LUT University

School of Business and Management

Degree Program in Computer Science

Simo Pöysä

Integrating unit testing into an organization with legacy ASP.NET applications

Examiners: Professor Kari Smolander

M.Sc. (Tech.) Timo Storhammar

Instructors: Professor Kari Smolander

M.Sc. (Tech.) Timo Storhammar

TIIVISTELMÄ

LUT -yliopisto

School of Business and Management

Tietotekniikan koulutusohjelma

Simo Pöysä

Yksikkötestauksen integrointi organizaation ja olemassa oleviin ASP.NET aplikaatioihin

Diplomityö

49 sivua, 5 kuvaa, 3 taulukkoa

Työn tarkastajat: Professori Kari Smolander

Diplomi-insinööri Timo Storhammar

Hakusanat: yksikkötestaus, yksikkötesti, perinteinen järjestelmä, ketterä, ohjelmistokehitys

Keywords: unit testing, unit test, legacy system, agile, software development

Tämä diplomityö tarjoaa tutkimuksen yksikkötestauksesta käyttäen “Design Science” -tutkimusmenetelmää. Tutkimus selittää mitä yksikkötestaus on, miksi se on tärkeää, kuinka yksikkötestejä tehdään ja mitä asioita yksikkötesteihin liittyy. Työ sisältää myös tapaustutkimuksen, jossa keskustellaan yksikkötestauksen aiheuttamista muutoksista ohjelmistokehitysprosessiin sekä ongelmista ja ratkaisuista yksikkötestien lisäämisestä olemassa olevaan applikaatioon.

ABSTRACT

LUT University

School of Business and Management

Degree Program in Computer Science

Simo Pöysä

Integrating unit testing into an organization with legacy ASP.NET applications

Master's thesis

49 pages, 5 figures, 3 tables

Examiners: Professor Kari Smolander

M.Sc. (Tech.) Timo Storhammar

Keywords: unit testing, unit test, legacy system, agile, software development

The thesis conducts research on unit testing using Design Science Research Methodology. The research explains unit testing, what is it, why is it important, how unit tests are done and what is related to unit testing. After the research, a case study of implementing unit testing into an organization with legacy ASP.NET applications will be reported. The case study discusses required changes to the current development process as well as the problems and solutions of implementing unit tests to the legacy applications.

ACKNOWLEDGEMENTS

I would like to thank my instructor, professor Kari Smolander for his guidance and fast responses regarding the thesis. I also want to thank my girlfriend Anja for proof-reading the work and her general support. Last but not least, thanks to all who helped with the unit testing integration in the company.

Table of contents

LIST OF ABBREVIATIONS

TDD	Test-Driven Development
DSRM	Design Science Research Methodology
AAA	Arrange, Act and Assert
CUT	Code Under Test
DI	Dependency Injection

1 INTRODUCTION

This thesis focuses on the challenge of applying unit testing to an organization and its legacy ASP.NET applications. According to Myers et al. there are three main points for doing unit testing. First point is testing is easy to manage as each of the tests are focused on small sections of the code. Secondly, unit testing makes debugging easier as it is faster and more accurate to find the error spots. This also means unit testing helps to prevent errors at earlier stage and less bugs end up in the production code. Final point is multiple units are being tested simultaneously and the tests are very fast because of their small size (Myers, et al., 2012).

Unit tests are by nature very fast so they can test big parts of the system with ease. Unit testing also forces to produce shorter methods/functions. By having short methods/functions, it is much easier to read the code and therefore easier to understand and maintain the whole system (Bowes, et al., 2017).

For software development, it is fairly common to develop technical debt over time. The idea of technical debt is the product or features have deadlines and by implementing things the fast/easy way usually causes extra work later on. The extra work comes from the found bugs or the difficulty of adding new features or making changes to the solution is higher so it takes longer (Holvitie, et al., 2014). Technical debt can be intended or unintended. As previously mentioned, one way is when deadlines are closing up and the release of the product or feature has to be done in haste. This means usually unclean code and probably the testing was also neglected to some extent. The unintended technical debt can come from flawed design by not realizing to take some parts or possible effects of changes into consideration (Holvitie, et al., 2014). Other way for unintended technical debt to develop is by overtime as the application grows bigger and more complicated. This leads to hard to maintainable system and causes the need for refactoring (Holvitie, et al., 2014).

Making unit testing a part of the development process, risk of getting unintentional technical debt will be reduced. As mentioned earlier, unit testing helps preventing/finding errors already at early stage so less bugs will end up in the production code (Myers, et al., 2012). Second important point is the maintainability of the system. For example, it is much easier to read and understand 60 lines long functions than 1000 lines long. If you are not familiar with the huge function you need to work with beforehand and you need to make changes to

it, it will take a lot more time to understand it than a group of smaller functions with descriptive names. Unit testing forces you to create smaller function as the point is to test small units of code.

