Defining and Identifying Legacy Code in Software

A case study developing 3D visual camera surveillance software

Niclas Svensson

Computer Science
Bachelor level, 15hp
2018
Supervisor: Dipak Surie
Sammanfattning

Legacykod är något som är svårt för utvecklare att förhindra. Medan det finns mycket forskning kring att motarbeta legacykod och förnya legacysystem som använder utdaterade teknologier och designer, så har inte mycket fokus lagts på hur legacykod kan identifieras i sitt tidigaste skede. Hur dessutom definierar man legacykod?

I denna studie så görs en litteraturstudie för att ta reda på vad för forskningsbakgrund det finns bakom legacykod i allmänhet. Därefter så utvecklas två prototyper i syfte att upptäcka mönster under utvecklingen som kan vara egenskaper till legacykod. I slutändan så utförs en enkät riktade åt mjukvaruutvecklare för att få insyn och tankar på vad legacykod betyder för dem och för att testa idéerna som togs fram från implementationerna.

Den rekommenderade definitionen för legacykod är kod som inte har testats, ingen eller för lite dokumentation och är allmänt svår att förstå, vare sig med dokumentation eller utan. Observationerna från implementation av de två prototyperna bildade 7 riktlinjer som tar upp hur man kan identifiera uppväxten av legacykod.

Nyckelord

Legacykod, legacysystem, mjukvaruutveckling
Abstract

Legacy code proves to be something difficult for developers to prevent. While much research exists to combat legacy code and renewing legacy systems that use outdated technologies and designs, not much focus has been put on how legacy code can be detected at its earliest stage. And how do you define legacy code?

In this study, a literature review is done to find out the scientific background behind legacy code in general. Afterwards, two implementations of the same software are done to observe any events during the software development that can be characteristics of legacy code. In the end phase, a questionnaire-based survey is handed out to developers to get their insight and thoughts on what legacy code means to them and to test the ideas brought up from the implementations.

A recommended definition for legacy code is code with no testing and little to no documentation and generally being hard to understand, whether with documentation or not. From the observations implementing the prototypes, 7 guidelines are formed to help identify and discover legacy code evolution.

Keywords

Legacy code, legacy system, software engineering
# Table of Contents

1 INTRODUCTION ...................................................................................................................... 8

1.1 LIMITATIONS ................................................................................................................... 8
1.2 LAYOUT OF THE ARTICLE ............................................................................................. 8

2 BACKGROUND ....................................................................................................................... 10

2.1 LEGACY CODE ............................................................................................................... 10
2.2 EXAMPLES OF LEGACY CODE ..................................................................................... 11
2.3 IMPLICATIONS ................................................................................................................ 11

3 LITERATURE REVIEW ......................................................................................................... 12

3.1 PREREQUISITES .............................................................................................................. 12
   3.1.1 Inclusion/exclusion criteria ...................................................................................... 12
   3.1.2 Keywords ............................................................................................................... 12
   3.1.3 Search Results ....................................................................................................... 12
3.2 SURVEY OF RELATED WORKS ...................................................................................... 13

4 METHOD ................................................................................................................................. 16

4.1 LITERATURE REVIEW .................................................................................................... 16
4.2 COMPARATIVE STUDY ..................................................................................................... 16
4.3 QUESTIONNAIRE ............................................................................................................ 17

5 IMPLEMENTATIONS AND COMPARISON ........................................................................... 18

5.1 PROTOTYPE A AND B DESCRIPTION ........................................................................... 18
5.2 PROTOTYPE A .................................................................................................................. 18
   5.2.1 Prototype A Observations ...................................................................................... 18
5.3 PROTOTYPE B .................................................................................................................. 20
   5.3.1 Prototype B Observations ...................................................................................... 20
5.4 DEVELOPMENT OBSERVATIONS AND COMPARISONS .......................................... 21
5.5 SUGGESTIONS FOR IDENTIFYING LEGACY CODE .................................................... 22

6 QUESTIONNAIRE SURVEY .................................................................................................. 24

6.1 STRUCTURE .................................................................................................................... 24
6.2 STATEMENTS ................................................................................................................... 24
6.3 RESULTS .......................................................................................................................... 24

7 DISCUSSION .......................................................................................................................... 26

8 CONCLUSIONS ..................................................................................................................... 30
REFERENCES ........................................................................................................ 31

APPENDIX A ........................................................................................................ 33
Glossary

Refactoring Rewriting parts or the whole software to, for instance, increase performance, readability and reusability while keeping system behavior. Synonyms to refactoring can be reengineering.

Software evolution The process in software maintenance where an application is repeatedly updated.

Developer In this context to mean software developer, a person that writes software.

IP-based camera A camera that can be connected to an Internet network.

Unit testing Testing small parts of a software to verify it produces expected and failsafe results.

Library External collection of files, classes, functions and executables that can be included in other software.

