TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

SUSANNA SINISALO

ENHANCING UNIT TESTING TO IMPROVE MAINTAINABILITY OF THE SOFTWARE

Master of Science thesis

Examiner: Prof. Hannu-Matti Järvinen
Examiner and topic approved by the Council of the Faculty of Computing and Electrical Engineering on 4th May 2016

ABSTRACT

Susanna Sinisalo: Enhancing unit testing to improve maintainability of the software
Master of Science Thesis, 50 pages
October 2017
Master's Degree Programme in Information Technology
Major: Software Engineering
Examiner: Professor Hannu-Matti Järvinen

Keywords: legacy project, maintenance, quality improvement, refactoring, unit testing

Many companies are using software that have been developed many years ago. These legacy programs are important to their users and often their development is still ongoing. New features are developed and old ones adapted to suit the current needs of the users. Legacy programs often have quality problems, which makes it increasingly difficult to maintain them. But because of their size, numerous features, and business rules that are documented only in the code, it is difficult to replace them. Therefore, it is imperative to improve their quality safely without introducing defects to the current features. Quality can be enhanced by unit testing and refactoring, but in legacy projects writing unit tests and refactoring is usually error prone and difficult due to the characteristics of legacy software. Taking quality into account when developing new and legacy software is important to ensure the software can be developed further and used in the future. High quality code is maintainable even when it has been developed a long time.

First in this thesis, legacy software problems in quality, refactoring and unit testing were researched. Then, solutions to avoid the problems and increase the quality of software and especially legacy software mainly by safe refactoring and unit testing were searched from literature and research publications. Quality control and metrics were also included in the thesis to supplement the quality enhancement process.

In this thesis, it was found that safe refactoring, unit test quality, code and test quality control, and developer training need to be considered to improve software quality. Refactoring should be done in small steps while writing tests to ensure the current functionality does not change. Unit test quality, especially, test isolation, readability, and test focus need to be considered while writing tests. The quality of the tests need to be monitored to maintain a proper quality level. It is also important to support the developers for them to improve and maintain their unit testing and refactoring skills.

TIIVISTELMÄ

Susanna Sinisalo: Yksikkötestauksen kehittäminen ohjelmiston laadun parantamiseksi
Tampereen teknillinen yliopisto
Diplomityö, 50 sivua
Lokakuu 2017
Tietotekniikan diplomi-insinöörin tutkinto-ohjelma
Pääaine: Ohjelmistotuotanto
Tarkastaja: professori Hannu-Matti Järvinen

Avainsanat: yksikkötestaus, laatu, legacy-ohjelmisto, refaktorointi

Monessa yrityksessä on käytössä ohjelmistoja, joiden kehitys on aloitettu useita vuosia sitten. Nämä legacy-ohjelmistot ovat käyttäjilleen tärkeitä ja usein niiden kehitys jatkuu koko ajan. Uusia ominaisuuksia lisätään ja vanhoja muokataan täyttämään käyttäjien tarpeita. Legacy-ohjelmistot sisältävät usein laatuongelmia, jonka takia niiden ylläpitäminen ja kehittäminen vaikeutuu koko ajan. Niitä ei voida kuitenkaan helposti korvata kokonsa, lukuisten ominaisuuksiensa ja vain koodiin dokumentoidun tietonsa takia. Tämän takia niiden laatua pitäisi saada parannettua turvallisesti ilman, että virheitä ilmestyy toimiviin ominaisuuksiin. Yksikkötestauksen ja refaktoroinnilla voidaan parantaa ohjelmiston laatua, mutta niiden käyttönottaminen ja käyttäminen legacy-ohjelmistossa voi olla riskialtista ja vaikeaa legacy-ohjelmiston ominaisuuksien takia. Laadun huomioonottaminen uuden ja legacy-ohjelmistoa kehittäessä on tärkeää, jotta ohjelmistoa pystytään kehittämään edelleen ja käyttämään myös tulevaisuudessa. Korkealaatuinen koodi on melko ylläpidettävää myös, kun sitä on kehitetty pidemmän aikaa.

Aluksi tässä työssä selvitettiin, mitä ongelmia kuuluu legacy-ohjelmiston yksikkötestaukseen ja refaktorointiin. Sen jälkeen etsittiin ratkaisuja, kuinka välttää näitä ongelmia ja kuinka parantaa ohjelmien erityisesti legacy-ohjelmistojen laatua turvallisella refaktoroinnilla ja yksikkötestauksella. Tapoja etsittiin kirjallisuudesta ja tutkimusjulkaisuista. Laadun valvonta ja metriikat otettiin täydentämään laadun parannusprosessia

Työn aikana saatiin selville, että ohjelmiston laadun parantamisessa pitää ensisijaisesti huomioda testien laatu, turvalliset refaktorointitavat, koodin ja testien laadunvalvonta ja kehittäjien osaamisen kehittäminen. Refaktorointi tulee tehdä pienissä askeleissa kurinalaisesti ja testien avustamana. Yksikkötestien laadussa tulee erityisesti ottaa huomioon eristyneisyys, luettavuus ja kohdennus. Koodin ja testien laatua tulee valvoa, että se pysyy yllä. Kehittäjiä täytyy myös tukea, jotta heidän yksikkötestaus- ja refaktorointitaitonsa kasvaisivat ja pysyisivät yllä.

PREFACE

Tämän työn aihe sai alkunsa kiinnostuksestani ohjelmistojen laatuun ja halustani tietää, kuinka varsinkin legacy-ohjelmistojen kehitystä voisi helpottaa ja parantaa. Työtä tehdessäni olen oppinut paljon ja innostukseni asiaan on vain kasvanut.

Haluaisin kiittää vanhempiani kannustamisesta tälle mielenkiintoiselle alalle sekä aviomiestäni Gakua henkisestä tuesta. Haluaisin myös kiittää veljiäni Aleksia ja Markusta tuesta koulutaipaleella ja myös tämän työn kanssa. Kiitos myös professori Hannu-Matti Järviselle neuvoista tämän työn kirjoittamisessa.

Tampereella, 15.10.2017

Susanna Sinisalo

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

API	Application programming interface
DIP	Dependency inversion principle
DRY	Don't repeat yourself, a design principle
DSL	Domain-Specific Language
ISP	Interface segregation principle
IoC framework	Inversion of control framework
LSP	Liskov substitute principle
OCP	Open / Closed principle
SOLID	Mnemonic acronym for five design principles: SRP, OCP, LSP, ISP, DIP
SRP	Single responsibility principle
TDD	Test-driven development

1. INTRODUCTION

Many software companies have legacy projects that have been in maintenance state for a long period of time. There has been multiple changes and corrections to the code, and the software has adapted to the current needs of the customer. Therefore, they are very important to their users and contain a large amount of business-related logic. [1] Legacy projects usually have low quality due to several reasons: the developers' lack of knowledge on how to develop high quality software, sub-optimal design choices caused by time pressure, and lack of refactoring. These quality problems complicate further maintenance and increase the risk to introduce defects to the existing functionality. Additionally, it is difficult to introduce changes, add functionality, and locate and correct errors. [2] However, maintenance has to be continued, because replacing the system contains great risks, because of the size and numerous features of the legacy project. [1] Therefore, the quality of the legacy project has to be somehow improved without introducing defects in the process.

This thesis was carried out as literature review, and suitable articles, books and research papers were searched mostly from IEEE Xplore and 24x7Books. Some of them were found by a search engine or they were recommended by unit testing and refactoring book authors. Unit testing was selected over integration testing to be the tool for improving quality, because of the short feedback loop that it provides over integration tests. Problems and solutions on unit testing and refactoring a legacy project were searched from the literature. The most resource-efficient and important solutions are presented in the result section.

The goal of this thesis is firstly, to provide suggestions on how to improve the quality of existing software safely by refactoring, using unit tests, and quality control. The second goal is to suggest methods to write high quality code.

Chapter 2 defines, what a legacy project is and what kind of problems are related to its quality, unit testing and refactoring. Chapter 3 discusses methods to improve quality in general and how to write high quality code. Chapter 4 introduces methods to write high quality unit tests. Chapter 5 examines methods to do safe refactoring. Chapter 6 presents suggestions on how to start improving the quality of a legacy project and how to maintain good quality.

2. LEGACY PROJECT QUALITY PROBLEMS AND CONSEQUENCES

One common definition of legacy software is that it is software that has been developed by someone else and handed down to new developers. According to [3] legacy software is a software without automated tests. The industry generally sees legacy software as software that is difficult to understand and modify [3].

Even though legacy software can be difficult to understand and modify, they are important to their users and have been in use for a long time. They are usually hard to replace because of the size and numerous features that have accumulated over time. Generally, the documentation is incomplete so the features that should be in the new system cannot be easily specified. Business logic and processes exist only in the code and usually there is no other documentation of them. The knowledge would be easily lost in writing a new replacing system. Rewriting a legacy system is known to contain great risks. Exceeding the planned budget and schedule is common. Therefore, their maintenance has to be continued even if the existing software is difficult to maintain. [1]

2.1 Common traits and quality problems in legacy projects

Legacy software often have quality problems due to their common traits: poor structure, duplicated code, poor readability, and lack of tests. Low quality makes maintenance and implementing new features difficult and error-prone, which can lead to delays in deliveries and customer dissatisfaction.

There are some common traits that most legacy projects share. One of them is poor structure. In legacy project, there have been multiple modifications, corrections and refactoring over a long period of time. The original structure is not visible anymore, and some features and methods are not in the classes or modules they should be. Some classes are too wide and contain methods that should be separated into another class. This is due to the fact that when doing modifications developers do not usually think or know about the greater design. The developers may not be aware of the architecture, because the system is so complex that it takes time to understand the complete structure, or the system is so complex that the architecture does not exist anymore. The developers might also have insufficient knowledge of patterns and antipatterns to recognize poor structure, and to create good structured code. There can also be schedule pressure which forces developers to do hacks. This leads to accumulating problems. Developers tend to make changes to the parts of the system they know. Those parts will then grow, and become more complex and difficult to maintain. Therefore, it is highly important to make the whole team aware of the architecture, and to assess it from time to time. [1-4]

Duplicated code is also a common trait. Duplicated code emerges, when a developer copies a part of the system that they need to another part of the system, and modify the variable names or the code a little to suit their needs. When this a modification is needed in this code, developers are forced to do the same modification to multiple parts of the system, which makes the process more error-prone. More modifications mean more risks. Duplicated code is difficult to find without an automated tool. Therefore, it is highly likely that some duplicated parts will go unnoticed, and the intended modification is not implemented in all of the duplicated parts. In the case of a defect, this means that the same defect that was corrected already, will surface in other part of the system. [4]

Legacy project code has been developed by multiple developers, and therefore the code contains numerous different coding styles. This forces the developer reading it to learn to understand all the different styles and the developer has to think if there is a meaning behind the style change. The code may also contain memory and performance optimizations, which makes the code more difficult to understand especially for more inexperienced developers. Long functions, poorly and inconherently named variables, unreachable code, deep nested conditional statements, and poor structure make the code difficult to read and understand. Therefore, modifications and maintenance require more time and effort. Modifications to unreadable code is also more error-prone, because complex structure and not fully understanding the code increase the probability of making a mistake. [1; 2; 4].

Other common trait in a legacy project is the lack of automated tests. The problem with not having tests is that the software cannot be verified, and therefore, it cannot be modified with confidence. Increasing quality by refactoring might introduce regression faults in the software that might not be noticed. It is also common that code without unit tests has testability problems, and therefore it is usually difficult to get units under a test harness without doing modifications. [2; 3] The lack of tests forces the company to do extensive manual testing or system level automate testing, which use a large amount of resources in people and in time. [5]

It is also common for legacy projects that there is not much other documentation on the system apart from the source code [1]. Important business logic is not documented or the documentation is obsolete [2]. Therefore, it may be that none of the developers knows exactly how the system is supposed to work. During refactoring, business logic might have to be retrieved from the source code. This may be difficult because of the poor structure of the code and readability issues. It may be also difficult to know when system has an error and when it is working correctly.

In legacy projects, it is usual that also the developers have insufficient knowledge about code antipatterns, good quality patterns and principles and experience in applying them. Developers who do not know coding principles and patterns or have not been using them in their work will make code that lacks in quality. They will also not do well in code reviewing and can mentor other developers into following wrong practices. [2]

2.2 Causes and consequences of low quality

According to [2] low quality consists of four parts:

1. Code: Static analysis tool violations and inconsistent coding style.
2. Design / structure: Design antipatterns and violations of design rules.
3. Test: Lack of tests, inadequate test coverage, and improper test design.
4. Documentation: No documentation for important concerns, poor documentation, and outdated documentation.

