the POST method previously (Figure 20). HTTP verb is changed, and minor changes are made to the function for updating records instead of adding them.



**Figure 22** Update API using PUT

This completes the implementation of the backend functionality. In a business environment the code would now be ready for code review by peers. Review by peers ensures there is less chance of producing code that does not align with business practises and bugs may be discovered during the review process that need to be fixed.

## Web Frontend

Frontends are interfaces for users to interact with the underlying system. In the case of this thesis the system to be interacted with is the backend software. The backend is used to save and update data in the system. For the user to be able to trigger the events, the frontend must be able to perform several tasks.

1. The user must be able to access the frontend application through a web browser, such as Google Chrome
2. The frontend application must be able to display different forms
3. The user needs to be able to navigate to the form they require
4. The frontend application must be able to send and receive information to and from the backend.
5. The frontend application needs to provide feedback to the user from the backend when operations complete or fail

User interface- and user experience designing is an important part of developing any frontend. Their purpose is to make the app visually pleasing and easy to use, among other things. Frontends are vast ensembles on their own and take time to build from the ground up. Instead, a ready-made template is chosen and modified to fit the purpose of this PoC. One template that has all the features needed at this point in development is the Devextreme angular template.

Devextreme angular template is one of many browser-based templates available online. The template comes in downloadable angular project that can be modified to ones needs. For this thesis, the project was extracted, and its dependencies were installed using node.js commands. The project was then opened using Visual Studio Code- code editor. The editor makes editing and running commands easy and allows for faster development. Running the template code makes the website accessible in a local environment.

Using the template, it is simple to setup a UI that is navigable (Figure 21). On the left-hand side are the main categories that enable management, such as maintenance and sales statistics. Each category is a collapsable menu item that opens more nuanced management categories. Clicking a category opens a form that allows editing values and settings for each system.

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

Kuvaus luotu automaattisesti

**Figure 23** Navigable side panel

Under the maintenance main category exists the system bookkeeping account – form, which is being migrated (Figure 21). Each control such as buttons and text areas, need to be created to the web interface. Figure 22 presents a simple example implementation of bookkeeping accounts – form. The form is split into four quadrants, each with their own logical function. The first quadrant displays information about the selected bookkeeping account. Below the bookkeeping account information is a search box of all the bookkeeping accounts in the system. On the right are additional controls and actions. The UI will need to be designed from the ground up before it is ready for production usage.

Kuva, joka sisältää kohteen teksti, kuvakaappaus, ohjelmisto, Tietokonekuvake

Kuvaus luotu automaattisesti

**Figure 24** Web-based UI

To enable communication with the server, code must be written that contacts the backend APIs. HTTP methods need to be defined. An example implementation in Code Sample 5 presents a function that takes the Bookkeeping endpoint, and data to be sent as parameters. This function makes use of basic http utility, and Angulars Observable-type, which is in essence a type of data stream.

Kuva, joka sisältää kohteen teksti, kuvakaappaus, Fontti

Tekoälyn generoima sisältö voi olla virheellistä.

**Code Snippet 5** Defining POST request

When a user has modified an account, pressing the Save-button in the web UI launches the buttons defined function (Code Sippet 6). The function first parses all the required information from the form into a JSON-formatted data set. PUT request is then sent to the server, targeting the appropriate endpoint with the data set as it’s payload. The final responsibility of the function is to handle the response from the server. The response can be that the operation was successful, and the database was updated to reflect the user’s input, or it may fail the validation on the server’s end, or any number of other error responses. The user should be made aware of the end result of their action by for example, displaying a toast or alert on the page.

Kuva, joka sisältää kohteen teksti, kuvakaappaus, ohjelmisto, Multimediaohjelmisto

Kuvaus luotu automaattisesti

**Code Snippet 6** Save button logic

## Integrating to legacy

# roadmap

## Risk analysis

This thesis’s risk analysis adapts ISO 31000 principles of applying several analysis techniques for complex projects. Failure mode and effect - analysis and SWOT-analysis are common analysis methodologies that can be used in software projects. SWOT analysis gives a high-level view of the positives and negatives related to the project. FMEA analysis is a complementary analysis methodology that can be used to identify specific risks and their severities.