Technical debt Choosing to implement quick and easy solutions instead of hard and lengthy ones, that can cause problems in the future if they are not addressed at a later time creates technical debt.

PTZ Short for Pan Tilt Zoom, functionality that makes a camera’s field of view movable and dynamic.

LOC Short for Line of Code, a metric that can be used to see the code size of a program, module or method.
1 Introduction

Legacy code is an interesting phenomenon in that past written code becomes more and more essential to the foundation of an application to a point where developers – either the author of the written piece of code or other developers – do not dare to touch or replace it in fear that it might break some, if not all, of the application [5]. Legacy code can in return rely on older legacy code or modules which makes something as simple as refactoring a module turn into refactoring dozens. This takes up more effort for the developers and more lost time into resolving the issues that pops up.

Resolving legacy code becomes tedious for developers as proposed solutions like refactoring code or rewriting systems all come with consequences, related or not, even though they would solve the original problem. If further written code has to accommodate the inconvenience and vulnerabilities prior legacy code has, it too can end up being legacy code and the application for instance becomes less flexible to change.

The software industry is heading in a smarter direction in terms of development, employing agile software development models where the development in a project can respond quickly to changing requirements [20] as well as the software design in a project. This allows software to stay ahead and quickly evolve for tomorrow’s needs. But legacy code is still a nuisance in software development and has the power of posing major problems in the future [3]. With the variety of definitions that exist, what should the recommended definition of legacy code be? And how can we determine at what point during development does code become legacy code?

1.1 Limitations

Legacy code can exist in all types of computers and software. This study will only focus on legacy code and legacy systems developed for personal computers since the prevalence of software development is higher among developers than embedded systems development. Legacy systems tailored for mainframe computers, servers and such are therefore not covered by the scope in this study.

1.2 Layout of the Article

Section 2 will describe the scientific background that exists regarding legacy code. Section 3 will describe the literature review that was done and where the scientific community stands on research of legacy code. Section 4 will bring up the methods used to find results on how legacy code can be defined and identified. Section 5 will present the results from the comparative study of the two prototype implementations. Section 6
will describe the questionnaire that was done. Section 7 will contain reflections on the results found and Section 8 will summarize the research.
2 Background

Before a technique to observe code’s evolution to legacy code can be done, a clear definition for what legacy code means is needed.

The word legacy in software development is usually associated with legacy systems but is related to legacy code as well. Legacy systems have many definitions, but Bennett [1] defines it as “large software systems that we don’t know how to cope with but that are vital to our organizations”.

Legacy systems run the risk of exposing security vulnerabilities, incompatibility issues and loss of data. A common reason why legacy systems are still in use is because of their compatibility with the operating system that was used back then. As computers upgrade to better operating systems, these systems are suddenly not compatible [13].

With the legacy systems definition in mind, legacy code could be considered as the starting point for legacy systems. As a system evolves to include more features and implements newer technologies, its system complexity increases, and the code becomes less flexible to change.

2.1 Legacy Code

Legacy code is a rarely occurring terminology that doesn’t have an official definition. Just like with legacy system, there are many opinions on what the term means. But legacy code is a common problem in software development and can occur in any software project.

Software developers in the industry have many different interpretations of what legacy code means. Some of these are:

- Difficult-to-change code
- Code that “seems beyond care”
- Code that needs “to change but [developers] don’t really understand” [2]

A consensus seems to be that legacy code is code that is hard to change.
2.2 Examples of Legacy Code

The field below shows a code block that converts Celsius to Fahrenheit:

```javascript
var degreesCelsius = 18;
function convertToFahrenheit() {
    var fahrenheit = degreesCelsius * (9/5) + 32;
    return fahrenheit;
}
```

The function `convertToFahrenheit` uses a global variable named `degreesCelsius` to calculate a value.

Assume that further in the development, calls to `convertToFahrenheit` are made dozens of times. If the program wants to use another value than 18, it would have to set/change `degreesCelsius` whenever it was needed. So, you want the code block to receive input of degrees in Celsius instead because it’s a better design.

With the new changes, it now looks like this:

```javascript
function convertToFahrenheit(degreesCelsius) {
    var fahrenheit = degreesCelsius * (9/5) + 32;
    return fahrenheit;
}
```

The code block looks cleaner, but now there is a problem: all references to `convertToFahrenheit` in the rest of the code are now going to break because the rest of the code relied on the previous code structure to function. When no global variable `degreesCelsius` exists anymore and the function needs an input, the program will crash.

2.3 Implications

What kind of implications does legacy code have? A blog post suggests that legacy systems that arise from legacy code can be less secure than newer code, which opens the system up to vulnerabilities. And after some time of use, they pose a threat to the company [16].
3 Literature Review

3.1 Prerequisites

3.1.1 Inclusion/exclusion criteria

To find studies related to legacy code (and to some extent legacy systems) and find answers to the research questions, inclusion criteria is required.