Poor structural quality increases the time and effort to understand and maintain software. New changes are impacted by the existing poor design and they have to be adapted to the poor structure further lowering the quality of the software. Poor structure encourages or even forces developers to do sub-optimal design decisions to implement the change. These kinds of changes will lead to increasingly lower modifiability and eventually the system may have to be abandoned. Poor structure also impacts the morale and motivation of the developers, because changes are difficult to make and refactoring the structure is not trivial either. [2]

Some examples of the consequences of poor software quality include the following [6]:

- Delivered software frequently fails.
- Consequences of system failure are unacceptable, from financial to life-threatening scenarios.
- Systems are often not available for their intended purpose.
- System enhancements are often very costly.
- Cost of detecting and removing defects are excessive.

Delivering low quality software to customer can have great negative impact on the reputation of the company, and therefore it should be monitored and managed properly. [2]

2.3 Problems in unit testing a legacy project

Unit testing a legacy project is usually difficult. It is difficult to write tests for existing code, because dependencies are usually difficult to replace, and the state of the object is difficult to observe. The existing code would need to be refactored first to implement unit tests easily, but the low number of existing tests makes refactoring unsafe, and the risk of introducing new defects high. It can be difficult to decide where to start unit testing. Getting high line coverage and improving quality will take time. [7; 8]

If there are existing tests, they usually have low quality, which causes problems during development. Maintainability for tests is highly important, because unmaintainable unit tests may jeopardize the project schedule. Low quality tests break often and require resources to be maintain without giving the regression safety-net they should. [8]

Legacy project unit tests may have dependencies to other parts of the system that makes them slow to run. They may be difficult to run, for example, they are started from command line and run in a separate window from the development environment. The tests can also require configuration before they are run. Developers will not want to run the tests if they take a long time to finish or if they are difficult to run. If the tests are not run, regression will not be noticed until it is already difficult to know, which part of the new code broke the tests. [8-10]

Non-isolated tests fail randomly, because other tests affect their results. This can be because tests have to be run in certain order, tests call other tests, or they share in-memory state or a resource, for example, a database without resetting it in between. Randomly failing tests make it difficult for the developers to trust their results, and real defects can go unnoticed. [8; 10]

Overspecified tests break easily when unit’s internal code is changed. Internal code is frequently changing, and therefore overspecified tests have to be maintained often. Overly specified tests usually test purely internal behavior, check communication with doubles when it is not needed, or assume specific order or exact string when it is not required. A needless test on internal behavior can, for example, test the internal state of the object after initialization. Using doubles to test communications between the unit under test and its dependency, exposes the internal call order and structure of the unit, which can change often. The test tries to force the unit to use its dependency in a certain way, which is not maintainable. Assuming specific order of a list or exact string in unit’s output is not maintainable, because order and messages can change often. [8]

Unreadable tests can have test names that do not tell what the test does. If the test name does not contain enough information about which method is tested, with what input and what is expected, the reader may have to read the test code to find out this information, which is slow. Tests using plain numbers instead of well named variables can, especially in combination with poor test naming, make understanding the purpose of the test difficult. The developer may even have to read the original code to understand the test. Having a method called inside an assertion makes the test difficult to read as well. [8]

The existing tests may be unfocused and contain multiple assertions. Unfocused test has only small logical coverage, which means that the code under test may still contain defects. It is also more difficult to determine the cause for failure, when there are multiple assertions in one test instead of multiple tests, because most test frameworks end the test, when one assertion fails. The remaining assertions will thus not be run and their results cannot be used in investigating the cause of the defect. Multiple assertions add complexity to the test, which makes it more difficult to read. Also, using setup methods in an unreadable way makes the test less readable. Unreadable way to use setup methods is, for example, to initialize objects or doubles that are not needed in all the tests, which makes it is difficult to know for the reader, what preassumptions the test uses. Long and complex setup code lowers the quality of the tests. [8]

Even if the dependencies have been replaced there may be problems with the double objects themselves. If Application programming interface (API) of the dependency is poorly done, the user has to know too much about the internal implementation of the dependency and how to use it. This makes creating doubles more difficult, because many return values for methods have to be specified in the setup phase, which makes the test needlessly long and difficult to understand. The architecture may not provide ways to replace dependencies easily. Mock frameworks cannot usually mock direct implementations of a class, but nowadays there are some frameworks that can: TypeMock [11] and JustMock [12]. These frameworks can make mocking legacy projects easier, but it is argued that they should not be used extensively, because they do not encourage good coding practices like normal frameworks do. [10]

Replacing dependencies in a legacy project can be difficult due to several reasons: [5]

- Can't instantiate a class.
- Can't invoke a method.
- Can't observe the outcome.
- Can't substitute a collaborator.
- Can't override a method.

Even though some frameworks enable replacing any kind of dependencies, the setup process can be too difficult to be effectively used. Implementing too complex mocking setup will result in brittle tests that break when a small change is done. Therefore, problems related to doubles should be first and foremost solved by safe refactoring and not extensive mocking. [5]

If the tests have been written after the code, the tests themselves can contain defects, which will cause them to pass and break unrelated to the code they are trying to test. Especially logic in tests increases the probability that they contain defects. A test case with logic most likely tests multiple features, which leads to it being less readable, and more fragile. The test can be also difficult to re-create, when it finds a problem. If the tests frequently contain defects, the developers will not be able to trust them, and they will not run them. [8]

Low quality unit tests are easy to make, but they give no extra value to the project. They rather lower the maintainability of the software by making refactoring and changing the code difficult. These tests usually contain too many references to other parts of the system. [9]

When unit tests are written it is common that unit test quality is not monitored and there is no strategy in writing them [13], which can lead to unmaintainable tests and uncomplete test sets.

2.4 Problems in maintaining a legacy project

There are four reasons to change a program: 1. implementing new functionality, 2. defect correction, 3. improving design (a.k.a. refactoring), 4. improving the use of resources (a.k.a. optimizing). When working on the code, there are three things that can change: structure, functionality, and resource usage. [3]

When implementing new functionality only a small amount of functionality is added, while the rest of the existing functionality needs to be preserved. [3] Preserving existing functionality is difficult, which makes maintenance more demanding than writing new code. The process of implementing new features is the same in maintenance, but the restrictions of the existing system have to be taken into account, when writing new functionality for a legacy system. Maintenance tasks require wide knowledge about software development: ways of observing a program, maintenance and testing tools, and software testing, including process of writing new features. Maintainer also need knowledge of the legacy system itself. [1]

Usually maintainers do not have enough information about the software and the application field. Documentation is usually insufficient, deprecated or does not exist, and therefore the information has to be acquired from the code. Consequently, the quality of the code is highly important, and it affects greatly on how maintainers gain knowledge of the system. [1]

Refactoring is the act of improving design without changing its behavior. The software's structure is altered to make it more maintainable. There are common problems in refactoring. One of them is that the refactored part of the system is critical and developers are afraid to change it. Without automate tests, that legacy projects commonly lack, it is difficult to preserve the existing functionality. Therefore, it is common that developers minimize the risks by adding code to existing classes and methods, which leads to increasing method and class sizes, and unreadable code. This leads to more problems, because refactoring and understanding large methods and classes is difficult. [3; 5]

Dependencies between classes are one of the greatest challenges in refactoring. Classes that depended on concrete implementations are difficult to test and modify. Working with legacy project is largely breaking dependencies to make modifications easier. Reasons to breaking dependencies are 1. sensing, when dependency prevents inspecting of the values that the code calculated, and 2. separation, when unit cannot be put into a testing harness because of the dependency. After separation, a double can be inserted instead of the dependency. [3]

Changing published interfaces is more complicated than changing interfaces that are used by the code you have access to. If interface has been published and it is used by others, old function has to be supported for a while after implementing the new function, so that users have time to adapt their software to the new. [4]

3. IMPROVING QUALITY

Traditionally, quality has implied that the software fulfills its requirement specification. However, this definition has its problems, because requirement specifications and their documents are generally incomplete. It is difficult to fully document all the requirements customers have for a software system. Therefore, fulfilling an incomplete set of customer requirements does not guarantee customer satisfaction. In addition to the customers' requirements, the software should also fulfill the requirements of the people developing it. Other definition for quality is “fit for use”, the system does what the customer needs. This definition includes quality attributes that benefit the customer. These include attributes closely related to customers like usability and reliability, but also attributes related to developer work like maintainability, testability, changeability, extensibility and reusability. Quality attributes are interrelated, and therefore high reliability cannot be achieved without paying attention to internal attributes of the system. Quality can be improved and measured. [1; 6]

3.1 Quality management

Quality can be achieved only, if it is taken into account during software development process. It costs multiple times more to correct defects, when they are found late in the software project or by the customer. This can hurt the reputation of the software company, and cause schedule problems and financial losses. Therefore, quality management and defect prevention, although causing costs during development, will ultimately reduce the costs caused by defects, and lead to customer satisfaction. [6]

Quality can be enhanced by doing quality management. Procedures in quality management include quality assurance, quality planning and quality control. Quality assurance aims to establish policies and standards that lead to high quality software. Quality planning means choosing policies and standards and adjusting them to different software projects. Quality control contains defining and approving processes that ensure the use of these policies and standards. [1; 6]

Some risk management strategies and techniques include software testing, technical reviews, peer reviews, and compliance verification. [6]

Verification and validation are part of the quality assurance process. Verification is proving that product meets the requirements specified during previous activities, and it is done throughout the development life cycle. Validation confirms that the system meets the customer requirements at the end of life cycle. Traditionally, software testing has been considered a validation process, a life cycle phase that is carried out after programming is completed. Verification should be combined with testing so that testing occurs throughout the development process. Verification includes systematic procedures of review, analysis, and testing employed throughout the software development life cycle, beginning with software requirements phase and continuing through the coding phase. Verification ensures the quality of the software production and maintenance. Verification emerged as a result of the aerospace industry's need for extremely reliable software in systems in which an error in a program could cause mission failure and result in enormous time and financial setbacks, or even life-threatening situations. The concept of verification includes two fundamental criteria: the software must adequately and correctly perform all intended functions, and the software must not perform any function that either by itself or in combination with other function can degrade the performance of the entire system. The overall goal of verification is to ensure that each software product developed throughout the software life cycle meets the customer's needs and objectives as specified in the software requirements document. A comprehensive verification effort ensures that all software performance and quality requirements in the specification are adequately tested and that the test results can be repeated after changes are installed. Verification is a "continuous improvement process" and has no definite termination. With an effective verification program, there is typically a four-to-one reduction in defects in the installed system, which reduces the costs of the system, even though the initial costs may be greater than without verification. Error corrections can cost 20 to 100 times more during operations and maintenance than during design. [6]

Quality control is defined as processes and methods used to monitor work and observe whether requirements are met. It focuses on reviews and removal of defects before shipment of products. Quality control consists of well-defined checks on a product that are specified in the product quality assurance plan. For software products, quality control typically includes specification reviews, inspections of code and documents, and checks for user deliverables. Usually, document and product inspections are conducted at each life cycle milestone to demonstrate that the items produced satisfy the criteria specified by the software quality assurance plan. Inspections are independent examinations to assess compliance with some stated criteria. Peers and subject matter experts review specifications and engineering work products to identify defects and suggest improvements. Inspections are used to examine the software project for adherence to the written project rules. They are kept at a project's milestones and at other times as deemed necessary by the project leader or the software quality assurance personnel. An inspection may be a detailed checklist of assessing compliance or a brief checklist to determine the existence of such deliverables as documentation. Responsibility for inspections is stated in the software quality assurance plan. For small projects, the project leader or the department's quality coordinator can perform the inspections. For large projects, a member of the software quality assurance group may lead an inspection performed by an audit team. Following the inspection, project personnel are assigned to correct the problems on a specific schedule. [6]

Quality control is designed to detect and correct defects, whereas quality assurance is oriented toward preventing them. Detection implies flaws in the processes that are supposed to produce defect-free products and services. Quality assurance is a managerial function that prevents problems by heading them off, and by advising restraint and redirection. [6]

Quality management, measuring quality and low-quality prevention tasks require resources. The total cost of effective quality management is the sum of four component costs: prevention, inspection, internal failure, and external failure. Prevention costs consist of actions taken to prevent defects from occurring in the first place, for example quality planning, code reviewing, testing tools and training. Inspection costs consist of measuring, evaluating, and auditing products or services for conformance to standards and specifications. Internal failure costs are those incurred in fixing defective products before they are delivered. External failure costs consist of the costs of defects discovered after the product has been released. Low quality causes additional work, correction work and possible refunds. [1; 6].