1.1 Background

Unit testing is fairly commonly used in agile development. Some even say that it is not truly Agile Development if you are not using automated unit testing. The main noticeable benefit from unit testing is that it results in less bugs, they are easy and fast to locate and it is faster to get a working code. In reality, there can still be some bugs even after the release of new features. However, unit testing helps to prevent bugs at earlier phase so it can save a lot of time and money. Unit tests also help to show the precise locations of the problems faster. (Klammer & Kern, 2015)

The idea for this thesis came from the need and the wish for wanting to do unit testing in the organization and to produce more testable and maintainable code. The case company has not really done much unit testing in the past. Now, the plan is to change that and implement unit testing as part of the development process. It is much harder to start using unit tests with legacy applications than it would be when starting fresh. The difficult part about adding unit tests afterwards is refactoring of the code so that it can be tested. The system has a lot of dependencies for example to database so this needs changes. Also the hundreds of lines long functions are hard to unit test so they need to be divided into smaller pieces. This is why the company wants to research how it can be integrated to the whole development process and also how to implement unit tests to the existing applications. The main aim is to find the proper way of implementing unit tests and providing unit testing principles instructions for the team. More about goals and aims will be explained in the next chapter.

1.2 Goals and limitations

There are two main problems to be solved. **The first** is to find out how to integrate unit testing into an organization. How does it affect the people in different roles and their tasks? Do all developers write their own tests or will there be a common template or code snippet that supports doing it and so on. It is also important to figure out what needs to be changed and improved in the current development process. Does it also affect how the specification, design and other process phases should be done? After gathering enough information from

the literature, a case study will be made for the company. Case study handles implementing unit testing to the organization and implementing unit tests to the applications.

As already mentioned, **the second** problem is to solve how can unit tests be integrated for legacy ASP.NET applications. Is major refactoring needed or will it be okay with minor changes? As the research is limited to ASP.NET applications, it means that integration to other kinds of solutions and frameworks might be quite different. There will also be some code examples based on what was learned from the case study.

Simply put, **the goals** of this project are to create a plan how to apply unit testing to the organization, create instructions and plan applying unit tests to the legacy code. After finishing this project, the company should have a good idea on how to integrate unit testing to the development process. There should also be some real tests applied to at least one of the applications. Applying unit tests fully to the existing applications would take a lot of time which is not the goal of this project as the time is limited. Having a system full of unit tests is not very realistic either and would be very hard to cover it 100 %. **Primary goal is to provide instructions to the team on how to do unit tests.** Test-driven development (TDD) **is not** a part of this project as the two are separate things. Unit testing and TDD are often mentioned together which might cause confusion.

The values for the company from this research are benefits the unit testing brings and will be discussed more in details in the Unit testing chapter. Also worth mentioning is that integrating unit testing to an organization and legacy applications is a big project of its own. So far the company has not had the resources to actually implement it. This is why this research is important for the company.

Based on the research problems and goals, **the following research questions** were formed:

1. How to integrate unit testing into an organization?
2. How do software development process phases need to be changed for integrating unit tests?
3. How to integrate unit tests to legacy ASP.NET applications?

1.3 Research method

This project will follow the structure of Design Science Research Methodology (DSRM) as introduced in paper by Peffers et al. (Peffers, et al., 2007). Based on the paper, the structure of the project is divided into six main parts:

1. Problem identification and motivation
2. Define the objectives for a solution.
3. Design and development.
4. Demonstration.
5. Evaluation
6. Communication (Peffers, et al., 2007).

In the paper it is stated this may not always be the actual order some researcher would go with and that is completely fine. However, this project will mainly follow the order listed above. These six phases will be discussed below.

1.3.1 Problem identification and motivation

The problems and goals of this research are defined in the Background and Goals and limitation chapters. The motivations for the company are the benefits that come from achieving the goals. As a personal motivation, this project allows learning how unit testing works in an organization and how to actually do them. Both are interesting and useful.

1.3.2 Objectives for a solution

As already mentioned in the Goals and limitations chapter, the objective of this project is to implement unit testing into the organization. Second objective is to apply some working unit tests to the legacy ASP.NET applications to get the main idea how to even do them. This will create a foundation for the future to keep using unit test alongside the new code and to keep coding in a way that supports unit testing.

1.3.3 Design and development

The project is mainly focused on a case study. Before the case study, literature will be used first to explain in more detail what is unit testing, its benefits and how to create them. The literature was retrieved from three well known scientific literature databases: IEEE Xplore, ScienceDirect and ACM. The reason for choosing these three is they all have good-quality papers. The reason for choosing three different databases is that they most likely some papers that are not published in the other two. Only papers written in English will be searched and

most of them published from 2010. If some very relevant ones come across that are older, those will be taken in as reference as well. The references of chosen articles are checked. Related articles might be considered for further inspection. References mainly consists of scientific articles, books and websites.

The case study is practical part of the project. It includes a report about the steps taken when integrating unit testing into the development process. The report should answer to what steps were taken, what were the challenges and how were they overcome. This phase also shows the planning and preparation for implementing the unit tests to the legacy application.

1.3.4 Demonstration

Demonstration of the project are the practical implementation results of the case study showing how to apply unit tests to the legacy ASP.NET applications. The demonstration provides more technical details related to programming and reports about the challenges and successes.

1.3.5 Evaluation

In evaluation part the success of the case study is evaluated. Were there problems or challenges? How much effort did it take for the team to learn how to use and make unit tests? Is it worth start using unit tests with legacy applications? Are there other similar studies which were conducted before? How does the future seem like for the company regarding unit testing? All this will be taken into account in the Discussion chapter.