Legacy systems and legacy code are related to each other in some ways, so there is a benefit in finding papers relating to legacy systems as well as legacy code.

The following inclusion rules that were formulated are:

- Papers that can be found in IEEE Xplore
- Papers of any age
- Papers that include legacy code (and/or any of its synonyms) in the text
- Papers that focus on researching legacy systems in personal computers
- Papers that can include a definition for legacy code (not all papers need one)

Papers to be excluded are:

- Lacking in a method or background segment
- Research of legacy systems on mainframe computers

After the first search process, another inclusion criterion was added: that legacy code or old code has to be included in the document title.

3.1.2 Keywords

Keywords that were used (and their synonyms) to find the papers are compiled into a table seen below.

<table>
<thead>
<tr>
<th>Original keyword</th>
<th>Synonym(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>software maintenance</td>
<td>software evolution</td>
</tr>
<tr>
<td>code maintenance</td>
<td>Refactoring, reengineering</td>
</tr>
<tr>
<td>legacy software</td>
<td>old software, legacy systems, old systems</td>
</tr>
<tr>
<td>legacy code</td>
<td>old code</td>
</tr>
</tbody>
</table>

Figure 1: Keywords and their synonyms for the literature review

3.1.3 Search Results

From IEEE Xplore, the following number of articles could be found for the original search strategy:

<table>
<thead>
<tr>
<th>Search term</th>
<th>No. of results</th>
</tr>
</thead>
<tbody>
<tr>
<td>“legacy code”</td>
<td>1430</td>
</tr>
</tbody>
</table>
Once the new inclusion criteria were added, the results were:

<table>
<thead>
<tr>
<th>Search term</th>
<th>No. of results</th>
</tr>
</thead>
<tbody>
<tr>
<td>“legacy code” in Document Title</td>
<td>53</td>
</tr>
<tr>
<td>“legacy code” in Document Title AND (detection OR discovery OR evolution)</td>
<td>9</td>
</tr>
</tbody>
</table>

Figure 3: Number of results once “legacy code” in document title was included

### 3.2 Survey of Related Works

While research about dealing with legacy code is plenty, there is not enough research on what legacy code is nor is there a lot of research on how legacy code is formed in a software.

Feathers [2] has studied the process of refactoring code and legacy code. Feathers states legacy code to be a “rot” that can spread in a program during development and eventually makes changing things in a software difficult for developers. Ultimately, his definition for legacy code is code with no unit testing involved.

Rizvi and Khanam [3] brought up a methodology for refactoring legacy code in a way to “minimize the required efforts and maximize the effective results”. In their article, they describe legacy code as badly written code that needs to be refactored to “extend its functionality” for instance when future features are made.

Mehta [4] writes that “aging code” results in lost assets for organizations along with wide complexity gaps between “user expectations and the software functionality” in the development of software. How aging code does that is not mentioned, but Mehta mentions that writing software with feature engineering – making features the main focus in software engineering that affects requirements gathering, testing and so forth – as an approach in the development helps to evolve legacy systems. This approach was
tested on a real software in a continuing work by Mehta [9] where system features that had gone through extensive maintenance and were likely to change were identified, similar feature implementations found in the system would be intertwined, and very small components from the legacy system were picked out. These components would exist between the original platform where the legacy system was based in and a web application planned for development.

A problem information systems face is increasing workloads for a software. Jin, Tang, Han and Liu [7] developed a “performance evaluation and prediction approach” that could help predict future production system performance. Testing their model on a real case scenario of electricity distributors’ Meter Data Systems, the businesses involved wanted to learn how long the current legacy system could last the massively increased data process volume before daily processing within a time frame would fail. Their model showed them that the prediction calculated was “accurate within the measurable range” and that they could approximately determine how long their system would last before the eventual collapse.

Mullen [8] considers in his blog post that code that goes through tests are less susceptible to unintentional breaks in the software than code that isn’t tested. However, badly written code should not be considered as legacy code if it has gone through tests. This statement falls in line with Feathers’ definition of legacy code as non-tested code [2].

In determining legacy system status, Sneed and Huang [10] conducted a pilot experiment to determine the size of maintenance tasks in web applications, which they believed had the ability to become legacy systems themselves. Their conclusion was that sizing maintenance tasks in an industrial environment was not an easy feat. Since it was only a pilot project with the scope on sizing “impact domains of maintenance tasks”, estimating task effort and comparing it with actual effort involved could have yielded interesting information for this thesis.

Siebra, Gouveia and Sodre [11] associate legacy code with code that once it gets refactored, it can cause unexpected problems in other parts of the program. Their research involved forming a test design methodology that can help developers further understand the meaning of what the code, code blocks and modules’ purpose is without full knowledge of how it works.