Making low quality code can be justifiable in some circumstances, for example, before a release, when schedule is tight. These parts should be refactored as soon as possible so that the software quality does not start degrading. Developers should be aware of traits in low quality code, how it affects the program, and how and why low-quality code is produced. With this knowledge developers can make good decisions, and accomplish objectives set for the project and quality. The amount of low quality code should be calculated. It helps to have examples of low quality code to identify problem areas and device a plan to correct them. The amount of low quality code should be monitored and reduced by refactoring from time to time. The amount of low quality code should be managed and made sure that it does not grow. [2]

3.2 Metrics

Quality metrics are used to measure software attributes. The purpose of metrics is to give indications on quality of the software components and the system in general, so that the low-quality areas can be improved, and the quality of the software monitored. Metrics can be used to estimate, for example, if the quality level has changed, how large a refactoring process is going to be, where to focus on during refactoring, and if refactoring has improved quality. The success of the refactoring process can be evaluated by using the same metric before and after refactoring. Before using any metric, it is important to have an objective. Measured attributes are then chosen according to the objective. For example, the number of errors found, and lines of code can be measured, when the objective is to reduce the number of errors in code. [1; 10]

Metrics can be used to measure two types of attributes: measurable and quality attributes. Measurable attributes do not depend on any other attributes, so they can be measured directly. Some examples of measurable attributes are lines of code, number of subroutine calls and external coupling meaning the number of references to outside. The result is a numerical value that can be used to conclude the level of the attribute. The result can also be used in a formula used to calculate a quality attribute or some other more complex type of attribute. Quality attributes are measured through measurable attributes. Quality attributes usually depend on other quality attributes and they can also be overlapping or inclusive. Generally, they promote each other. For example, having good testability makes the software more likely to be also reusable, portable and flexible. First the measurable components of a quality attribute have to be defined. They are selected based on what kind of attributes are being aimed for. [1]

There are some important points to take into account when defining metrics. Metrics should be simple and easily calculated. Measuring them should be fast and easy and the process should be easy to learn. The meaning of the metric should be intuitive for example the complexity grows when the metric result grows. The results should be unified and objective, different measurers should get the same result. Metrics should also be independent of the programming language used. [1] Some examples of metrics are lines of code and cyclomatic complexity. Having less lines of code means that there should also be less defects and duplicate code. Cyclomatic complexity means how many unique paths there are in a unit. The greater the number the more complex the unit is. [10]

The measuring process is defined in [1] like the following:

1. Choose a goal for the measuring and evaluation.
2. Choose quality attributes based on the aim.
3. Choose measurable attributes based on the quality attributes.
4. Choose the parts of code to measure: most critical parts, presentable group of parts or something similar.
5. Measure the selected parts using metric tools.
6. Evaluate the results: compare the results to earlier results and verify them.
7. Prepare for enhancement activities. Based on the results what kind of enhancements are needed and how large they are. What parts are the most important.

It is also possible to return to the first step and select different metrics if needed.

IMPORTANT METRICS

Metrics are useful for keeping the code clean and they give objective measures that can be used during code reviews to point out design and style flaws. When style flaws are pointed out by the tools, it is easier to focus on more important algorithm and design matters. Metrics benefit the most when they are measured automatically during builds and when they are available for developers working on their desktop, so they can check them before committing code into version control. [14]

Some useful metrics to measure are duplication, lack of adherence to a specific standard, and other language specific violations, for example modifying a parameter that has not been defined as out parameter. All of the metrics are difficult to notice for developers, but easy for a tool to measure. Duplication cannot be detected simply by comparing strings, because it is likely that developer has changed some variables names in the copied code. The scan should tokenize the code, and compare the tokens, not individual lines. [14]

Unused and commented code is easy to find for a tool and therefore a good metric to automate. Tools can find unused imports and method calls, which would be difficult to notice otherwise. This will keep the code as small as possible, which will prevent defects. Unused code will have to be maintained with rest of the system, which uses resources that could be used for more important tasks. If the code is not maintained, it will cause problems, when a new developer accidentally starts using it again. All old unused code should be fetchable from version control, so there is no need to keep it in the current version system. There it can be fetched again, if it is needed in the future. [14]

Cyclomatic complexity is very useful metric, when measuring code quality, because complex parts of the code usually contain most defects and are difficult to change. When one defect is corrected, another will surface. At first much of the code will be defined as too complex, but maintenance and new features will be easier to implement, when complex parts are made simpler. [14]

Code test coverage measures how unit tests are exercising the code. It provides visibility into how much of the code is being unit tested. This is useful, because the code that does not have unit tests probably contains the most defects. It is difficult to have 100% code coverage on all modules especially, when the project is legacy code. For that kind of code 80% coverage is sufficient, after that increasing the coverage will become increasingly difficult and inefficient. [14]

If the program contains threading, inconsistent mutual exclusion is a very useful metric to have. It is a scan that produces a report that informs how much of the access to an object is synchronized. This can prevent defects that are otherwise difficult to find and reproduce. [14]

3.3 Important quality attributes

Defining good structure is difficult, but there are some attributes that are usually found in good quality code, and some that are characteristic to low quality code. [14]

Good quality code is error free, and does not contain many defects that affect the user. The code structure is flexible. Some things that make it inflexible is duplication, because when a correction is done to one part of the duplicated code, it has to be also done to other duplicated parts. [14]

Complexity makes the code difficult to change and understand. It also makes it error prone, and the errors are hard to find, because of the complexity. Close coupling means that dependent code is tightly relate to how the dependency is implemented, and therefore change more often, when the dependency changes. The implementor of the dependent should only know, how to use the dependency, not why it works. The number of dependencies should be minimized, and they should point to the right direction. Details should be independently changeable and they should be dependent on the needs of the overall program structure. Dependencies should flow from the most general to most specific pieces. [14]

Good quality code is expressive and easy to understand. Its variables and methods have clear names. Developers should divide their work into small working pieces and check them in often. This will make merging the code from different developers easier and help them to divide the implementation into small manageable pieces, which can be easily written and validated. [14]

Unit tested code has a regression harness to inform the developer if their changes have broken any existing functionality, which makes them more reliable. They may also have fewer dependencies if they have been developed keeping unit tests in mind. [14]

3.4 Good quality code principles and patterns

There are patterns and principles that are designed for making quality code. Using them can improve the quality of the code.

Don’t repeat yourself (DRY) principle is designed to minimize the amount of code which is important especially for legacy projects. DRY means that the same code should not exist in multiple places in the software. [15]

SOLID principles are a set of five principles to address problems that can arise with object oriented programming. The mnemonic acronym SOLID stands for Single responsibility principle (SRP), open / closed principle (OCP), Liskov substitution principle (LSP), interface segregation principle (ISP), and dependency inversion principle (DIP). The principles are presented below and their explanation is based on [15].

The SOLID are recommended by [10] and [15]. It is commonly perceived that SOLID principles and TDD compliment each other [10; 15].

3.4.1 Single responsibility principle

First of the SOLID principles is single responsibility principle. It defines that "A class should have only one reason to change". When a class has to change, it means that it has to be rebuilt, tested and deployed, which all take resources. Changes always introduce the risk of introducing defects. A responsibility is a task that the class is responsible for, and a reason for a class to change. Having multiple responsibilities in one class usually makes the responsibilities coupled. Making changes to another may impair or inhibit other responsibilities. This kind of coupling leads to fragile designs that break in unexpected ways when changed. To discover responsibilities in a class, it is useful to think whether the class has more than one reason to change. If it does, it contains more than one responsibility. Sometimes it is justifiable to keep two responsibilities together, if they always change together. In that situation separating them would bring needless complexity into the program. Test-driven development can help discover responsibilities that need to be separated. [15]

3.4.2 Open / Closed principle

Open / Closed principle defines that "Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification". The design is rigid, if a change to a class causes changes to dependent modules. OCP advises us to refactor the system so that further changes of that kind will not cause more modifications. If OCP is applied well, further changes of that kind are achieved by adding new code, not by changing old code that already works. Modules that conform to OCP has two primary attributes: [15]

1. They are open for extension. This means that the behavior of the module can be extended. As the requirements of the application change, we can extend the module with new behaviors that satisfy those changes. In other words, we are able to change what the module does.
2. They are closed for modification. Extending the behavior of a module does not result in changes to the source, or binary, code of the module. The binary

> executable version of the module whether in a linkable library, a DLL, or a .EXE file remains untouched.

This can be achieved with abstraction. With it is possible to contain fixed yet unbounded group of possible behaviors. The abstractions are abstract base classes, and the unbounded group of possible behavior are represented by all the possible derivative classes. A module that depends on an abstraction is closed for modification, since it depends on an abstraction that is fixed. Yet the behavior of that module can be extended by creating new derivatives of the abstraction. This can be achieved by using interfaces and inheritance. No design can be 100% closed, there will always be changes that require the module to change. Therefore, the designer must strategically choose the most likely changes against which to close the design. Conforming to OCP increases the complexity of the design, because of the abstraction and it takes resources to implement all the derivants. Therefore, it is recommended that it is used only after changes that require it are needed. OCP offers benefits of object-oriented design: flexibility, reusability and maintainability, since changes are closed inside a class and need to be changed only there, and the class can be derived easily. [15]

3.4.3 Liskov substitution principle

Liskov substitution principle states that “Subtypes must be substitutable for their base types”. It addresses class hierarchy rules, what kind of hierarchies to create and what to avoid. Hierarchy is often considered to be a is-a relationship, but in practice it does not guarantee that one class can be derived from another. Is-a relationship should be considered in terms of behavior. A class can be derived from another if it behaves like the base class. LSP states that derived class must adhere to restrictions of the base class. This means that the derived class can replace preconditions of its base class with equal or weaker than those of the base class. The postconditions can be replaced by equal or stronger than those of the base class. Weaker meaning that all conditions of the base class are not implemented, and stronger meaning that, in addition to conditions of the base class, conditions can be added. The derived class must accept any input that the base class accepts. The output of the derived class has to conform to all constraints established for the base class. When considering whether a particular design is appropriate, one must view it in terms of the reasonable assumptions made by the users of that design. Test-driven development (TDD), which states that the tests should be written first, can be good tool to find these assumptions. If the code using a derived class has to check its type, or if the derived class removes some functionality of the base class, the hierarchy does not conform to LSP. Anticipating all user assumptions is impossible, therefore also this principle should be used after the need has arisen. Only most obvious assumptions should be implemented at first. [15]

3.4.4 Dependency inversion principle

Dependency inversion principle is twofold, it defines that [15]

1. High-level modules should not depend on low-level modules. Both should depend on abstractions.
2. Abstractions should not depend upon details. Details should depend upon abstractions.

The dependency structure of a well-designed object-oriented program is “inverted” with respect to the dependency structure that normally results from traditional procedural methods. It is the high-level modules that contain important business logic, and therefore details should depend on them. The important business logic should not have to change when lower implementation details change, risking introducing defects on the process and making the logic non-reusable. Inverting dependencies makes the high-level modules reusable. The low-level modules are usually reused in programs in the form of subroutine libraries, but the reusability of higher modules is not thought about. Reuse is possible if high-level modules depend on abstractions, interfaces, instead of concrete classes. In fact, these interfaces should be defined with the client, not the implementing module, because in DIP interface is the property of the client and should only change when the client changes. If there are many clients, they should agree on a service interface and publish in a separate package. Another interpretation of DIP is that no class should depend on a concrete class. All relationships in a program should terminate on an abstract class or an interface. [15]

1. No variable should hold a reference to a concrete class.
2. No class should derive from a concrete class.
3. No method should override an implemented method of any of its base classes.

This heuristic is usually violated at least once, when the instances of the concrete classes are created. This heuristic should be used, when classes are prone to change, which most of the developer written classes are. When an interface of a volatile class must change, the change affects also the abstract interface, which will then affect clients. Therefore, it is a better option to define interfaces with the client instead of the implementing class. DIP is needed for the creation of reusable frameworks. It is also important for the construction of code that is resilient to change. Since abstractions and details are isolated from each other, the code is much easier to maintain. [15]

3.4.5 Interface segregation principle

Interface segregation principle states that non-cohesive interfaces should be broken up into groups of methods that serve a different set of clients. There are classes that require non-cohesive interfaces, but client should not have to know about them as a single class. Instead, client should know about abstract base classes that have cohesive interfaces. When an interface contains methods that do not belong there, some classes implementing it have to provide degenerated implementations to some of the methods, which potentially violates LSP. Those classes will potentially have to import definitions not needed by them, which introduces needless complexity and redundancy to the code. Sometimes users will require changes on the interface, and if the interface does not conform to ISP it will affect all the users of the interface. This creates coupling between the clients as well. Two ways to implement ISP is: [15]

1. To create a delegate which inherits and implements an interface. Now, the users of the original class do not have to change, when that interface changes. Delegate requires a little extra memory and resources, and the pattern should be used only when translation is needed between two objects or different translations are needed at different times in the system.
2. To inherit from multiple interfaces or abstract classes. The users can then use the interface they need. This is considered to be the better alternative.