1.3.6 Communication

The following is the structure of the research. First unit testing is explained in chapter 2. After reading chapter 2, the reader should understand what unit testing is, what are its benefits, how to do unit tests, the best practices, what are the useful frameworks and what are the challenges of it. After unit testing chapter follows the case study in chapter 3. Case study chapter plans and solutions for integrating unit testing to the development process and also implementation of unit tests into the legacy ASP.NET MVC application. The case study chapter shows the problems encountered when applying unit tests to the legacy application and the solutions with examples. After the case study chapter, the Discussion chapter 4. evaluates the conducted research and Conclusion chapter 5. offers a summary of the research.

2 UNIT TESTING

Testing is an important part of development process and it has basically two main objectives: 1. To see if software meets the set requirements. 2. To make sure the software does not have defects and it is working the right way (Sommerville, 2011). Unit testing applies to both definitions. These two kinds of unit tests are also known as happy and sad tests which basically means verifying the system or trying to break it down (Bowes, et al., 2017). Simply put, a unit test is a code of its own that tests a small part (unit) of a system. It is up to the creator to decide on the size of a unit for the unit test. It can be only one function or a method or it can be an entire class or several classes (Osherove, 2013). However, the smaller the size of the unit, the easier it is to write tests for it. Usually one test is made for one method/function.

Quite often the articles about unit testing also mention TDD. The main idea of TDD is to create tests before the implementation. In practice this means you first create a test that will fail. The idea is to prove that the new functionality is missing. Second step is to create the functionality so it passes the test. Finally, the created functionality can be refactored to be even more readable. These three phases are also known as Red, Green, Refactor -phases (Osherove, 2013).

Based on different sources, it would seem more effective to use the red-green light method and write the tests before doing the actual implementation code. In Table 1 we can see the results of different kinds of literature reviews on TDD. For most of the time it seems that quality is always improved. On the other hand, we can also see that it has negative effect on productivity or the results were inconclusive (Karac & Turhan, 2018).

Table 1. Systematic literature reviews on TDD (Karac & Turhan, 2018).

Study	Overall conclusion for quality with TDD	Overall conclusion for productivity with TDD	Inconsistent results in the study categories
Bissi et al.[3]	Improvement	Inconclusive	Productivity: Academic vs. industrial setting
Munir et al.[4]	Improvement or no difference	Degradation or no difference	Quality: • Low vs. high rigor • Low vs. high relevance Productivity: • Low vs. high rigor • Low vs. high relevance
Rafique and Mišić[5]	Improvement	Inconclusive	Quality: Waterfall vs. iterative test-last Productivity: • Waterfall vs. iterative test-last • Academic vs. industrial
Turhan et al.[6] and Shull et al.[1]	Improvement	Inconclusive	Quality: • Among controlled experiments • Among studies with high rigor Productivity: • Among pilot studies • Controlled experiments vs. industrial case studies • Among studies with high rigor
Kollanus[7]	Improvement	Degradation	Quality: • Among academic studies • Among semi-industrial studies
Siniaalto[8]	Improvement	Inconclusive	Productivity: • Among academic studies • Among semi-industrial studies

One benefit of TTD is that in this way of development the test itself will be tested. If the test succeeds right away, then something is wrong with the test. The new functionality should not already be there; it should always fail first. However, TDD also has a problem of the learning curve. In the beginning, it takes time to learn and get used to a certain way of developing which means it will take longer to finish with the tasks (Osherove, 2013).

As in this research the goal is to apply unit tests for legacy application, we will not start applying TDD to the existing development process. The reason is that unit testing already has a big learning curve and especially the required code refactoring takes a lot of time. It might be a good idea to consider using TDD in the future once the team has successfully adapted the ways of unit testing and refactoring.

2.1 Benefits from unit testing

Why is unit testing useful? Firstly, unit tests work as a safety net against regression defects. Regression defects are defects that come up after making changes to the application. In this

sense it is safer to make changes to the production code (Reese, 2018) (Bowes, et al., 2017). With unit tests the code becomes more testable and the quality of code is higher.

Another benefit of unit tests is that they are executed very fast, within milliseconds (more about the qualities/principles of unit tests in chapter 2.3.). This means the functionality testing will be much faster than a person manually clicking through the program. A lot of time can be lost if the new features are crashing because of simple mistakes: for example, not checking for null values and dealing with them. When these kind of mistakes are taken care of already at earlier stages, it reduces time used for reporting errors and returning a task back to the developer. This also means the software tester can focus more on logical tests without having to worry about the system crashing because of simple mistakes (Reese, 2018).

Unit tests are also a good documentation of what different components do. This is especially the case when the unit tests are given descriptive names, which is a part of the good coding practices. Good naming improves readability and understanding of the code (Reese, 2018) (Bowes, et al., 2017).

Another good benefit of unit tests is that if the defects are spotted at an early stage, it will be easier and cheaper to fix them (Ganesan, et al., 2013). If the defects are noticed when the code is already used in production, it will take more time to fix and therefore produce extra costs.

2.2 How to do unit tests

Usually unit testing consists of three main phases: **Arrange**, **Act** and **Assert** (AAA). The **Arrange** phase means the objects that will be inserted to the Code Under Test (CUT) will be initialized and given values. For example, a parameter for a function. The **Act** phase is where the selected unit is invoked. In practice this means calling the function. The **Assert** phase is the final phase of the unit test in which the results of the test are checked and verified. This phase will tell if the unit test was successful or it failed. If a test fails, then the CUT is not working correctly and it needs to be fixed to pass the test (Osherove, 2013).