With a rapid and expected evolution of software and hardware, systems have the capacity of turning into legacy systems before it is even released. Zhou, Zedan and Cau [12] recognize a system’s road to “legacy status” as the evolution of the system in the
shape of “changes in the original requirements, adopting a different hardware platform or [improvement of] its efficiency.” Zhou, Zedan and Cau developed an approach to analyze the effect legacy code has when an application goes through changes, and afterwards implemented a tool that utilized this approach.

Kanat-Alexander believes refactoring to be a better direction rather than rewriting systems [15]. Rewriting legacy systems can end up introducing more complexity, if not introducing the same amount of complexity as the old one, to the new system. But rewriting is not the only option when a system becomes harder and costlier to maintain. Kanat-Alexander’s suggestions to only rewrite systems when or if:

- Estimates show that rewriting is more efficient than maintaining the old system
- You have plenty of time to write a new system
- You’re “a better a designer than the original designer of the system” to make a more improved version
- You intend to “design this new system in [...] simple steps and have users who can give you feedback for each step”, and
- You can maintain both the old and new system at the same time (since the old system will still be in use during the transition).

From the conducted literature review, a lot of focus in research has been on refactoring legacy code and conducting tests and techniques to decrease legacy code’s impact on the rest of the software. But any kind of legacy code discovery or prevention is brought up lightly in the found literature. The majority of definitions to legacy code points to being code with no testing done.

The explanation to legacy code having so many definitions unofficially can be because of the industry and the scientific community’s varied and subjective opinions on what legacy code means.

The table below shows a summary of the definitions of legacy code made by some authors:

<table>
<thead>
<tr>
<th>Authors</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feathers [2], Mullen [8], Kanat-Alexander [15]</td>
<td>Code with no testing</td>
</tr>
<tr>
<td>Rizvi, Khanam [3]</td>
<td>Badly written code that needs to be refactored to “extend its functionality”</td>
</tr>
</tbody>
</table>

Figure 1: Summarized table for definitions of legacy code according to authors found in the literature review.
4 Method

This section details what methods were used in this study.

4.1 Literature review

A literature review is needed to find out more about research regarding legacy code.

Since the thesis is more focused on legacy code as a definition and how we can identify it, papers and literature that cover code that can be defined as legacy will be prioritized if they can provide a description on what legacy code means to the authors. Since the research does not concern refactoring methods or reengineering, papers that cover refactoring techniques without any background or description of legacy code will be disregarded.

4.2 Comparative study

Two prototypes are to be made during the research.

The prototypes will be camera surveillance systems that can work in multiple platforms. IP-based cameras that are connected to the local area network should be accessed by these prototypes and be presented in a 3D environment.

A research project conducted in 2011 involved developing a 3D visual surveillance system with 3D models and utilized person tracking and crowd analysis [17]. These prototypes will utilize the 3D visualization and camera feed aspect but features like camera feed projection on the 3D space and object tracking should not be included.

The first half of the research will be developing the first prototype, and the second half will be the implementation of the second prototype.

In development of the first prototype, the development is agile. Requirements will change, the software structure is not fully defined and there is no end goal for prototype #1’s features or structure.

In development of the second prototype, all requirements gathered from the previous development will be used and new ones can be added but is not necessary. The software structure is predefined and the result of prototype #2 will be a first version of the application.

When these prototypes have been developed, a comparison of the development processes between the prototypes will be made. Observations on where legacy code was forming and suggestions how it can be identified will be noted and presented in the results for the implementation.
4.3 Questionnaire

To evaluate the definitions of legacy code found from the literature review and the suggestions for identifying legacy code, a questionnaire will be created based on the results from the literature review and the observations made from the implementations.

The first half of the questionnaire will be structured to present the literature review’s findings on legacy code definition. The second half will look at how the recommended suggestions for identifying legacy code, formed from the implementations, compare to the respondents’ own opinions. The respondents will be able to rank statements in the questionnaire with a score from 1 to 5, 1 means disagreement with the statement, 3 is neutral and 5 is agreement with the statement.

Finally, the respondent will be able to write down what their own definition of legacy code is.
5 Implementations and Comparison

5.1 Prototype A and B Description
Both prototypes that were developed are of the same type of software: a web-based camera surveillance system that uses 3D visualization to position surveillance cameras in a 3D world to make the user understand better where in the real world the cameras are located.

The user can select cameras in the 3D world to retrieve its camera feed and the user can select multiple cameras and switch to another window where the camera feeds cover up the entire screen.

\[\text{Figure 2: A diagram of how the computer and cameras are connected to a local area network.}\]

5.2 Prototype A
IP-based surveillance cameras act as web servers in a local network. The prototype can retrieve camera feed and properties like serial number and manufacturer through their API located at their IP address.

The metadata and camera feed sources in an object is hard-coded in the root component. This means that when the program is restarted, any changed values for the camera like which direction it’s facing will be reset to its original values.