Like with all principles, also this principle should not be overused. [15]

3.4.6 Dependency injection pattern

In unit testing, injection of double objects is important, because we want to test the logic in the unit under test not the dependencies. Dependency injection pattern helps to decouple dependencies from the classes using them. In the pattern dependencies are passed, or injected, through parameters to the class rather than the class creating or finding them. Dependency injection supports DIP. There are three ways to inject a dependency

1. Receive an interface at the constructor level and save it in a field for later use.
2. Receive an interface as a property get or set and save it in a field for later use.
3. Receive an interface just before the call in the method under test using
    a. a parameter to the method (parameter injection)
    b. a factory class
    c. a local factory method
    d. variations on the preceding techniques

When interface is received at the constructor level, the object is passed as parameter to constructor method. The constructor then sets the received parameter to a local field to be used later in the program. This will make the dependencies non-optional, and the user will have to send in arguments for any specific dependencies that are needed. Having too many dependencies as parameters can make the code complex and more difficult to read. Inversion of control (IoC) frameworks can help with injecting dependencies. They provide mappings from interfaces to implementations that can be used automatically, when creating an instance of an object. Many non-optional dependencies can also make testing more difficult, because the test setup has to be changed when a parameter is added to the constructor. Constructor parameter is a good choice when the dependency is not optional, because it forces the user to give it. [8]

Dependency can also be gotten through a property, when the user sets the property. This means the dependency is optional or it has a default instance that is used if the dependency has not been set by the user. [8]

The dependency can also be gotten just before it is used in the code. This can be through a parameter of the method, when the dependency is passed from the test code to the code under test. Other way to get the dependency is through a factory. The code under test will get the dependency by calling a method of the factory class. The factory should have set and reset functionality to enable replacing dependencies it provides. Another way is to get the dependency through factory method in the tested class itself. To make it replaceable, the factory method has to be declared as virtual and then overridden in a class that inherits the class under test and is then used to test the code under test. This is a simple and understandable way to replace dependencies. It can be used, when a new constructor parameter or interface is not a good option. It can be more difficult to create a derived class than passing a double, because it may not be clear what dependencies need to be overridden. [10]

4. GOOD UNIT TESTING

Unit testing is the process of testing single subroutines, functions or classes [16]. There are differing opinions in literature [3; 7-10; 16] and in companies [17], whether unit tests should to be executed in a testing harness, isolated from the other system (e.g. databases, filesystem and web services), or in a complete or almost-complete system environment. In Test-driven development and according to many authors today, unit tests should be completely isolated from the other system including isolation from referenced classes [3; 8-10; 16]. Unit tests are the lowest level tests, that are the first ones to catch faults in the code. This makes them important part of software testing.

4.1 Purpose of unit testing

There are multiple reasons for unit testing. Unit tests offer a good regression safety-net for the software, help in designing and implementing the software and they may also help to find defects.

Unit testing can help in designing the software. A testable design usually follows object-oriented principles. Tests point out design issues and enforce principles that are part of an object-oriented design, for example, SOLID principles. Implementing unit tests during or straight after the coding process ensure that the software has been designed to be modular. A modular software has better testability, maintainability and reusability. Unit tests are done only one unit at a time. This enables developer to focus on one element at the time, which also makes designing tests easier to manage. Writing test makes the developer the first user of the unit, which can help in designing a clear API. [5; 16]

Each unit test represents a requirement or a specification. Therefore, unit tests are the most accurate specification of the unit's behavior. These tests offer a way to compare the function of the software to unit's specification. Passing tests show that the software functions as expected, according to some customer requirements. Other developers can also look at the tests and know how the unit is supposed to work. [7; 9; 16]

Unit tests are good for regression testing, they show when functionality of the software has changed. They make refactoring possible by providing a regression safety-net. They also reduce the number of failing higher level tests, because unit related regression and defects are found early. Early found defects are cheaper and easier to correct, and manual testing has to be done less. [7; 9; 18]

4.2 Characteristics and quality attributes of good unit tests

Unit tests are usually written using a unit test framework [7] for example Nunit [19] or Microsoft Unit Test Framework [20]. Unit test consists of four parts:

1. Setup
2. Act
3. Assert
4. Teardown

Setup contains the creation and initialization of objects. Data structures and environment variables are also initialized there if used. Second, in the act part, the initialized object is used to test a certain requirement. Third, in the assert part, the state or output of the tested object is asserted to see if the result is what was expected. If the assumption made of the result is correct, the test passes otherwise the test fails. The fourth teardown part is optional. There the used environment variables or objects are reseted or cleared. [8; 10]

The following characteristics of good unit tests are combined from [10], [8] and [3]:

- Isolated
- Focused
- Automated and repeatable
- Predictable
- Should run fast and be easy to run
- Easy to implement

Most of the characteristics affect multiple quality attributes, and complement each other. [8] defines the following important quality attributes for unit tests:

- Trustworthiness
- Maintainability
- Readability

These attributes contain characteristics. Some of them are related to multiple quality attributes. Isolation and focus are characteristics that affect all three of the quality attributes.

According to [3], unit tests should be isolated from other system and environment for three reasons. Testing only one class in isolation makes it easier to locate the source of the failure, because the piece of code executed is small. Separating the test from its environment allows the test to run fast. Short execution time is important, because unit tests should be run often to notice regression as soon as it emerges and to make localizing defects easy. Unit tests should not use databases, communicate across network, use file system, or do anything else environment related like editing configuration files, because these operations are slow.

Unit tests are supposed to test all the functionalities of the unit and cover the logic widely, which means that the number of tests to write is large. Therefore, it is important that the tests are easy to write. [8] Tests are easier to write when the code under test is small and isolated, because it is easier to see the connection from input values to the logic that is tested [3]. [9] says that the architecture has to support isolation by providing ways to replace referenced classes. A poor architecture not allowing separation can this way lower the quality of the tests.

[8] emphasizes that test isolation means separation from other tests. Separating the tests makes them more reliable. A unit test should be independent of other tests and their in-memory state and external resources. In-memory state should be set to expected state before test in a setup method or by calling specific helper methods. To avoid having shared state problems, a new instances of the class under test should be used in every test when possible. The state of static instances should be reseted in setup or teardown methods of the tests or by calling a helper method within the test. If singletons are used, there should be an internal or public setter so that tests can reset them to a clean object instance. A test should not call other tests or require a certain run order to be in an expected state.

Isolating tests from external resources and unpredictable data makes the test reliable [10]. This is important because developers have to be able to trust that the unit test results are accurate [7].

According to [7], a unit can contain a few simple classes, if the tests are still fast and do not use a database or other external resources. [10] and [3] advice against testing multiple classes at a time. [10] stresses that when a unit test fails, it should be obvious in which method or part of the code the defect is. If there are multiple classes being tested at the same time, localizing the defect becomes more difficult. Whereas finding the defect is trivial, when tests are properly isolated. [3] considers that when dependencies are allowed in unit test, the test dependency chain tends to grow and it will be more difficult to separate the classes, when time passes and the code has grown. Multiple dependencies will also make the test slow.

A focused tests test only one thing and contains only one assertion. The test is easy to name and the cause for failure easy to locate, because instead of one large test there will be results from multiple focused tests to give their information on the defect. The tests will be more trustworthy, when the localization of the error is easier. If there is a need to assert multiple properties of an object, assertion can be used to compare full objects instead of multiple assertions. This will make the test more readable, because it is easier to understand that one logical block is being tested instead of many separate tests. [8] According to [10], focused tests and the methods under test are more likely to be following SPR and are therefore of better quality and more readable, because it is easier to know what the tests are testing.

Trustworthiness is important for unit tests, because developers should be able to run them frequently to verify the current state of the software. Developers do not want to run tests that are not reliable and fast. Therefore, it is also important to separate unit tests to their own project from integration tests that access, for example, filesystem and other services not in developer's control. Trustworthy tests have no defects and they test the right functionality of the object. Trustworthiness can be achieved when logic is avoided in unit tests, and by using TDD and writing the tests before code. Failing tests have to be deleted or changed. To ensure correctness, the tests should be peer reviewed preferably by the writer and a reviewer face to face. According to [8], trustworthy tests can be written by following the rules below:

- Decide when to remove or change tests.
- Avoid test logic.
- Make tests easy to run.
- Assure code coverage.

When a unit test has been written, it should generally not be changed or removed. If the test fails, it should be a sign that there is a defect in the production code and the code under test has to be corrected. Still, there are situations when tests have to be changed. It is important to know how and when to change or remove a test. A test should be changed or removed when:

- Test contains a defect.
- Semantics or API change in the code under test.
- Conflicting or invalid new test.
- Renaming or refactoring the test.
- Removing duplicate test.

When a defect is found in a test, it is important to make sure that the test is now defect free. Following steps should be used when correcting a failure:

1. Correct the defect in the test.
2. Make sure the test fails when it should.
3. Make sure the test passes when it should.

After correcting the defect in the test, a defect should be introduced in the production code to make sure that the test catches the defect it should. Then the defect in production code should be removed again, and the test should be passing. If it does not pass, there is still a defect in the test and step 1 is resumed.

Unit tests may have to be changed if unit's semantics change and therefore the way to use the class under test changes. In these situations, it helps that the tests are as maintainable as possible and use common utility functions to initialize the object under test. This way the semantic change hopefully needs to be done to only a few places.

A new requirement may conflict with an existing one and an old test starts failing, when the feature is implemented. It is important to notice that neither of the tests is wrong, but the requirements conflict. It then has to be decided, which requirement to keep, and the test verifying the unneeded requirement should be removed.

Renaming and refactoring tests should be done always, when low quality tests are discovered. Usually duplicate tests should be removed, because maintaining them will take extra resources.

Generally, tests with logic in them replace original simpler tests, and making it more difficult to find defects in the production code. Logic also increases the possibility of a defect in the test, and makes the test less readable, and more difficult to recreate. The test probably tests multiple things, and is therefore more difficult to read and name.

Code coverage tools can be used to measure how much of the code is covered by the tests. Additionally, the code can be inspected and defects introduced to it on purpose to verify if there are places that the tests do not cover. Missing tests should be added, when they are found.

[8] and [9] agree that maintainability is highly important in unit tests, because unmaintainable unit tests may jeopardize the project schedule. Unmaintainable unit tests take much time to maintain, because they have to be changed every time a small change is done to the code. Therefore, it is important that the tests are isolate. Developers will stop maintaining the tests and may disregard them, if they are of low quality and the schedule is tight [8].

According to [8], to achieve maintainable unit tests the following things should be considered:

- Test only public methods
- Remove duplication

Only public methods should be tested, because they are the only functionality that is interesting to the user of the unit. The private functionality is the class's internal functionality and may change often. If a method needs to be tested, it should be probably made public or at least internal. Making the method public will inform the other developers that the method has a known behavior or contract against the calling code, and the caller has to be considered when the method is changed. If the method cannot be declared public, it may be declared internal and then exposed only to the test project. The method can be extracted to a new class, if it can stand on its own, or it uses state in the class that's only relevant to the method in question. The class can then be tested separately.

Duplicate test cases should be removed, because duplication means that more code needs to be changed, when a change needs to be done. Duplication can be removed by three ways: using a helper method, setup method or parametrized tests. Tests can use helper methods, for example, to initialize variables used in the tests, assertion logic or calling out code in a special way. Automatically run setup can be used to initialize variables that are used in all tests, so it cannot be used extensively. Most unit test frameworks provide a way to run the same test with different parameters multiple times. Parameters are usually written above the test. The test is then run with different inputs, and the failed inputs are shown with the error message.

[8] says that readability may be the most important quality attribute of the three. Readability connects to trustworthiness and maintainability. The developers need to be able to understand what the tests do to maintain, use and trust them. The following points need to be considered, when making readable tests:

- Naming unit tests
- Naming variables
- Creating good assertion messages
- Separating assertion from actions

Naming standards are important, because they give the template that outlines what should be explained about the test. The name has three parts:

1. The name of the method being tested.
2. The scenario under which it is being tested.
3. The expected behavior when the scenario is invoked.

The name of the method is needed to locate the tests related to a certain method. The scenario part gives the constraints of the tests for example, the function is called with a null value. The expected behavior defines what the code under test should do or return based on the current scenario. With these three parts, the developers should be able to understand, what is being tested without reading the test code. The parts are usually separated with underscore.

It is important to name the variables and used constants well to let the reader know, what the value means. For example, error codes should be given a clear variable name instead of using pure numbers.