Unit tests should have their own controllers with test methods that separate them from the normal code. Simple example of a unit test would be a function (unit for test) is supposed to sum two numbers. In unit test first (**Arrange**) the parameters (the numbers) are decided and

the class to be tested is initialized. This example case is so simple that we can skip defining parameter names and instead just give the numbers to the function. In second phase (**Act**) the numbers are inserted to the function Sum and the function is called. Lastly (**Assert**), the return value of Sum-function is inspected if it calculated as planned. Code for the example can be seen below with xUnit.net-framework used in .NET. The xUnit.net-framework contains Assert-class which has several useful methods that can be used for making different kinds of comparisons for the result value (Osherove, 2013). In the example code below, there is first a test function and then the selected function which is to be tested.

xUnit.net example

```
[Test]
public void Sum_TwoNumbers_SumOfNumbers()
{
	var calculator = new Calculator();
	int result = calculator.Sum(1, 2);
	Assert.Equal(3, result);
}

public class Calculator
{
	public int Sum(int x, int y)
	{
		return x + y;
	}
}
```

In the example above, everything works fine as long as the numbers are integers. If the test would be changed so that it tests numbers with decimals, it would result in failure and the function under test would need changes to work properly.

2.3 Unit test principles and best practices

Unit tests have certain basic principles that should be followed:

- Easy to implement.
- Fast execution.
- Isolated from everything else. Can be run on its own and is not dependent on external factors/components.
- Consistent and repeatable. If nothing is changed, then the end result should always be the same.
- The test can automatically detect if it passed or failed.
- It is easy to pinpoint the problem spots in case the test fails.

(Osherove, 2013) (Reese, 2018) (Bowes, et al., 2017) (Ganesan, et al., 2013)

All characteristics above are required for a good unit test. How about the best practices of making unit tests? First of all, when programming, it is always a good idea to name the parameters, functions etc. so it is easy to understand what is their purpose. This makes it easier for maintaining and understanding what is happening in the code. It is known among programmers that sometimes there are confusing and hard-to-understand variable or function names that do not quite describe the real purpose. By following the good naming principles, the code itself works as a documentation. There are different kinds of naming standards. One example for naming is to name the test method in the following way:

> *"-The name of the method being tested.*
>
> *-The scenario under which it's being tested.*
>
> *-The expected behavior when the scenario is invoked."* (Reese, 2018)

For demonstration purposes, the test function was named like this in chapter 2.1. The proper naming should also be done for the used variables.

Another guideline is to code the tests according to the AAA pattern as explained in chapter 2.1. This also makes it easier to read and maintain the test code. Another good practice is to make the tests as simple as possible so it is easy to make changes at a later stage. Keeping the tests simple also prevents possible bugs in the test function/method itself. This means that if possible, it is best to avoid using logic in tests, such as loops and conditions. It is also better to have only one Assert-call per one test function. The reason for this is if Assert fails, the test function execution stops, which means that if there were other Assert calls in the same function, they will be skipped (Reese, 2018). Furthermore, when there are more tests with only one Assert instead of one test with lots of asserts, it is a lot faster and easier to see why something stopped working after changing the CUT (Dohnert, 2018). More specifically, the attention should be focused on the number of asserted objects (Aniche, et al., 2013).

Final note on good practices is the private methods or function should not be invoked with test functions. Instead they will be validated by invoking the public methods/functions that call the private ones. The reason is that the public method/function might manipulate the values from private method/function. (Reese, 2018)

2.4 Unit testing frameworks

There are several existing unit test frameworks. For .NET the three most popular ones at the moment seem to be:

- Visual Studio Unit Testing –Framework (also known as MSTest)
- NUnit
- xUnit.net

xUnit.net is the newest and was made based on NUnit from the same creator (xUnit.net, 2019). Do not confuse xUnit.net and xUnit as they are two different things. xUnit.net is a unit testing framework and xUnit is a common term used for identifying a framework that belongs in the same test automation framework used to automate hand scripted tests (Meszaros, 2007). This means the members of xUnit family usually share the following:

7. Provides a way to specify tests as test methods.
8. Has Assert methods to inspect the return values of CUT (this is the core part of each unit test as in this part the actual test is done by verifying the result).
9. Possible to run tests as a single operation.
10. Gives a report of the results.

(Meszaros, 2007)

All three frameworks have their advantages and disadvantages. **MSTest** is the default unit testing framework in Visual Studio. It is the easiest to use without any extra installations. The bad sides are the slower performance compared to others and in some cases interoperability has been a problem (Dietrich, 2017). MSTest also encourages bad habits and it is slower to understand what the code is doing. The bad habits are related to the use of Setup and Teardown methods. Using these methods is bad practice because it slows down reading of the tests as for each test you first have to navigate to the mentioned methods to see what is being done (Osherove, 2013). By not using Setup and Teardown methods, it is needed to do everything inside one test method. This way you can see everything that the test method does and requires inside the method itself. Using ExpectedException attribute is also bad as it conflicts with the AAA structure by having to create Assert first (Killeen, 2015).

NUnit is ported from JUnit to have a framework supporting C# and it has a long history. It executes tests fast and has some customization features. NUnit creates single object instance per test class. NUnit can be installed as NuGet packages in Visual Studio (Dietrich, 2017).