5.2.1 Prototype A Observations
The first prototype, to be referred to as PA, was initially structured as a tree branch where a root component (“Wrapper”) oversaw all components underneath it, and any communication between “sibling components” were to be channeled to the root component and passed onwards to the target component. Calls to the backend initially was called solely from “Wrapper” but evolved into being called from many other components.
The backend had the sole purpose of connecting to the database where the camera information was stored, with values like IP address, video source and camera model name.

![System Diagram](image)

*Figure 3: PA's system diagram. The dotted line between component 1 and 2 indicates an implicit connection.*

The development time of the prototype was divided into two-week sprints. During the first week, PA had clear and well-defined requirements. PA’s software structure was designed, and PA was developed to follow this design.

```java
setup3D() {
    // setup 3d canvas
    // 100 lines of code...
}

setFloor() {
    // make all but the selected floors invisible
}

setHighlights() {
    // 50 lines of code...
}
```

For instance, these three functions existed in Component 1 and 2 in the program and did the same thing. When one of these functions were changed in ex. Component 1, there was a risk that the function in Component 2 would be forgotten to change.

However, some components later needed to have features that originally were exclusive to one component. For instance, Component 2 needed to have 3D features to show the 3D model of a building using basic functionality. Component 1 had those methods and variables related to the 3D features which Component 2 could reuse, but the 3D feature was combined with multiple other functions not relevant to Component 2’s needs which made breaking the 3D-related methods and code out of Component 1 difficult. The 3D feature coded in Component 1 was written for Component 1, no consideration was made that other components might need the 3D features. This
resulted in the code utilizing 3D was written into both Component 1 and 2 rather than in one single place.

As time went on and more functions started to replicate across multiple components, large quantity of code needed to be refactored, which in turn creates more effort and time spent for those assigned to deal with this issue. In the end, the program was not modular enough which would allow for components to be easily swapped or maintained. A change in the software design was determined at this point, and a new prototype was to be made.

5.3 Prototype B
In Prototype A, data about cameras and their video feeds were hard-coded into the program. The program resets all values when the program restarts. With cameras that use PTZ, the PTZ settings will always be reset when the program is closed.

Prototype B solves this by reading and editing a local data file that contains all the cameras to be included in the 3D map. This file can be read, edited and saved for the next time the program is run. Changes being made will be updated for the next time the prototype is run.

5.3.1 Prototype B Observations
PA was designed in a way that would make future additions easier to implement since there would be some code already written and ready to use in the future when more code would be added. It turned out that a lot of this code would not be used at all and became obsolete. Prototype B, PB, scrapped the obsolete code that was known to exist in the program.

PB was developed for a month and a half. The prototype was also a web-based camera surveillance system like PA. The requirements that had been gathered from the development of PA was used for development of PB.

From the requirements gathered from PA, considerations were taken like certain features are shared among components.
PB has a clearer structure in areas like variable usage, component structure and purpose for each module and code block. Refactoring was made easier. Stateless helper modules were used by many components. The Wrapper component was freed of many of its former tasks (for instance handling which cameras are selected that could have been done in the child components) that the child components had been relying on, by letting the Wrapper component passing on the relevant variables that every component needed to function. The issue with “sibling components” that needed to communicate was solved almost entirely due to the new system design that was made. Components were combined, refactored and reused in several places in the program.

5.4 Development Observations and Comparisons

<table>
<thead>
<tr>
<th>Component No.</th>
<th>PA – No. of methods</th>
<th>PB – No. of methods</th>
<th>PA – Other component method calls</th>
<th>PB – Other component method calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>51</td>
<td>47</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>1A</td>
<td>18</td>
<td>11</td>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>13</td>
<td>2</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>14</td>
<td>7</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 5: PA and PB's number of methods and their external component method call count.

As PA evolved to become more complex, its methods and number of components increased. By the end of development, PA had 5 components vital to the system and 9 sub-components not too important to include in the system diagram.
On further inspection, it was discovered that Component 1 and 1A had a tight coupling. This made code reuse of Component 1 or replacing it difficult without having to change or include Component 1A even though it had a very specific purpose.

Component 2 inadvertently became connected with Component 1 during development. This poses an issue if any of the components must be replaced, becomes outdated or goes through refactoring. A quick hotfix can turn into longer, more complicated bug fixes for both Component 1 and 2 (and any other that might be connected to them). Had documentation existed for times when unconventional and poor code were made or comments on code that needed to be improved later, the refactoring process could have been easier.

PA’s software design was not structured to handle more than the main functions: displaying data about cameras in a 3D world and displaying camera feed. When more features were added that concerned anything other than the 3D and camera feed aspect (which accounted for the whole program at the time), refactoring in many parts had to be done.

In the case of PB, these new requirements that were formed during PA development led to less refactoring needed due to the known nature of expected and unexpected behavior in the code, which led to more time implementing the rest of the features.