When writing custom assertion messages, it is best to not write one, if there is no special reason to do it. Automatic assertion messages are usually enough. Test name and test framework outputs should not be repeated. When custom assertion message is needed, it should contain information about what should have happened or what failed to happen, and possibly when it should have happened, for example, "Calling a certain function in certain scenario should have returned a certain value".

The method call should always be on a different line than the assertion. It makes the test more readable, and it is easier to notice which function was called with what values.

Unit tests should be fast and easy to run [3; 7; 8]. It should not take more than 1/100 second to pass for a unit test, because, in a large project, there can be numerous unit tests [3]. Fast tests offer quick feedback to the developer on the state of the software, and whether their changes have introduced failures to the system. The tests should be named so that, when they fail, it is easy to understand what was expected and what went wrong. [7] It is very important that the tests do not need configuration and are automated, because they need to be run very frequently [8].

Unit tests should always be included in the source control, so that the developers have easy access to them. The tests should have similar project structure to the main software. Every project in the software should have its own test project. Every class or functionality should have its own test class. The tests should be named clearly starting with the method name being tested. This way every test can easily be mapped to methods in the actual project. Unit tests should be separated from slower tests, so that they can be run fast easily. Having slower and less reliable tests with the unit tests may lead the developers starting to not run the unit tests as well, because reliability and speed problems the other tests have. [8]

[3] and [10] agree that because unit tests only can find defects inside the logic of a unit, other types of tests are also important. They make sure that the system works as a whole and that units can be integrated together [10]. They cover the interactions in an application and can be used to define behavior for a set of classes [3].

4.3 Unit testing methods

Testing methods are generally accepted testing principles. It is impossible to completely test a software, and it is not possible to use endless resources on testing. Therefore, it is important to find test cases that can find the most number of defects. Test cases chosen at random are the most ineffective option. It is recommended to use testing methods when designing test cases. Methods should be used to make unique test cases not to create overlapping cases. [16; 18] This is important to remember, especially when writing test cases for legacy projects, where the number of tests can be large even if all of the test cases are unique. Overlapping test take unnecessary resources to maintain.

White-box methods are mostly used in unit testing, but they should be complemented with black-box methods. White-box methods focus on software logic and test coverages White-box methods concentrate on how much of the unit's logic and execution paths are covered. It is difficult to achieve 100% coverage with more complex coverages, but usually less than 100% coverage is sufficient. Coverages are used to evaluate test data but they can also be used when evaluating the quality of the unit tests. [16]

To be able to focus on most critical parts of the software, before deciding how high coverages are needed, following things should be considered [18]:

1. Which parts of the software are very critical?
2. Which features and code parts are the most used?
3. Which units are the most complex?
4. Which units have gone through most changes?
5. Which units have produced the greatest number of failures?
6. What inputs the software should support?
7. What can be concluded base on the history of similar systems?

Statement coverage states the percentage of the lines being executed in a unit. 100% statement coverage means that all the lines in the unit are executed at least once during testing. Statement coverage is a weak coverage and does not tell much about the quality of testing [10]. Basically 100% statement coverage is the minimum requirement and should be complemented with other coverages [18]. Decision coverage also called as branch coverage states that each decision should take true on and false outcome at least once. This criterion ensures that every decision is executed, but it will not necessarily exercise every decision outcome. [18] Decision coverage is a fairly weak coverage, but it is better than statement coverage [16]. Condition coverage states that each condition in a decision should take on false and true outcomes. This coverage is usually stronger than decision coverage, but it generates a lot of test cases. Therefore, a test case generator tool can be used to generate them. Decision / condition coverage states that each condition and each decision should take on true and false outcomes at least once. [18] There are also stronger coverages that may need to be used when software is safety critical.

Only functional classes should be tested and included in the coverage calculations. Data storage classes do not need to be tested, because they contain no logic and are, therefore, not error-prone. [10]

Black-box methods are based on the specification of the software and aim to find areas where software does not work as it should. Testing all inputs is impossible so it is important to find inputs that find the greatest number of defects. Erroneous inputs usually find the greatest number of defects, because most failures occur when the software is used in some new way. Black-box methods include, for example, equivalence partitioning where all inputs are divided to their corresponding valid and invalid input groups. It can be assumed that input in the same group will cause the same functionality in the software and therefore only one input of the group has to be tested. Edge-pair analysis uses inputs just under and over acceptable input values. These values usually find more defects than other values. [16]. Edge-pair analysis is simple, inexpensive and effective way to make test cases [21].

4.4 Unit testing tools

Unit test frameworks offer a unified programming model for defining tests. Tests are usually defined as methods inside classes that call the system under test. Unit test frameworks usually offer a test runner, which enables developers to run all tests with one click. The test runner will then show the results, and information about failures and exceptions. Usually GUI is provided for running the tests and showing the results. Ease of use is very important in unit test frameworks, because tests should be run frequently. Test frameworks also provide a way to setup and teardown the environment quickly. [10]

A test framework is an important tool, which will be used frequently by the developers. Therefore, when selecting the framework for a project it is important to consider, if it offers everything that is needed. Does the framework have active community? How difficult is it to learn to use it? Does the project team have experience of other frameworks? [10]

Usually when writing unit tests, dependencies to other parts of the system are mocked away. This can be done with mock frameworks. Framework offers a set of APIs that make writing doubles simple and easier. Developers do not have to write code to simulate communication between objects. With doubles developers can control called modules completely, which enables them to concentrate on testing the unit under test. [8] Mocking frameworks can be used to make doubles easily. They offer a Domain-Specific Language (DSL), which can be used to give run-time directions for the mock. Usually mocking has builder restrictions and only objects that implement an interface can be mocked. Recently some frameworks [11; 12] have emerged that can also mock objects during run-time. These frameworks can mock implementations and may be useful in legacy projects. [8; 10]

Test generation tools can be used when testing legacy projects or to find weaknesses that have not been yet found. These tools should not be used alone, because the tests they generate may be very weak [21], but they offer good support when creating complete test sets. They can also be used as regression tests. [10]

Tools can be used to evaluate test set quality and completeness. Mutation analysis tools generate defects in to the tested code and measure how many of these seeded defects were caught by the tests. Coverage tools measure how much of the code is executed by tests. Good mutation tools can find defects even when branch and code coverage are 100%, which makes them a good addition, when measuring completeness of a test set [22].

4.5 Replacing dependencies with doubles

Test doubles are in a key role when doing unit testing. Unit tests are supposed to test only one unit's code, and therefore dependencies usually need to be replaced by doubles. Doubles replace part of the system by simulating them at run time. The interactions of the testable unit can be restricted and therefore its functions are deterministic and fast. This will also make locating the defect easier, because you can be sure that the unit's code is causing it and not the dependency. The double can be instructed to return values that can be difficult to reproduce without doubles, for example, exceptions. The tests are separated from each other, because they do not have to use, for example, the same database or filesystem. [8; 10; 23]

There are various types of doubles: dummies, fakes, stubs, mocks and spies. According to [8], a stub is a controllable replacement for an existing dependency in the system. By using stubs, code can be tested without dealing with dependencies directly. According to [23], stubs provide predetermined answers to the calls made during the test, usually not responding at all to anything outside what's programmed in for the test. Mocks, on the other hand, are used to test interactions between tested unit and the dependency. According to [8], a mock object is a double object in the system that decides whether the unit test has passed or failed. It does so by verifying whether the object under test interacted as expected with the fake object. There is usually no more than one mock per test. According to [23], mocks are preprogrammed with expectations that form a specification of the calls they are expected to receive. They can throw an exception if they receive a call they don't expect and are checked during verification to ensure they got all the calls they were expecting. [23] states the following about dummies, fakes and spies. Dummy objects are passed around but never actually used. Usually they are used to fill parameter lists, for example, passing null to constructor parameters. Fake objects actually have working implementations, but usually take some shortcut that makes them not suitable for production. A good example of this is the in-memory database. Spies are stubs that also record some information based on how they were called. One form of this might be an email service that records how many messages were sent. It can be used to verify values that the system under test passed to its dependency.

The type of double used depends on the situation. Usually stubs and fakes are enough and when using them, the caller does not have to know much about the dependency logic. Mocks can replicate more complex actions, for example, check whether calls were made in the right order. Using mocks extensively can make the code tightly coupled, because the caller has to know much about how to call the dependency. Spies can be used when the method under test lack normal output. [10]

There are different ways to inject stubs into the code. To increase testability, it is recommended to use dependency injection and interfaces. There are frameworks that promote dependency injection and do some of the injections automatically. Some examples of these framework are Castle Windsor [24] and Ninject [25]. These frameworks can be useful, when constructor has a large number of dependencies. When designing software only the interface should be taken into account, developer should not need to know about the implementation. When creating classes, the implementation should not show in the interface, but it should be abstracted away or mocking them may be difficult. Dependencies should be minimized, this way the software will be more robust, open for extension and it is easier and efficient to mock. Mocks have to be implemented properly even when it might return a complex data structure. Low quality double lowers the quality of the test, which again lowers the quality of the whole software. [8; 10]

4.6 Test-driven development

When considering improving the quality of a legacy project, it is important to not only consider correcting the old code but also to consider the quality of the new code that is added. If the quality of the new code is not high, the quality of the legacy project cannot be improved. Therefore, the new code should also have unit tests and they can be implemented, for example, by using TDD.

Test-driven development is a process for writing software. The first step is to write a failing test, then write enough code to make it pass, then refactor the code or the tests, then the process is repeated. [8]

Before starting, requirements need to be divided into small, simple, isolated and testable parts. Dividing requirements makes them easier to understand, manage and implement. Completing these small tasks will offer sense of accomplishment for the developers. Tests can be written in two ways: bottom-up or top-down. When using bottom-up method, tests are written from core up, which means that mocks are needed less at first and the core can be designed to be simple. This method is recommended for inexperienced team. When using top-down method, testing starts from UI down. Therefore, the interfaces have to be designed early, which can increase their quality. [10]

The first step, writing the test, should be done before anything else, even a reference to the tested library or a class being tested should not be added. Tests should also be done for other parts of the system, if they need to be changed in the process. Unit tests are isolated so it is not necessary to write implementation for called classes or functions immediately. Only an interface is needed for mocking. Also, the class under test is tested through an interface, so the developer implementing the class is the first user of the interface they have designed. This will increase the usability of the interface, because the developer has to take into consideration the user, and not settle for the easiest implementation. This way interfaces will be simpler and more focused. [8; 10] Interfaces should be defined in the same file as the implementing class to make the code more readable and the connection clearer. [10]

It is important that the test fails at first. If the test does not fail, it might mean that the test does not test what it is supposed to, the test uses a wrong functionality, the requirement is a duplicate or the feature has already been implemented. When implementing defect corrections, it is important to first write a test that reveals the defect. This way, the regression suite will contain a test that will prevent the defect from occurring again. [10]

When implementing new features, unit tests should be run regularly after any small changes. This way, if regression occurs, it is noticed immediately. If there are error messages from the compiler, they can be used to lead the development. Only the functionality that is being tested should be written, for example, if a test expects false from a method, it is first enough to write only one line of code to return false. Additional code should be added only, when tests require more complex functionality. This way code base will stay as clear as possible and only the currently needed business logic is implemented. The smaller the code base is, the less there is to maintain. The feature is ready when all the unit tests pass. [8; 10]

In the third phase, the code and tests are refactored to improve their quality, for example, by making them simpler. At first, business logic is implemented in the easiest way, because the most important thing is that the functionality works. Therefore, the code quality may be low at first and refactoring is needed. Also, additional tests can be added to improve the quality of the test set. It is important to write also “unhappy” tests with erroneous inputs and circumstances or the test set can give false impression on the quality of the software. TDD process itself promotes only happy test cases, so it is recommended to use testing methods to complete the test set. Test cases should use randomly generated data if possible, so that the code does not adapt to the test input. The tests prevent regression during implementation process, and enable fearless refactoring. [10]

TDD helps to make high quality code, tests and better design. As the result the code usually follows SOLID principles and dependency injection. The code is closely coupled with requirements, because they are written in the form of executable tests. Input and output need to be defined for the tests in an early phase, which encourages developers to communicate with the customer. Having focused tests make locating defects easy and debugger does not need to be used extensively, which makes debugging faster. [8; 10]

Important skills to have when using TDD are 1. writing maintainable, readable and reliable unit tests 2. writing test first 3. designing good and maintainable code [8]. Developers usually have problems with starting with a test and letting the test lead the development. It is also common that developers write more complex code that is needed and decorate the software with unneeded features that they think might be useful in the future. It is recommended that TDD is taken into use slowly, because there it takes time for the developers to learn all the aspects related to TDD and unit testing. Metrics should be used to measure quality before and after using TDD. [10]

4.7 Unit testing and impact on quality

Test-driven development has been known to increase code quality [26]. In studies [27-29] a great quality difference has not been found between code developed writing tests first and tests last. Therefore, it seems to be important that the tests are written, not when they are written. Developers using TDD tend to write many positive test cases even though negative test cases find more defects [27].