Finally, **xUnit.net** is the newest and was created by the original creator of NUnit. As it is based on NUnit the two have a lot of similarities. xUnit.net has shorter and simpler naming conventions than NUnit, making the code easier to read and understand. While NUnit has single object instance per test class, xUnit.net has single object instance per test method. This improves test isolation. In xUnit.net the test methods are divided into two groups: Facts and Theories. Facts are test methods that should always be true and Theories are true with the right data. xUnit.net can be installed with NuGet packages in Visual Studio (xUnit.net, 2019).

2.5 Mocking frameworks

Mocking frameworks can be used as a tool to fake data. Commonly mocking frameworks can be used to mock the return data of a function or to give a parameter for a function. Mocking is needed if the data needs to be an object or the data is fetched for example from a database (Osherove, 2013).

There are two main kinds of mocking frameworks: **constrained** and **unconstrained** (Osherove, 2013). **Constrained** framework cannot fake data in certain cases. For example, a constrained framework cannot be used to fake the data from a static method. Examples of constrained frameworks for .NET are RhinoMocks, Moq, NMock, EasyMock, NSubstitute and FakeItEasy.

Unconstrained framework can fake even static methods which makes them very good for legacy systems. Examples for .NET are TypeMock Isolator, JustMock and MS Fakes. From these the only free to use framework is MS Fakes if you have Visual Studio Enterprise version.

2.6 Code coverage

Code coverage tool is useful to see how much of the code is covered with tests. It shows the coverage for the whole system and separately for every project and its classes and functions. The reason why it is a good idea to know the code coverage is to know if a function is fully tested. A function may have many “branches” to go through: different if or which-case

statements, which may return different values. All different outcomes should be tested. Live Unit Testing in the Visual Studio shows right away which parts of the CUT have or have not been tested.

2.7 Dependency injection

Dependency injection (DI) was introduced by Martin Fowler and is based on Inversion of Control (Fowler, 2004). DI is a programming technique used to create more maintainable code. In practice it is noticed by the reduced code coupling. By having object coupled to an interface instead to a specific implementation, the reusability of the code is higher as it is easier to make minor changes and risk of breaking the code is lower (Weiskotten, 2006). By following the DI principles, the result will be a more maintainable system and there is possibility to reuse the code more often. Main benefits for using DI are listed in table 2. Seemann tells about the misbeliefs towards DI and how it is not only useful for one certain thing but it is even more useful for general maintainability. For example, one misbelief was that DI is only needed with unit testing. As we can see the many benefits from Table 2 this is not the case at all. From unit testing point of view, DI is needed in order to be able to create stubs for classes (Seemann, 2012).

Table 2. Benefits of DI (Seemann, 2012).

Benefit	Description	When is it valuable?
Late binding	Services can be swapped with other services.	Valuable in standard software, but perhaps less so in enterprise applications where the runtime environment tends to be well-defined
Extensibility	Code can be extended and reused in ways not explicitly planned for.	Always valuable
Parallel development	Code can be developed in parallel.	Valuable in large, complex applications; not so much in small, simple applications
Maintainability	Classes with clearly defined responsibilities are easier to maintain.	Always valuable
TESTABILITY	Classes can be unit tested.	Only valuable if you unit test (which you really, really should)

There are three main types of DI: Constructor Injection, Property Injection and Method Injection (TutorialsTeacher.com, 2019). Basically, there are three different approaches or places of implementing DI which can be used depending on what needs to be tested.

2.8 Challenges of unit testing

The biggest challenge in this unit testing project is to apply unit tests to the legacy applications. This means a large part of the code will need to be refactored to make unit testing possible. For example, some parts of the system depend on calculations made together with database connections. These calculations are done by retrieving certain values from database.

2.8.1 Where to start

When applying unit tests to a legacy application, the first question is where to start adding the tests. Usually, the development team knows the most problematic places and components can be decided by the team. If you have no idea where to start, Osherove suggests the best approach is to make a priority list of components for which the tests would make the most sense. The factors that affect the priority are logical complexity, dependency level and priority in a project. Each of the three sections would be rated according to the chosen components. Based on the points some estimation can be made: How much work would it take? How much value would it bring? There are two ways to decide what to start with: ones easier to test or ones harder to test. Both of the cases are important to be tested. For legacy

code it is recommended to use unconstrained mocking framework so some components can be tested without having to be refactored. Example of these components are static methods and properties (Osherove, 2013).

2.8.2 Dealing with legacy code

Understanding legacy code can be difficult and time consuming. Before changing the existing code, it is necessary to understand how the code works so nothing will break. The need for large refactoring usually emerges when:

- Small refactorings are put off for too long.
- The architecture is poorly implemented.
- Bigger changes to existing features are needed or some new features will have an effect to existing ones (Roock & Lippert, 2006).

Feathers introduces the legacy code dilemma:

"When we change code, we should have tests in place. To put tests in place, we often have to change code." (Feathers, 2004).

The problem with changing code without having tests in place is that bugs might be created. They are not so easy to notice or even the functionality might change without noticing.