Badly written code can also turn into legacy code. If a badly written method or component becomes too important for the software, making a replacement might result in a similar structured method/component and resolves no major problems. But good written code can have the same effect as well, especially if the design that was made at a time when it was most suitable is no longer appropriate.

5.5 Suggestions for Identifying Legacy Code

Identifying legacy code as they form or were about to form was difficult especially when the software complexity increased by time. However, there were bad habits being made during both implementations that could likely cause legacy code.

An experienced developer can understand people’s code by reading it and understanding what it does. But if the developer cannot understand why the code does this and that, it becomes an issue. The developer might not know how important this code is to the software because of either lacking or badly written documentation. A suggestion for identifying legacy code is if a certain block of code with or without documentation cannot explain why it does what it does.
Software design, like code, needs to evolve through time as a software grows to include more features and better performance. In the development of PB, the software design wasn’t remade from scratch. It was edited to work with the new requirements and features. Remaking the software design did not make sense in the context of this research since there were no other software designer in this thesis. It should be acknowledged that bad software design can make bad software. When a new software design is made, the code that remains from the remake of the design will still be written based on the old software design. If this code is not refactored, it can open up to more problems when the new software design becomes more prominent and the old code becomes more important. A suggestion for legacy code identification could be software design that was made poorly from the start.

There is also the risk of depending on too many external libraries and modules – either written by the developers, or by a third party – since these can evolve as well. Some libraries can evolve faster than the software being developed, and others can depreciate leaving you with few options like migrating to another library (which means rewriting the affected code) or having to maintain the depreciated library to work with the software. If that is the case, legacy code evolution could be identified when it starts to depend on too many external libraries and modules.

There is also the aspect of quality in writing code. If an easy-to-implement code is made unnecessarily difficult, that could make legacy code form. Also, if methods and variables are given foreign, too long, too abstract and irrelevant names, that can cause future problems for not only the other developers in the project, but the author of the code as well as the more unusual names are made and the longer time it takes for the author to revisit the code.

To summarize, the following suggestions for identifying legacy code could be:

- Lack of description why a certain code block performs a task
- Badly written or no documentation
- Built on old and/or bad software design
- Depending on too many external libraries and modules
- Bad syntax

These suggestions will be evaluated with the upcoming questionnaire.
6 Questionnaire Survey

6.1 Structure

The observations and findings made from the implementations and the literature review were tested on six software developers in the form of a questionnaire. The survey was done face to face, and the participants were to cross their answers on a sheet of paper. The participants were given 12 statements where they could answer on a Likert scale of 1 to 5 if they agreed with the statement, 1 disagreeing with the statement, 3 being neutral and 5 agreeing. They were also given an option to describe what they personally think legacy code is.

6.2 Statements

The statements that were used in the questionnaire were:

1. Legacy code is not tested code.
2. Legacy code depends on too much external libraries and modules.
3. A method with too many operations should be considered legacy code.
4. Bad coding habits cause legacy code.
5. Bad software design causes legacy code.
6. Code doesn’t need documentation to be considered legacy code.
7. Code is considered legacy as soon as it’s written.
8. Code is considered legacy when requirements related to the code are changed.
9. Any developer can identify legacy code before it evolves.
10. Documentation prevents the code to become legacy.
11. Good documentation prevents the code to become legacy.
12. Bad variable and method names causes legacy code.

6.3 Results

The table below summarizes the survey and shows the average score for each questionnaire statement along with their sample standard deviation. Individual responses to the questionnaire can be found in Appendix A.

<table>
<thead>
<tr>
<th>Q. No.</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>2.33</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>3.5</td>
<td>4.5</td>
<td>1.33</td>
<td>4.33</td>
<td>2.5</td>
<td>2.67</td>
<td>3</td>
<td>3.33</td>
</tr>
<tr>
<td>St. dev.</td>
<td>1.37</td>
<td>1.1</td>
<td>1.41</td>
<td>0.89</td>
<td>0.98</td>
<td>0.84</td>
<td>0.52</td>
<td>0.82</td>
<td>1.05</td>
<td>1.21</td>
<td>1.41</td>
<td>0.82</td>
</tr>
</tbody>
</table>

*Figure 6: The statements in the questionnaire and how they scored on an average (mean). The standard deviation shows how spread out the scores are from the mean. The lower the number, the more the answers are closer to the average score.*
The standard deviation was calculated to show how spread out the answers were for each statement. This helps to show if the participants were sharing the same opinion on the statements. If a standard deviation for a statement is close to zero, then the average score for the statement would be the likeliest answer for any participant. For instance, statement #7 had a standard deviation of 0.52 with an average score of 1.33, meaning that almost all participants (4/6) gave statement #7 a 1.