Unit tests by themselves do not guarantee code quality, but they can reveal quality problems. If a module is difficult to test, it usually means that the module has quality problems, for example it is too complex or too large and it has to be divided into more focused classes. Therefore, the process of writing tests itself is important and the tests are only a byproduct. Testability seems to correlate with other quality attributes like maintainability and complexity, so that testable modules are more maintainable and less complex. Testable module often follows SOLID principles. Unit tests also form a regression safety net, which is important when refactoring or new functionality is implemented. The tests help to make the program more reliable. [8]

4.8 How to start unit testing a legacy project

First, the developers have to be considered when unit tests are taken into use. In the end, they are the ones to implement the tests, and their skills and motivation affect heavily on the adaption of unit testing. There will be developers and managers that are accepting a new way of developing and others that are not. It is important to convince everyone or they may start to work against the change. It is easier to first start with a small team or a sub team with developers open to change. More inexperienced developers and those who have experience on working with low quality code may be more open to change. The project itself should be fairly simple and have low risks to allow developers to implement tests without time pressure. [8]

There are different ways to start the change. One is that unit testing is adopted in a team by the developers without support or pressure from the management. This requires that the developers themselves are interested and use their free time to learn the new skill. Another way is that the management supports the change and will introduce it to developers. The third way is to get an outside consultant to help with the change. With all of the ways it is important that the process is given time and that the developers have time to learn. At the beginning, there has to be a strong driving force to enable the change. A lack of a driving force is a common reason for unit testing to fail. Someone has to be there to help developers to learn. Therefore, it may be a good option to hire an outside consultant or give one of the employees possibility to concentrate fulltime on the task. The management has to give its consent and allow the developers extra time to adapt to the new style. The process should be allowed at least three months to let everyone adapt. [8]

It is good to make the whole process visible for the whole company, so that teams can compete each other in how well they have adopted the new style, and to raise the awareness of the other developers that are not yet using it. To make the progress visible it is important to have measurable goals. Some metrics can be code coverage and bug-fixing time. Code coverage will help the developers to write also unhappy test cases and bug-fix time will show the results of the good quality code. [8]

When unit tests are implemented in a legacy project, at first, the tests will find a large number of defects. When the defects are correct the number of defects will decrease considerably, because of the regression testing support that the tests offer. [18] Implemented tests can find even critical defects that have been in the system for a long time [5]. At first writing unit tests will slow down the project considerably, since developers have to learn how to use unit test framework and how to write unit tests. [7; 18] Writing comprehensive unit tests for a legacy project will take a long time and require good planning, but it is possible.

There are processes for implementing unit tests for a legacy project. According to [18], first it is important to identify the unit testing tools to use by trying them. The way unit test failures are reported should also be decided on. Secondly, the units with most failures should be identified by measuring the number of bug-related changes, provoked build fails, and the remaining life time of the unit. The most low-quality units should be then unit tested and their faults corrected. Lastly, the project management should agree that from some date on all the new and changed code will be unit tested. Development process should be revised to have a quality gate preventing code from being checked-in without unit tests and results. A report should be provided about the number of units that do not have unit tests or whose unit tests could not be run, the number of failures found and corrected before release and after it, the change in the number of failures found after release compared to prior number of failures found. The report will show how the unit tests have impacted on the error rate of the software.

[23] states that before changing or adding anything to a legacy software, it should have system level automate tests for its key functionality. The software can then be changed by adding unit tests in the process. If any of the automate tests fail, it should be the first-priority for all the developers to correct them at once.

The following unit testing process is adapted from the guidelines in [8].

1. Where to start?
   (a) Create a priority list of components for which testing makes the most sense. Give each component a rating for these factors, from 1 (low priority) to 10 (high priority). Based on the rating choose the components with high priority and complexity, and ignore the ones with low priority, complexity and high dependency level. The following factors affect component's priority
      (i) Logical complexity – The amount of logic in the component, such as nested ifs, switch cases or recursion. Tools for checking cyclomatic complexity can also be used.
      (ii) Dependency level – The number of dependencies in the component. How many dependencies do you have to break in order to bring this class under test?
      (iii) Priority – The component's general priority in the project.
   (b) When you have created the list of components to test, there are two basic ways to decide what you would like to test first.
      (i) Choose the one that is easier to test: Writing tests will be initially quicker and easier, testing will get harder at the end of the project, when schedule is usually tight. This way lets inexperienced developers to learn testing techniques before dealing with more complex components.
      (ii) Choose one that is harder to test: When one component is brought under test, it may solve testability problems for its dependencies. Therefore, the time required for implementing tests declines quickly. This strategy requires experience in unit-testing techniques.
2. Write integration tests, if you have to refactor your code for testability, so you can write unit tests.
   (a) Add one or more integration tests to the system to prove the original system works as needed.
   (b) Refactor or add a failing test for the feature you're trying to add to the system.

(c) Refactor and change the system in small chunks, and run the integration tests as often as you can, to see if you break something.
(d) Work on the parts of the system that you need to fix or add features to. Don't focus on the other parts.

4.9 Other types of automate tests

The other types of automate tests in addition to unit tests are integration and system tests, deployment tests, and acceptance tests. There are also some other types of tests that can be automated that measure the efficiency of the system, for example, stress test, but they are not introduced here. Again, the terms are not used entirely consistently in literature and practice. Acceptance tests are written by or with the customer, when other types of tests are written by developers or testers [30].

Integration tests are used to verify the links between units, for example that the units get the input and output they were expecting [7; 9]. Integration tests make sure that the units work together and therefore they should be implemented as soon as possible, because it is possible that the units do not work together properly, even if they work on their own. They are usually slower than unit tests and use third party service like database, filesystem and web services [7]. Integration tests are written by using the same frameworks as unit tests, but no parts of the system are mocked. The tests should be able to control their environment to some extent, for example they should use a test version of a database or a web service. They should be able to reset their environment, and give deterministic results. Data should be unique values that are randomly generated so that it does not matter, even if environment did not reset fully, for example after a database write. Integration tests can contain multiple test steps for example writing and reading in one test case. Unlike unit tests in test driven development, it is okay if they pass immediately when they are written. New functionality should not be implemented when writing integration tests, but it is okay to add code to for example dependency injection framework. Tests can be also written before writing implementation to ensure that all functionality is covered. This may be useful when new developers are implementing new functionality. Integration tests should be run throughout the project automatically. This way it is easy to notice which commit broke the build immediately when it happens. Integration test may be the only way to implement automated tests for a legacy project if the project has not been developed in a testable way. Any tests are better that no testing at all, so it is recommended to first write integration tests and then write unit tests during refactoring and new implementation. [10]

According to [7], there should be also special integration tests that test only links between third party services, database, web service or filesystem. There should be two versions of the test: one that uses the actual third-party service and one that uses a mock version. This way the actual link to third party service is also tested.

System tests are long and comprehensive integration tests that cover full functionalities or a use cases. Preferably the test should contain full call chain from UI to database. These tests are few in numbers when compared to unit and integration tests. [10]

Deployment tests are run after the system has been deployed, and taken into use by the customer. They make sure that the deployment was successful, for example, installation and configuration were done properly, deployed software can connect to services it uses, and it that it responds to requests. [7]

Automatic acceptance tests verify that full features work in the application. They are written by customers, business analysts, testers and quality assurance specialists. Developers can use them to understand what kind of functionality is required of the features. They are the current and executable specification of the features. Acceptance tests are usually written in special specification language to make them easy to write and read for nontechnical people like customers. [15]

5. MAINTENANCE

Maintenance contains four types: corrective maintenance meaning corrections to defects, adaptive maintenance meaning adapting software to its environment and hardware, perfective maintenance meaning implementing new features, and preventive maintenance, which means making software structure better without changing functionality. Preventive maintenance is very close to refactoring and it is done by first observing the code and then implementing improvements. [1; 3; 4; 10] Improvements are, for example, renaming methods and variables, moving them from class to another, and dividing classes [10]. During refactoring, automate tests are important to verify that the original functionality is preserved. Refactoring should be done in a disciplined manner in small steps to minimize the risk of introducing defects. Refactoring is done to make the code easier to understand and change. [3; 4]

There are five evolutionary laws that apply for software systems and maintenance. Law one is continuous change which states that a system has to change in order to stay useful, due to, for example, law changes and new requirements. The second law is increasing complexity, which states that when software is changed, its structure becomes more complex. Resources have to be used to maintain and to simplify the existing structure, which is done by refactoring. Refactoring lowers the costs of maintenance in the future. The third law is evolution of large systems, which states that the size, release frequency, and the number of reported defects stays the same between releases, because developers do not want to or are afraid to change large system, because of the risk of introducing defects and deteriorated functionality. Law five is development speed constancy which states that the development speed of a system stays constant regardless of the resources available. Large systems seem to be in a saturated state and their change speed stays constant when more persons are brought into the project. The fifth law states that the amount of modifications implemented between releases stays the same. If one release introduced much functionality, the next release will contain many repairs for defects introduced in the last release. Therefore, functionality should be increased steadily. [1]

5.1 Refactoring process

Refactoring is the act of improving design without changing its behavior. The software's structure is altered to make it more maintainable. During the process, we make a series of small structural modifications, supported by tests to make code easier to change. Key part in refactoring is to alter the code safely without changing the existing behavior. [3]

Refactoring has positive effects. One of them is that it makes the software easier to understand. Refactoring unfamiliar software can help in understanding the software better as a whole. The real structure of the program becomes visible, when the details are refactored away. This also helps to find defects in the code, when knowledge on the software increases and the structure improves. Good structure makes implementing new features fast. Refactoring keeps the structure whole. Automate tests enable refactoring and make it possible to choose a simple design at first, and change it later when need arises. This way less code has to be written, because it is not necessary to prepare for future changes. There is also less pressure when designing the software, because the first version does not have to support all possible future requirements. Automate tests and good quality code make refactoring feel reliable, but changing low quality code feels intimidating and it is done only if it is necessary. [4]

According to [1], the general process of refactoring has two or three phases. The first phase, that is optional, is an analysis step. Developer studies the code and tries to recognize its functionality and structure. The second phase is a change step. The found design is improved and planned for. The last phase is a build phase, where the planned improvements are implemented.

To avoid risks and regression the following questions should be considered: 1. What changes are required? 2. How can it be verified that they have been implemented correctly? 3. How can it be verified that regression has not occurred? [3]

According to [3], the standard way of refactoring in the industry is to implement a change riskily by just changing the code. First, the code is examined to find the part where the change should be made. Then, the change is implemented, and the developer manually tests the program in ad hoc way. The software is then tested a bit more. There is no much reliability in this way of testing, because it is normally not done structurally.

A better way to refactor is cover with tests and change approach. The refactored code is first covered with tests that verify its current functionality. Then the change is implemented. Because of the automate tests, developer can be fairly sure that regression is not introduced and the functionality remains the same. With the tests in place, it is also easy to understand what the software does and how it should work, by reading the tests. [3]

Unit tests give immediate results and are therefore important. They make refactoring safe. Integration tests can also be used if the unit is hard to get into test harness. It is common that first refactoring has to be done to be able to put the unit into test harness and implement tests. Simple safe refactoring can be done with tools or by hand when using extreme caution and discipline. Changes done to the code to get it under test may lower the quality of the code momentarily, but it is more important to get tests in place without taking risks. [3]

Legacy software refactoring process is the following according to [3]:

1. Identify change points
2. Find testable points
3. Break dependencies so that doubles can be inserted
4. Write tests
5. Implement changes and refactor.

This way, with every change, the quality of the software increases.

5.1.1 Identifying change points

During step 1 the code is examined to identify where the changes need to be done. There are techniques to use to help to understand the code better, for example, making notes and sketches. These can be in a modelling language or in free format. The more confusing the system structure feels, the more it helps to have formal notes on the structure. Other technique is to print out the part of code that you think needs to be changed, and use a pen to mark things on the paper, for example, group things with a marker to make seeing different responsibilities easier, mark code block starts and ends to make understanding method structure easier and circle code that you want to extract. Scratch refactoring can be used to gain knowledge on the system fast. The idea is to refactor the system freely to understand its underlying structure and functionality, and then scratch the changes and do the real refactoring. This approach contains two risks: developer can get wrong impression on the code, if they change its functionality accidentally without noticing. The developer can also get attached to the structure they created, which can limit their options, when deciding on the new structure of the code. Deleting unused code is a good way to make the code more readable. The deleted code can be gotten back from the version control, if it is needed in the future. There are techniques to uncover the underlying structure, by discussing the architecture with other developers or writing diagrams. [3]

5.1.2 Find testable points

In step two, places in the code where tests can be written are searched. Finding them can be difficult in legacy projects, because of dependencies and high coupling. The tests may be difficult to implement, and also integration level tests need to be implemented to ensure that the change has not produced defects on parts of the system. First, it is important to clarify, what parts of the system will be affected by the change. Effect sketches in a free format can be a good way to do this. When writing effect sketches, reasoning can be done backward and forward. When reasoning backward, we think what can affect the result of the function, where the change is made, and others calling it. When reasoning forward, we think the opposite, what the effects of the change are, where can we detect the change. The clients have to be considered. In the sketch, variables that can be affected and methods, whose return value can change are written inside bubbles. Next, arrows are written from the bubbles that may cause changes to the bubbles that are changed. Effects can propagate in three basic ways:

1. Return values that are used by a caller
2. Modification of objects passed as parameters that are used later
3. Modification of static or global data that is used later.