Feathers also introduces an algorithm how to approach the problem and start changing the legacy code. The idea is to first check the places in the code that the changes will effect. Then find out where the unit tests should be written. Next, get rid of dependencies, e.g. from database or the function might have side effects. After removing the dependencies, it is time to write the tests. Final step is to do the changes and refactoring (Feathers, 2004). The structure of the algorithm is similar to how TDD works.

Sometimes code refactoring can take a lot of effort, time and in some cases it can be difficult to decide if it even makes sense to do. Before refactoring, the following needs to be considered:

- How much time would it take?
- Is it worth the time?
    a. How important it is to be unit tested?
    b. How often there might be changes made to the function?

i. If the answer is often, then it is important to have unit tests in place.

c. What is the gained value?

Klammer and Kern shared their experiences of applying unit tests to legacy application with two example cases. The first was with not being able to run the chosen unit in isolation which led to big refactoring. In the second example case, the use of static initialization code, static fields and object initializations in constructors caused problems. Here instead of refactoring, mocking (PowerMock) was used. The mocking made it possible to eliminate the need for big refactoring with the CUT as it offered support for dealing with statics. Conclusions from using the mocking were not good as the resulting tests were very unstable. This was the reason to cancel unit testing as it would have required too much effort to continue (Klammer & Kern, 2015).

2.8.3 Motivation

Final challenge is to motivate the developers. The main source of motivation for unit testing for all the developers in this project's company, comes mainly from the production manager and some of the senior developers. As I was also interested in learning how to do unit testing, the project to integrate unit testing into the organization and development finally started. The right kind of motivation to aspire for the developers is important so the quality of the unit tests is good and it is easier for everyone to keep doing the unit tests. Daka and Fraser conducted a survey with 255 software developers from 29 countries about different kinds of practices and experiences with unit testing. In Figure 1 we can see different kinds of motivation and their effects (Daka & Fraser, 2014). The most important one is probably one's own conviction. When motivation comes straight from the developers, it means they really understand its benefits and feel unit testing is important. It is beneficial to understand why unit testing is important and think of it as a tool to help with developing. In the best case, the motivational part of learning to do unit testing is not a challenge at all. Making it a compulsory part of development nullifies the lack of motivation because there is no other choice but to do it.

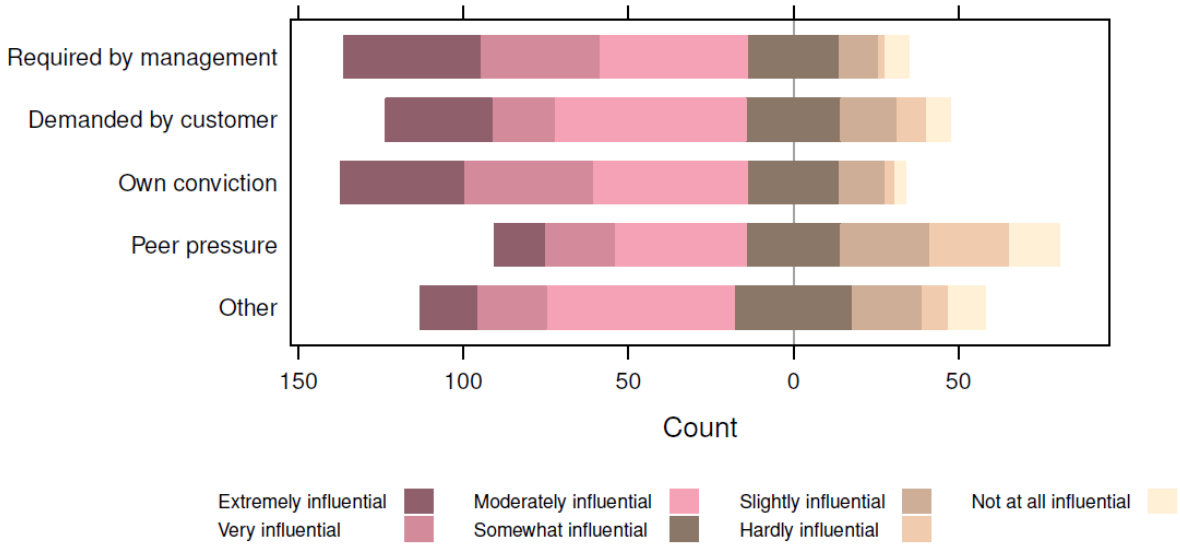

Figure 1. Different sources of motivation for workers (Daka & Fraser, 2014).

3 CASE STUDY

The case study company is a small with less than 20 employees. The company uses agile development as their software development strategy. The agile development methodology is something between Scrum and Kanban. The sprint principle from scrum is still followed. The period for one sprint is two weeks. At the end of the sprint there is a review and retrospective of the sprint and a planning of the next sprint. The planning of the sprint isn't as in usual Scrum. The way it differs is that there are no specific roles in the planning meeting e.g. scrum master. Also, there is no story point voting for tasks (story points give an idea for how much time a certain task might take to be finished). Story point voting can take a lot of time so this way the planning meetings are much shorter. Even though there are two week sprints, it is also possible to take other tasks during the sprint than what was checked in the planning of the next sprint. This means that the process is more flexible than Scrum as in Scrum there shouldn't be changes during the sprint. Being more flexible is better from customer point of view. The main purpose for sprints is to follow how things are going and if there are any improvement ideas for the development process or to the applications.