# 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 2024-05-09. 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 2024-05-10. https://www.geeksforgeeks.org/difference-32-bit-64-bit-operating-systems/

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

The Classic Visual Basic Programming Language. TIOBE. Retrieved 2024-05-10. https://www.tiobe.com/tiobe-index/classic-visual-basic/

Programming Language Statistics. ITJobsWatch. Retrieved 2024-05-10. https://www.itjobswatch.co.uk/IT-Job-Market/UK/Programming-Languages

VB6 IDE AddIns - What's essential?. VBForums. Retrieved 2024-05-12. <https://www.vbforums.com/showthread.php?804911-VB6-IDE-AddIns-What-s-essential>

Audacia. 2019. The cost of legacy IT. Medium. Retrieved 2024-05-12. 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: 2024-05-12. <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.

Kruchen P, Nord R, Ozkaya I. 2019. Managing Technical Debt: Reducing Friction in Software Development. Addison-Wesley Professional.

Fauerbach, C. 2020. How Do I Choose the Right Programming Language for My Team?. Retrieved 2024-05-13. <https://medium.com/capital-one-tech/lessons-from-the-leadership-trenches-how-do-i-choose-the-right-programming-language-for-my-team-36222aa78b38>

Fowler, M. 11.12.2023. Test Driven Development. Retrieved 25.5.2024. <https://martinfowler.com/bliki/TestDrivenDevelopment.html>

Schneider, J. 24.5.2022. Writing usable code. Retrieved 25.5.2024. https://www.algolia.com/blog/engineering/writing-usable-code/

Wagner, L. 13.7.2023. Mock Your Mocks. Retrieved 9.5.2024. https://blog.boot.dev/clean-code/writing-good-unit-tests-dont-mock-database-connections/

Thorben. 28.2.2024. SOLID Design Principles Explained: The Single Responsibility Principle. Retrieved 9.5.2024. https://stackify.com/solid-design-principles/

What is three-tier architecture?. IBM. Retrieved 22.8.2024. <https://www.ibm.com/topics/three-tier-architecture>

Boardman A, Greenberg D, Vining A, Weimer D. Cost-benefit analysis concepts and practice fifth edition. 2018. Cambridge university press.

Twinbasic. Retrieved 22.8.2024. https://twinbasic.com/

Cartwright I, Horn R, Lewis J. Patterns of Legacy Displacement. 5.3.2024. Retrieved 30.9.2024. https://martinfowler.com/articles/patterns-legacy-displacement/

Microsoft. Strangler Fig pattern. Retrieved 30.9.2024. <https://learn.microsoft.com/en-us/azure/architecture/patterns/strangler-fig>

Cartwright I, Horn R, Lewis J. Transitional architecture. 28.3.2022. Retrieved 30.9.2024. <https://martinfowler.com/articles/patterns-legacy-displacement/transitional-architecture.html>

Fowler, M. Legacy seam. 4.1.2024. Retrieved 6.10.2024. https://martinfowler.com/bliki/LegacySeam.html

Codeacademy. What is REST?. Retrieved 12.10.2024. <https://www.codecademy.com/article/what-is-rest>

Richardson C. STOP hurting yourself by doing big bang modernizations!. Retrieved 20.2.2025. https://microservices.io/post/architecture/2024/06/27/stop-hurting-yourself-by-doing-big-bang-modernizations.html

Kong. What is GraphQL? 1.3.2023. Retrieved 12.10.2024. <https://konghq.com/blog/learning-center/graphql>

De Marco A, Iancu V, Asinofsky I. COBOL to Java and Newspapers Still Get Delivered. 10.8.2018. Retrieved 25.2.2025. https://arxiv.org/pdf/1808.03724

Ghinaiya, S. 19.6.2024. Testgrid. Retrieved 20.10.2024. https://testgrid.io/blog/nunit-vs-xunit-vs-mstest/

ISO. ISO/IEC 25010:2023. 11/2023. Retrieved 17.2.2025. <https://www.iso.org/standard/78176.html>

ISO. ISO/IEC 31000:2018. 2/2018. Retrieved 28.2.2025. https://www.iso.org/standard/65694.html

Microsoft. Code metrics values. 29.11.2023. Retrieved 17.2.2025. <https://learn.microsoft.com/en-us/visualstudio/code-quality/code-metrics-values?view=vs-2022>

Mena D. Santórum M. Maintainability and Portability Evaluation of the React Native Framework Applying the ISO/IEC 25010. 11/2021. Retrieved 17.2.2025. <https://www.researchgate.net/publication/346166139_Maintainability_and_Portability_Evaluation_of_the_React_Native_Framework_Applying_the_ISOIEC_25010>

# 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