A good way to find effects:

1. Identify a method that will change.
2. If the method has a return value, look at its callers.
3. See if the method modifies any values. If it does, look at the methods that use those values and the methods that use those methods.
4. Make sure you look also for super classes and subclasses that might be users of these instance variables and methods.
5. Look at parameters to the methods. See if they or any objects that their methods return are used by the code that you want to change.
6. Look for global variables and static data that is modified in any of the methods you have identified.

These techniques will help in determining, which parts of the system will be affected by the changes and need tests. [3]

5.1.3 Break dependencies

It is usual that dependencies have to be broken before tests can be written. It may not be necessary to break all the dependencies, but integration tests can be written instead. With integration tests, it can be possible to test multiple methods or objects to enable their safe refactoring. First, you should find interception points in the code. Interception points are points in the program, where changes can be detected. The best way to find them is to start tracing effects forward and backward from the points, where you are going to make changes. The interception point should be as close as possible to the change point, because it is easiest to write tests that cover the change point, if there are not many extra steps between. When multiple classes need to be changed, it can be more efficient to pick a higher level interception point, and test multiple classes at once. The point where multiple changes can be detected is called a pinch point. When testing through pinch points, the tests are more difficult to write, but they cover a wide area of code. When using pinch points, it can be useful to make a change to the change point, and make sure that your test catches it. [3]

In steps 1 and 2, we found places where to write tests. In step 3, we break dependencies on those classes to make it possible to put them in a tests harness and write tests. Some frameworks can be used to mock implementations at run time like TypeMock [11] and JustMock [12]. If mocking is too difficult or the tools are not available, refactoring has to be done before dependencies can be broken. Because there are no tests, refactoring has to be done carefully. Refactoring tools can be used to make safe refactoring without tests. Extract method is one operation that is widely supported. It is usually possible to get the code into test harness with tool supported operations. When a tool cannot be used, doing one change at a time makes it easier to focus and make refactoring correctly. It is possible to introduce extra variables that enable sensing the state of the class. Then it may be possible to write tests that use the sensing variable to check the result, and refactoring can be done safely. When the refactoring is finished, the sensing variables and the tests can be removed or refactored to test the current class better, for example, they can then test new extracted methods. Another strategy is extracting small three or five line methods from the code and writing tests for them. When extracting methods, the coupling count should be minimized. Coupling count means the number of variables passed as input and output to the method. Their number should be minimized, because it is easy to make mistakes related to method parameters. The best way to see if dependencies need to be broken is to try to instantiate a class in test harness. The compiler will tell you what to do to make it instantiable. [3]

5.1.4 Write tests

In fourth step, the tests are written. When writing tests for a legacy project, it is important to write tests that characterizes the functionality the class has, not functionality it is supposed to have. This is because we want to preserve the existing functionality, not to find defects. Algorithm for writing characterization tests: [3]

1. Use a piece of code in a test harness.
2. Write an assertion that you know will fail.
3. Let the failure tell you what the behavior is.
4. Change the test so that its assertion part expects the result that the code produces.
5. Repeat.

The tests that seem to specify a defect are included in the test set, but marked as suspicious. Firstly, when writing tests for a method, tests are written until the developer is satisfied that they understand the behavior of the method being tested. Next, tests are written to detect problems that may surface when implementing intended changes. Tests are added until we feel confident that they will catch defects created during refactoring.

If we do not feel confident after writing all the tests, it may be better to implement only a part of the changes that were planned at first. When writing tests for classes, a good place to start is to try to think what the class does at a high level. It is easier to start with simple tests, and then move to more complex ones. Below are some heuristics that can help when writing characterizing tests for classes: [3]

1. Look for tangled pieces of logic. If you do not understand a piece of code, consider introducing a sensing variable to characterize it. Use sensing variables to make sure you execute particular areas of code.
2. As you discover the responsibilities of a class or method, stop to make a list of the things that can go wrong. See if you can formulate tests that trigger them.
3. Think about the inputs you are supplying to the code under test. What happens at extreme values?
4. Should any conditions be true at all times during the lifetime of the class? Often these are called invariants. Attempt to write tests to verify them. Often you might have to refactor to discover these conditions. If you do, refactoring often leads to new insight about how the code should be.

Important things about the class should be documented as tests. If you are attempting to extract or move functionality, write tests that verify the existence and connection of those behaviors on case by case basis. Verify that you are exercising the code that you are going to move and that is connected properly. Exercise conversions in tests. [3]

5.1.5 Implement changes and refactor

It is important to separate phases of refactoring and adding new functionality. During refactoring new tests or new functionality should not be written. When adding new functionality, the old code should not be changed. Tests should always be written before refactoring, because developers make mistakes even when they are careful. The tests are the only protection against regression, and they are the indicator whether defects have been introduced or not. An extensive test set should be created before starting, but it is important to remember that having a few tests is better than having none. Even if the developer does not feel that they are able to write a full test set, it should not prevent them for writing an incomplete one. Refactoring should be done in small steps, and tests run between them. This way it is easy to find and locate regression. The code should be so readable that there is no need for comments. Comments should be written only to give reasons for the decisions made, when not sure on how to implement a feature. Comments should be refactored to be visible in the code. [4]

5.2 When should refactoring be done

According to [4], refactoring should not have an assigned schedule but it should be done in small sprints constantly. Refactoring should be done, when old code is preventing easy implementation of a new feature or change, or when a method is too long or has too many arguments. It should be done when developers come across duplication the third time.

Refactoring should be done when implementing a new functionality. First refactoring can be used to make the code more understandable. Then refactoring should be done to make the feature easier to implement. If the old structure is preventing easy insertion of the new functionality, is should be first refactored and then the new functionality can be implemented. Descriptive variable and function names make the code more readable. [4]

Refactoring should be done when a defect is corrected. When code is refactored first to improve readability, the defect can surface in the process. Improved readability will prevent defects from occurring in the future. [4]

Refactoring can also be done during code reviews when two developers go through a piece of code that one of them wrote. Code reviews done face to face are effective in spreading knowledge through the project. Refactoring helps to understand other person’s code better. Reviewer proposes improvements and if they can be easily implemented, they are implemented during the review. Two persons in a review is ideal, when doing a code review. Larger groups should be used to assess structure with, for example, UML graphs. [4]

Rewriting code should be considered only, when the old code does not do what it is intended, and is full of defects. The code should be divided into components that have strong encapsulation, and then refactor or rewrite them one by one. Refactoring should not be done when deadline is approaching, but schedule problems might indicate that refactoring is needed to improve quality of the code. Developers should refactor regularly to maintain their refactoring skills, because refactoring becomes more difficult when not practiced regularly. [4]

5.3 Regression testing and validation

It is important to take into account the effects of maintenance to other parts of the system. It is easy to cause new defects that are difficult to find, because they exist in different parts of the system. [1]

Legacy software does not usually have modular and good structure, which is why changes can affect multiple parts of the system. First, positive tests should be written to verify that the existing functionality has not been changed. Then, more system tests can be written to test special cases, unwanted side effects, and normal error situations. Most of the defects occur in support code and not in the code implementing functionality. If only functionality code is changed without any modifications to the surrounding code, the tests do not need to be very comprehensive. Tests should be implemented only if they bring value. [7]

[3] stresses the importance of unit tests in regression testing, because of their fast feedback. When extensive unit tests are in place, the developer can be sure that they are changing only the functionality they meant to. System level tests are good for regression testing, but because of their long feedback time, they are insufficient for regression testing if used alone. With unit tests, it is certain which changes caused the failure, while with system level tests that are usually run only for nightly or weekly builds, the results are more inaccurate. It is difficult to know exactly, which commit caused the failure.

5.4 Refactoring tools

There are static and dynamic tools that can help in understanding the code better and to help decide what parts of the system require refactoring. There are tools that can calculate metrics, for example complexity, how deep is the control structure, and decoupling count, how many connections a class has. Documents can be generated from the code to visualize connections and structures. This can make the software easier to understand. Dynamic tools can monitor memory reservations and time usage during run time and identify efficiency problems. [1]

There are also tools that help in refactoring. They can be used, for example, to rename variables or functions, or to extract classes or methods. These tools save time and are safer than refactoring done manually. Some of them can be included in IDE and some are separate tools. Refactoring tools should guarantee that the refactoring actions they perform do not cause defects. If the tool cannot guarantee safety, these actions should not be used. Tools can help to bring the code into a more testable state safely. If refactoring can be done by using a tool, it should be done with it. Then tests are not needed before refactoring [3]

6. IMPROVING QUALITY OF SOFTWARE BY ENHANCING UNIT TESTING AND REFACTORING

Unit testing seems to be leading to a more higher quality product with higher testability, reliability, maintainability and reusability, but only if the unit tests are of high quality. High quality unit tests seem to uncover underlying design problems and support quality principles, which leads to higher quality designs. If the tests and doubles are done in an unmaintainable way, they will more likely hinder than help the development process. Legacy code usually has testability problems, which means that also refactoring has to be done before unit tests can be implemented properly. Refactoring without unit tests is error-prone which is why special techniques need to be used, if refactoring is done without them. With all the above things to take into account, taking unit tests into use in a legacy project is not easy and it has to be done in a planned way in iterations.

6.1 Enhancing unit test process

Eight matters were selected to be considered when enhancing unit tests process based on research in this thesis:

1. quality of unit test [8; 9]
2. ease of running [3; 7; 8]
3. unit testing methods [16; 18]
4. developer education [8]
5. safe refactoring [3; 4]
6. quality control [6; 14]
7. unit testing and mocking frameworks [10]
8. unit testing process [8; 23]

Some of them require more resources than others, and some are easier to take into consideration.

6.1.1 Quality of unit tests

Quality of unit tests is one of the most important matters to consider, because low quality unit tests can rather hinder than help the development process [8]. They do not bring much value to the project, because the regression support could be achieved by generated unit tests [10], and the unit tests make changing functionality difficult, because they will break, when a small change is made [8]. This will result in the developers not maintaining them [8]. Constantly failing tests will then make noticing new failing tests more difficult. According to research in this thesis good quality unit tests can be achieved, when these simple rules are followed:

1. Have only one assertion per test. [8]
2. Mock all dependencies. [3; 10]
3. Use TDD when writing tests. [8]
4. Use a commonly agreed naming for all tests. [8]
5. Separate test parts clearly in the test. [8]
6. Use common methods to eliminate duplicate code. [8]
7. No logic in test cases. [8]

Even though good quality in unit tests is very important, it is not very difficult to achieve. If developers in the project are aware of the above rules, and every commit is checked based on them merging the commit to version control. If these rules are not already followed, it is recommended that they should be taken into use in the unit test process.

6.1.2 Ease of running the tests

It is also important that the tests are run by developers. Therefore, the tests should be easy to run [3; 7; 8] or even automatically run during development and on version control server. Unit tests should also be clearly separated from other types of tests, so they can be run easily without the slower tests [8].

6.1.3 Unit testing methods

When writing test cases, testing methods should be used [16; 18]. In legacy projects the resources are usually limited, and therefore the importance of the class should be taken into consideration when deciding how extensive test set a unit needs. Core functionality should be tested more extensively than less important functionality. [18] Equivalence partitioning is a good way to narrow down all possible inputs. This will make it easier for the developers to choose what inputs to use in their tests. [16] The edge-pair analysis is also known for being cost effective [21]. Coverages can be used to assess the completeness of the test set.