The company has several products, from which two products bring the main source of income. The bigger from these two is made with ASP.NET Web Forms and the second with ASP.NET MVC. Both products are related electricity markets. Web Forms is made for electricity sales and helps e.g. managing customer contracts, managing the possible risks etc. The MVC application is for electric grid business to manage the pricing of their products and helping with business planning.

As these two products are the core of the company, they are the ones the unit tests will be applied to. This case study will focus on the MVC application. In the future, after the project is over, unit tests will be also applied to the Web Forms application. Visual Studio 2019 Preview was used for general development and for making the examples.

3.1 Problem identification and motivation

This case study includes planning the integration of unit testing to an organization and to legacy ASP.NET applications. The first thing to plan is what needs to be changed in the current development process and after that actual deployment of the plan. Then a plan for integrating the actual unit tests to the legacy applications is introduced: Where to start? What

kind of tests will be used and are needed? Once the plan has been introduced, a report of the practical experiences from creating unit tests will be given.

The **motivation** that drives the research forward are the benefits that unit testing brings. As was already mentioned in chapter 2. the main benefits from unit testing are to have more maintainable system, to notice bugs at earlier stage and to have code that is more testable. In the beginning it will be difficult and many challenges are to be tackled but for sure eventually it will be worth the effort because of the mentioned benefits. Working with code is much easier when it is properly formatted. The reduced amount of unintentional technical debt is also a good motivation for the company to make sure unit testing will be done in the future.

3.2 Objectives for a solution

The main objective is to successfully integrate unit testing as part of the development process. This research should serve as a guideline and as an example of how to properly do unit tests for legacy applications and how to deal with problematic situations. This will be done by code examples and explanations. The goal is not a certain amount of system's code coverage but to provide general instructions how to work with unit tests within ASP.NET application.

The objectives are based on realistic evaluation of how much can be achieved in the given time period. The objectives were defined together with the production manager.

3.3 Design and development

This part explains how to approach integration of unit testing as a part of the organization and planning of applying the tests to the legacy ASP.NET application.

3.3.1 Plan for integrating unit testing to the organization

The aim is not to follow TDD, but to create unit tests only after new code has been made. However, there are still quite many changes that need to be done throughout the whole development process. The software development process consists of several phases. The possible changes to the phases are discussed further below. The phases are: Requirements, Design, Implementation, Testing and Code review.

3.3.1.1 Requirements

Requirements phase will not change from the current activities. The tasks are given specifications and then another person will evaluate if it is sufficient. A requirement applied

to the overall developing process will make unit tests compulsory for all new features. This will only take place once the project is finished and development team has enough experience with unit tests.

3.3.1.2 Design

There are three ways to take unit testing into consideration in design phase:

1. Designer will instruct what kind of tests are needed and should be done for the new feature.
2. Developer will create the most basic tests and designer will note if some special tests should be done.
3. No changes to design phase. Developers will create all the unit tests in the implementation phase.

Designer is a role in which based on the written specifications on the task, the designer writes the technical details for implementation. **Developer** will then implement the feature based on the specification and the technical design. Usually the instructions are given in a step by step format.

At least in the beginning phase the designer will plan how and what kind of unit tests should be done. After the team has enough experience with unit tests, it may be possible to move (separate) plans for unit tests from the design phase to the implementation phase as developer's responsibility. There are two main cases when making changes to the system: completely new features are made or changes are made to the existing code. Depending on the case, the following should be considered with unit tests:

1. New features.
    a. What unit tests should be done?
    b. Is refactoring needed?
2. Changes to existing code.
    a. Unit test exists.
        i. What changes need to be made to the unit test?
        ii. Should some tests be added?
    b. Unit test does not exist.
        i. Is refactoring needed?

1. Plan for refactoring existing code if it requires major change. If it requires too much time and effort, then tests might not be applied.

ii. What kind of unit tests should be done?

3.3.1.3 Implementation

The developers will start creating unit tests for the new features. The tests will be created only after the new features are finished or after changing the production code. **Live unit testing** setting will be used with Visual Studio so the builds will fail if the unit tests fail. This makes it easier to discover the mistakes faster and create good quality unit tests. Live unit testing also runs the tests after any changes to the tests or to the CUT. In this way it is evident right away if the test or the CUT got broken.

3.3.1.4 Testing

Unit tests have no effect to testing phase. The designated tester in the company does not have a coding background and it would not make sense to start learning programming just to help create unit tests. Already unit tests themselves show if the tests are working or not. If the tests are failing, either the CUT has bugs or the test has been written in a wrong way.

3.3.1.5 Code review

After the new features have passed the tests, **pull requests** are created and the code is reviewed. This is how the process works at the moment. In addition to the current one, the created unit tests have to be included in the pull requests. If not, the pull requests will be declined. Quality, correctness and types of unit test are reviewed. If all, new feature and unit tests are fine, pull request can be accepted.

3.3.2 Plan for applying unit tests to legacy ASP.NET applications

As the applications are big, it does not necessarily make sense to create unit test for every single function or method in the code. Instead, the goal is to apply the unit tests to logically complex and calculation parts of the applications. Logically complex parts are basically the parts where there are loops and conditions. In chapter 2.8.1 it was explained where to start adding first unit tests to the legacy code. Parts of the code with most logic and dependencies can be divided into easier and more difficult. It was decided to start with the easier parts to get familiar with unit tests faster. As unit testing is new and should be learned, which can take time, it is more motivating to start writing working tests without first needing to refactor.