The participants were given an open question to define what legacy code means to them. The responses from the participants were mainly that legacy code is old code. Some participants defined legacy code as code that through time is no longer fit for use, might have “security holes” and uses depreciated libraries and modules and even coding standards that would have made sense earlier. Also, code that was poorly documented and inherited from another developer defines the code as legacy. Statement #1 dealt with legacy code definition explicitly in that legacy code could be untested code. This statement received an average score of 3, which is neutral.

Statement #6, #10 and #11 dealt with legacy code and documentation. The results show that good documented code preventing legacy code is something developers are neutral to, documented code preventing legacy code is something developers disagree with and code not needing documentation to be legacy is something developers agree with.

Results from the survey showed that the participants on average mostly agreed with statement #5, that software design that’s poorly done causes legacy code.

The participants were neutral about legacy code depending on too much external libraries and modules, with statement #2 receiving an average score of 3.
7 Discussion

There have been plenty of opinions, interpretations and arguments on legacy code. This makes classification of actual legacy code difficult to determine during the implementation phase. During the research, no definition for legacy code was made before the implementation. Had for example Feathers’ theory of legacy code – code with no testing done – been tested, an experiment on building a program with one half of the code built with tests and the other with none could have produced interesting findings.

The choice of participants for the questionnaire was important because they needed to be developers, or at the very least be involved with software code. There were some developers however that programs mainly middleware or hardware code, which might make their interpretation of legacy code different from developers who deal with software. This could be an interesting future research to find more information. For this questionnaire, it should be assumed that legacy code is a phenomenon that exists on all kinds of levels, from software to hardware, and those developers who program hardware and middleware deal with the same issues as a software developer.

The questionnaire was mainly formed around what the scientific community considers legacy code to be. There is a risk that the scientific community’s definition of legacy code is not in alignment with what the industry believes legacy code to be. That can produce differing results in types of research where developers in the industry are involved like surveys and personal interviews.

While most of prior research considers legacy code to be ugly code that needs to be refactored and untested code, it is well known in the industry that legacy code is also old. This is something that some participants in the questionnaire considered legacy code to primarily be. Some participants also described legacy code with characteristics that can be attributed to “old” code: depreciated libraries and security exploits being examples of things that happen in time.

Is legacy code a characteristic of code or a phenomenon? Techniques to try to identify legacy code has been difficult when it is not certain how legacy code should be treated as. Legacy code is not something that is written into the code that makes it legacy, it is mostly when circumstances and the software environment changes, does code that currently exist become legacy. Taking this into account, this makes code reach legacy status instantaneously when something as inevitable as software requirements being added or changed happens. When requirements are made, they often build on another functionality or set of requirements already existing. This
usually means refactoring the code related to the old requirements or functionality to make the new code function. When requirements are only changed, the code is refactored as well. For instance, if it is requirements that deals with providing more secure procedures (like input validation), the already existing code needs to be changed to fulfill these new requirements. If the task is easy and the code is easy to change, then the code should not have legacy status. But if the code is too deep-rooted into the current code base, then making improvements and changes will require more work and effort to accomplish. One of legacy code's characteristics should be code that is difficult to change due to being too “fixed” to the code base. This seems like code being dependent on external libraries, modules and other code. It is hard to change because other code depends on it, and it is hard to change because it depends on current versions of libraries, modules or other pieces of code to function. You could identify legacy code if a piece of code is too deep-rooted into other parts of the software, or even worse, deep-rooted to the point that the rest of the code depends on it. Many IDE's available can check for references for any selected method or class and it gives the developer a clearer view of how “important” it is to the program.

Tracking requirement changes and mapping them in a good way to the code could potentially be used to identify legacy code, but it would require a large amount of documentation to aid in finding all code related to the “old” requirements whenever new requirements based on the old ones are made. This can be time-consuming for developers, especially when they must work quickly.

Legacy code is such an abstract concept that it’s hard to tell when code becomes legacy because there is no appropriate currently existing metric that can be used for explicitly showing its legacy status. Time is inappropriate since it’s among other factors the circumstances in the project and the software structure that affects how quickly a code is considered legacy. LOC is not good either, since the amount of LOC should have no correlation with legacy code status. Refactoring for instance can lead to more LOC but will in most cases increase system performance and even code readability, so LOC is technically not an appropriate metric to identify legacy code.

During the implementation, legacy code happened to form even though measures were being taken to identify and prevent them. Reflecting on the work done and the previous literature review, there should be some practices established to identify and/or prevent legacy code from occurring. The degrading quality of a modular design was one of the factors in making legacy code form, which was mostly observed in PA.
Comparing the development between PA and PB, more legacy code was more likely to exist in a software that was developed with ever-changing software requirements rather than in software with software requirements set from the start of the project. It could be assumed that linear software development processes are less prone to producing legacy code than agile processes, but there needs to be more research.