6.1.4 Developer education

The quality of the unit tests and how well they are written ultimately depend on the developers. This is why the people and their skills are the most important part of the unit testing process. The initiative to take unit testing into use will most likely have to come from the management to be widely adopted in the company. The key is to find developers who are enthusiastic and willing to learn unit testing, and to find a suitable project for them to try unit testing in practice. It is important that the developers have the opportunity to write unit tests in practice, because that is the only way their skills can improve. It might be beneficial to give one of the team members a full-time task to work on helping others, monitoring the tests, and making sure unit testing is done. Lack of time and drive is a common reason for unit testing to fail, and therefore is important that there is a person responsible for taking the project forward. When the team members feel they know enough, they could be rotated to different projects to teach others. In new teams the purpose and benefits of unit tests has to be discussed, and it has to be made sure that everyone is willing to participate. If this is not done, some project members may start to work against the change, because they think that unit testing may affect their work negatively. To summarize, these are the most important things to remember when introducing unit testing to developers: [8]

1. To have a strong driving force behind unit testing: a person to help and monitor
2. To have a project with loose enough a schedule to allow developers to learn unit testing in practice
3. Make sure everyone is willing to participate before starting

6.1.5 Safe refactoring

Developers also need to be aware of safe refactoring and how to write tests for refactoring. Therefore, they need to be educated on refactoring too. Developers have many parts to learn: TDD, unit testing, refactoring, and how to use unit testing, refactoring and other supportive tools. It may be best to introduce these parts gradually to give developers time to learn each part before moving to next.

Before unit tests can be written, there is usually refactoring to be done [3]. Before developers have acquired sufficient skills on safe refactoring techniques, it may be most resource effective to just cover the most important functionality with integration tests before refactoring, and correct regression when it surfaces [7]. Developers should learn to refactor in small steps, trace their change backwards and forwards, and to use safe refactoring techniques when possible. Characterization tests should be written whenever it is possible before refactoring. Refactoring should be done constantly to maintain the program’s quality. [3]

6.1.6 Quality control

To achieve sufficient quality, the quality of the code and the unit tests has to be monitored [6]. Monitoring can be done by peer reviews and automatically measured metrics [1; 10]. Some metrics to consider are adherence to a selected coding standard, code complexity, and code coverage of the tests [14]. Metrics can be very useful especially at the start of unit testing project, when the developers do not have a good view of quality. Metrics can be used to point out quality problems, which can be discussed in code reviews. It is good to have an automated measurement for adherence to a standard, because when syntax problems are pointed out automatically, developers can focus on structural problems instead. [14] The metrics used should be carefully selected, simple, and few in numbers [1] or interpreting the results can become cumbersome. Measuring metrics does not require much resources after it has been automated. The results of the unit tests and metrics should be easily visible to developers. Metrics can also be made internally public, and shown for example near the coffee machine to advertise unit testing and its benefits on quality [8]. This can also encourage different project groups to compete against each other on quality, which may lead to good results. Unit tests should be run automatically and their status monitored [1]. Commits should not be merged to version control, if unit tests do not pass [7]. To summarize, below are some points to consider about monitoring quality

1. Select few simple metrics and automate measuring them. [1; 10]
2. Use the results before and in peer reviews. [14]
3. In peer reviews focus on structure and technologies instead of syntax (syntax is taken care by metrics). [14]
4. Make metric and unit test visible to developers. [8]

6.1.7 Unit testing and mocking frameworks

Before unit testing is started, the right unit testing and mocking framework need to be selected. The tools will be an important part of the developer's work during the project, so different tools should be tested and studied before deciding. At the moment, there are differences, for example, in what kind of assertions and test cases frameworks support. Also, there are differences in how easy to use the mocking frameworks are.

6.1.8 Unit testing process

Unit testing can be taken into use in two ways:

1. unit testing on the side of development [3; 8; 23]
2. extra refactoring project [8]

The way number one is recommended even if way number 2 is done as well. Unit testing on the side of development means that unit testing is done always when implementing new or changing old code [3]. This way does not waste resources on potentially unnecessary testing. The second option, extra refactoring project, means that unit tests are implemented to some core units to increase their quality and reduce the number of defects they contain. First, the modules to be unit tested have to be selected. The modules have to be prioritized by some suitable attributes, for example, their importance for the system, complexity and defect count. After the modules have been prioritized some suitable number of them are selected from the start of the list. Next, the order in which to test the selected modules has to be decided on. If the developers do not have much experience on unit testing, the easiest module with the smallest number of dependencies should be selected first. This way the developers can gain experience before testing the more difficult modules. The down-side to this method is that unit testing gets more difficult towards the end of the project, which is usually busy. If the developers are experienced, they can start testing from the most difficult module. This causes the testing to get easier towards the end of the project, because breaking the dependencies of one module makes testing its dependencies easier too. Refactoring project can be especially helpful if the system has core modules that contain a large number of defects. Refactoring them will reduce the number of defects and make correcting them and making other changes easier. [8]

6.2 Evaluation

This thesis suggested some basic principles on how to improve and maintain good quality in a program by using safe refactoring and unit testing. In the future, these principles should be extended with other types of tests to guarantee more complete regression safety net. Manual and some form of system level tests would catch also problems in the collaboration of the units and UI that are difficult to catch in unit tests. Additionally, the discussion on how to select a suitable unit testing and mocking frameworks could be broadened. These principles have not yet been adopted in a real world legacy project, which would be the next step to validate them.

In this thesis, problems related to legacy projects and their solutions were searched from literature. Most of the solutions and problems were based on books from authors that promote unit testing and TDD, thus their opinions were very similar. On the other hand, there seems to be a number of literature that shares views on how to carry out good unit testing, on the other hand there may be difficulties related to unit testing that are missed, because the authors’ aim is to promote it. The opinions on the internet opposing unit testing seemed to be tightly related to the quality problems in unit tests, introduced in this thesis, or the lack of developer skills. In the literature found from sources used in this thesis, there was no content opposing unit testing. Opposing opinions seemed to be in blog posts found with search engines. This could mean that when unit testing is done well, there are no great problems related to it. Still, lack of skill and quality in tests seems to be a problem in many projects, and therefore great consideration should be put into educating developers and quality control, when unit testing is taken into use.

7. CONCLUSIONS

The goal of this thesis was to provide suggestions on how to improve and maintain the quality of a program safely by refactoring, unit tests, and quality control. Literature review was done, and based on the solutions found, suggestions were given on unit test quality, unit test methods, safe refactoring techniques, quality control, developer education, and unit testing process.

It was found that to achieve good unit test quality a few principles have to be followed. Among these principles were that a unit test should contain only one assertion, be named based on a common naming standard, and contain no logic. The unit tests should also be fast and easy to run, because they should be run often.

Test methods should be used due to limited resources. Equivalence partitioning can be used to narrow down inputs, edge-pair analysis is known to find defects and being cost-effective. Coverages can be used to find missing test cases and assess the completeness of the test set. Unit test frameworks and mocking framework are important tools for the developers, and they should be selected carefully, because at the moment there are differences in their usability.

The quality of the project depends on its developers, and therefore they should be educated on quality code patterns and principles, unit testing, and refactoring. It should also be made sure that they want to participate in any new quality related projects. Otherwise they may start working against the change. There should be a mentor to help the developers with unit testing when it is taken into use.

This thesis examined problems and solutions related to unit testing and refactoring in legacy projects, and found that refactoring, quality control and good unit tests increase the quality of the software. It was noticed that especially quality in unit tests has a great impact on the success of unit testing. Based on the findings, some principles were proposed on how to write good quality unit tests that benefit the quality of the project. Additionally, ways to refactor safely, practice quality control, and start unit testing were proposed.

Refactoring has to be done safely either by utilizing simple safe refactoring techniques or accompanied with tests. The tests can be integration tests or unit tests, which ever is easier to write. It is also possible to write integration tests to only the core functionality, and correct defects when they appear. It is important that refactoring is done constantly to maintain the quality of the code.

Code and test quality should be monitored with automatically measured metrics and peer reviews. The selected metrics should be simple and few to make them light to use.

Unit testing should be taken into use by starting to test all changed and new functionality from a certain time on. Additionally, a refactoring project can be executed where most important parts of the software are refactored and unit tested to improve their quality.

Following the above principles should increase and maintain the quality of a software. In the future, it would be interesting to see them used in a real world project.

REFERENCES

[1] M. Harsu, Ohjelmien ylläpito ja uudistaminen, 1st ed. Talentum Media Oy ja Maarit Harsu, 2003, 292 p.

[2] Suryanarayana, Girish, G. Samarthyam, T. Sharma, *Refactoring for Software Design Smells: Managing Technical Debt*, Morgan Kaufmann Publishers, 2015, 259 p.

[3] M.C. Feathers, Working Effectively with Legacy Code, 1st ed. Wait, John, United States, 2004, 434 p.

[4] M. Fowler, K. Beck, J. Brant, W. Odyke, D. Robets, Refactoring: Improving the Design of Existing Code, 1st ed. Addison-Wesley Professional, 1999, 464 p.

[5] C. Klammer, A. Kern, Writing unit tests: It's now or never! Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on, pp. 1-4.

[6] W.E. Lewis, Software Testing and Continuous Quality Improvement, Third Edition, 3rd ed. Auerbach Publications, 2009, 704 p.

[7] P.M. Duvall, S. Matyas, A. Glover, Continuous Integration: Improving Software Quality and Reducing Risk, Addison-Wesley Professional, 2007.

[8] R. Osherove, The Art of Unit Testing, Second Edition, with examples in C#, 2013.

[9] S. Sanderson, Writing Great Unit Tests: Best and Worst Practices, web page. Available (accessed 05/08): blog.stevensanderson.com/2009/08/24/writing-great-unit-tests-best-and-worst-practises/.

[10] J. Bender, J. McWherter, Professional Test-Driven Development with C#: Developing Real World Applications with TDD, Wrox Press, 2011, 360 p.

[11] Typemock Inc, TypeMock, Typemock Inc, web page. Available (accessed 19/02/2017): https://www.typemock.com/.

[12] Progress Software Corporation, JustMock, web page. Available (accessed 20/02/2017): http://www.telerik.com/products/mocking.aspx.

[13] P. Runeson, A survey of unit testing practices, IEEE Software, Vol. 23, No. 4, 2006, pp. 22-29.

[14] B. Holtsnider, T. Wheeler, G. Stragand, J. Gee, Agile Development & Business Goals: The Six Week Solution, Morgan Kaufmann Publishers, 2010.

[15] R.C. Martin, M. Martin, Agile Principles, Patterns, and Practices in C#, Prentice Hall, 2006, 768 p.

[16] G.J. Myers, C. Sandler, T. Badgett, The Art of Software Testing, Third Edition, 2012.

[17] P. Runeson, A survey of unit testing practices, IEEE Software, Vol. 23, No. 4, 2006, pp. 22-29.

[18] P. Farrell-Vinay, Manage Software Testing, 1st ed. Auerbach Publications, New York, 2008, 600 p.

[19] NUnit, What Is NUnit?, web page. Available (accessed 2017/02/04): https://www.nunit.org.

[20] Microsoft, Writing Unit Tests for the .NET Framework with the Microsoft Unit Test Framework for Managed Code, web page. Available (accessed 2017/02/04): https://msdn.microsoft.com/en-us/library/hh598960.aspx.

[21] S. Wang, J. Offutt, Comparison of Unit-Level Automated Test Generation Tools, Software Testing, Verification and Validation Workshops, 2009. ICSTW '09. International Conference on, pp. 210-219.

[22] R. Ramler, T. Kaspar, Applicability and benefits of mutation analysis as an aid for unit testing, Computing and Convergence Technology (ICCCT), 2012 7th International Conference on, pp. 920-925.

[23] J. Humble, D. Farley, Continuous Delivery
ReliableSoftware Releases through Build, Test, and Deployment Automation, Addison-Wesley, Indiana, 2011, 442 p.

[24] Castle Project, Windsor Castle Project, Castle Project, web page. Available (accessed 03/04/2017): http://www.castleproject.org/projects/windsor/.

[25] Enkari Lean Software, Ninject, Enkari Lean Software, web page. Available (accessed 03/04/2017): http://www.ninject.org/.

[26] A. Causevic, D. Sundmark, S. Punnekkat, Factors limiting industrial adoption of test driven development: A systematic review, pp. 337-346.

[27] A. Cauevic, S. Punnekkat, D. Sundmark, Quality of Testing in Test Driven Development, Quality of Information and Communications Technology (QUATIC), 2012 Eighth International Conference on the, pp. 266-271.

[28] H. Erdogmus, M. Morisio, M. Torchiano, On the Effectiveness of the Test-First Approach to Programming, IEEE Transactions on Software Engineering, Vol. 31, No. 3, 2005, pp. 226-237.

[29] Divya Prakash Shrivastava, R. C. Jain, Unit test case design metrics in test driven development, Communications, Computing and Control Applications (CCCA), 2011 International Conference on, pp. 1-6.

[30] Software Testing Class, Difference between System testing and Acceptance Testing, web page. Available (accessed 19/02): http://www.softwaretestingclass.com/difference-between-system-testing-and-acceptance-testing/.