The prototypes were written using procedural programming, a programming paradigm that structures program tasks and instructions into functions (procedures). Functions are in general modular and state-less, which lets the scope of a program change without too much inconvenience. Another common programming paradigm is object-oriented programming, where a program is structured with classes and instantiating objects of these classes to perform program instructions. Procedural programming could more susceptible to legacy code forming faster than object-oriented programming and vice versa, but future research is needed.

Unit testing works by setting up tests to a system on a “unit” level. When a code block or subroutine goes through changes, the unit tests related to the code that already exist and were previously known to pass can raise errors if the tests no longer pass on the new code. It is disconcerting to see that the mean score on Statement #1 in the questionnaire (Legacy code is not tested code) is so low, because unit tests can determine if they pass or not on factors like expected output and data types, if errors are handled gracefully and even performance and downtime. So, despite legacy code being untested is not agreed upon by the respondents in the questionnaire, it should still be included in the definition since code that include tests very likely have higher odds of staying fresh compared to code that doesn’t.

The implementation phase could have been improved by having more authors who could work on the prototypes. With more than one author, making further observations would be possible. Observations that could add more value for this research had there been others involved who could evaluate each other and themselves in the implementation. Using yourself as a tool in the research (writing the software) should be acknowledged as one might not be skilled enough or understand code quality to provide accurate results, in this case good software and code.

No code review – where the code is ensured that it holds quality in terms of performance and readability for instance – was made during the implementation phase due to preference of producing functionality before assuring quality code. Had another
person been a part of this research who could do code review, then the implementation phase could have taken a different route.
8 Conclusions

The research questions of this thesis were:

- what the recommended definition for legacy code should be
- at what point during development can code be determined as legacy code.

From the literature review, the implementation of both prototypes and the subsequent reflection of the results, the recommended definition for legacy code should be code (either parts of the software or the whole software) designed for past circumstances, with no testing involved and being hard to understand. Legacy code should also be considered old by the owners of the code, since the questionnaire results showed that some of the respondents believed in that aging code can affect currently existing code to become legacy.

Legacy code is hard to identify as it grows and can only in most cases be apparent when it has fully evolved. This in turn makes it hard to tell at what point during development will code become legacy. This became evident during the development of the two prototypes. For this reason, there is more research on working with legacy code as it exists than identifying legacy code as it grows.

There are methods in software development that can make managing of legacy code at a later stage easier. Some of these methods can be:

- Setting up unit tests to ensure that the code produces the same results as it would if it and other dependent code was to be changed
- Minimizing dependencies and monitor external library changes and depreciation
- Looking out for anti-patterns like hard coding or bad variable naming
- Designing a resilient and flexible software structure design from the start and provide documentation
- Building more self-supporting modules for easy replaceability
- Providing documentation throughout the development and update when necessary

Legacy code can never be removed entirely from software since it exists in some degree. With the inconsistency of legacy code definition, the recommended and ideal definition brought up in this paper should be the one that any developer can accept as it can result in more accurate discussions revolving legacy code and systems and how they can be dealt with. Future research is needed to further understand legacy code in how it grows and how it can be prevented.
9 References


Appendix A

The following tables show the results of the questionnaire. Six participants were to score the twelve statements – which can be seen in Table A – and provide a description for what legacy code means to them, which is displayed in Table B.

<table>
<thead>
<tr>
<th>Participant</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statement #1</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Statement #2</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Statement #3</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Statement #4</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>Statement #5</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Statement #6</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>Statement #7</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Statement #8</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Statement #9</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Statement #10</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Statement #11</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Statement #12</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Table A: Scores given by every participant for every statement.
<table>
<thead>
<tr>
<th>Participant</th>
<th>“How would you define legacy code?</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Code that is no longer fit for its intended use. E.g. a software intern who cannot understand the codebase due to poor documentation/coding habits. E.g. external changes that forces the code to be updated.</td>
</tr>
<tr>
<td>B</td>
<td>Anything I don’t understand and/or I can get a grip of, is legacy code to me. If I have to spend one day just figuring out what it means its legacy</td>
</tr>
<tr>
<td>D</td>
<td>Some frequently used exiting code which does not require any sufficient changes.</td>
</tr>
</tbody>
</table>
| E           | Definition 1: “Bad/Ugly code” that doesn’t follow coding standards: Not tested, bad naming, bad structure, usually missing or incorrect documentation, can only be read by original programmer. Usually bad version control history with bad commit messages. 
Definition 2: Code that uses ancient and deprecated libraries or methods. Usually old code left for backwards compatibility, that might be very different from current coding standards, (but was ok when written), might have security holes and might complicate build process due to lots of unique prerequisites. 0 |
| F           | Old code that either is yours and was written at least a year ago, or code that once belonged to someone else that might not be here anymore. Oftentimes poorly done and hard to change |

*Table B: The answers given by the participants on what legacy code means to them.*