Once the testing framework starts to feel familiar, the next phase is learn refactoring. Same as before, it is suggested to start with a simple case and move on to bigger and more complicated functions at a later stage.

In the beginning, a work group of three people was assembled to plan which application to start with, in what places should first unit tests be done, which parts should be refactored next and lastly, apply unit tests to them. It was decided to start with the MVC -application as it is easier to approach. The application is smaller (although still very big) and seems like it would not need as much refactoring as the Web Forms -application. Architecturally, the MVC application already is in a more testable form. On the other hand, both applications have similar problems such as long functions and database dependencies.

3.3.2.1 Chosen unit test framework

xUnit.net and MSTest were chosen for practical comparison. NUnit was left out because xUnit.net is its newer version. A few simple identical tests with xUnit.net and MSTest framework can be seen in the examples below. There is not too much difference between the two. After running the example codes, and based on the information presented in chapter 2.5, xUnit.net seemes to be the better option. xUnit.net executed the tests a bit faster than MSTest. Based on the examples below, xUnit.net took about 360 ms and MSTest around 460 ms. It seem the difference is small but when there are more and more unit tests,the exection time will slowly pile up. Another reason for choosing xUnit.net is simple/shorter naming. (with below example MSTest's AreEqual versus xUnit.net's Equal). xUnit.net gives results in a more readable format. For example, when using parameterized tests with inserting multiple cases to one test, xUnit.net clearly shows which cases failed and which ones passed. It also separately gives information about the failed test cases. MSTest only shows the test failed and separately shows the information for failed test cases.

MSTest example

```
[TestClass]
public class Class1
{
	[DataTestMethod]
	[DataRow(1, 2, 3)]
	[DataRow(1, 1, 1)]
	[DataRow(2.1, 2.2, 2.3)]
	public void CalculateSomeValue_ThreeValues_CalculatedValue(decimal a, decimal
	b, decimal c)
	{
		var result = CalculateSomeValue(a, b, c);
		decimal x = 2.5M;
		Assert.AreEqual(x, result);
	}

}
```

xUnit.net example

```
public class Class1

{

	[Theory]
	[InlineData(1, 2, 3)]
	[InlineData(1, 1, 1)]
	[InlineData(2.1, 2.2, 2.3)]
	public    void    CalculateSomeValue_ThreeValues_CalculatedValue(decimal    a,
	decimal b, decimal c)
	{
		var result = CalculateSomeValue(a, b, c);
		decimal x = 2.5M;
		Assert.Equal(x, result);
	}

}
```

3.3.2.2 Setting up the unit test framework

xUnit.net is simple to get to work with Visual Studio. All that is needed is to install three NuGet packages to the project:

1. xunit
2. xunit.runner.console
3. xunit.runner.visualstudio

The NuGet package xunit is a meta-package and its purpose is to bring the core functionality of the unit testing framework (xUnit.net, 2019). xunit.runner.console package is a solution-level package and it enables running the tests (xUnit.net, 2019). Finally xunit.runner.visualstudio package enables running the tests with Visual Studio and the use

of test explorer (xUnit.net, 2019). With the test explorer it is possible to see all the implemented tests and the state of the tests (failed or success).

After the installation, needed references have to be added to the class libraries and then it is possible to create the unit tests for the selected code.

3.4 Demonstration

This part will demonstrate the findings from the actual implementation of unit tests to the legacy ASP.NET. This includes the lessons learned from testing, encountered problems and their solutions.

3.4.1 The implemented tests

First tests were written to a class where refactoring was not needed. The class contains several short methods related to using system's DateTime class. The most commonly used Assert was "Equal". Others are "True", "False", "Null", "NotNull" and one "Throw". "Throw" was used to confirm the system would throw an error if the user tried giving higher value for start date than for end date. For example, value of start date: 1.6.2019 and value of end date: 1.1.2019.

Both Theories and Facts were used. Theory is ideal if you need to repeat the same tests many times with different values. As was mentioned in chapter 2.5, Theory is true with the right data. On the other hand, Fact should always be true. Fact is better to be used if the inserted data in the Act phase is not so relevant and it would be enough to repeat the test just once. Example of implemented Theory can be seen below.

Implemented Theory

```
[Theory]
[InlineData(2019, 1, 1, 31, 744)]
[InlineData(2019, 2, 1, 28, 672)]
[InlineData(2019, 3, 1, 31, 744)]
[InlineData(2019, 4, 1, 30, 720)]
public    void    GetHoursInTimeZone_CalendarSpanWithNullTimeZone_MonthHoursInUTC(int
year, int month, int bDay, int eDay, int expectedhours)
{
       DateTime begin = new DateTime(year, month, bDay);
       DateTime end = new DateTime(year, month, eDay);
       DateTimeHelpers.CalendarSpan  cs  =  new  DateTimeHelpers.CalendarSpan(begin,
end);
       var result = cs.GetHoursInTimeZone(null);
       Assert.Equal(expectedhours, result);
}
```

CalendarSpan contains begin and end DateTime. `GetHoursInTimeZone` method returns the amount of hours between the DateTimes in the chosen time zone (which is null in this case). In the Theory example five parameters are used for the method with InlineData. From the parameters the final one is the expected result (amount of hours) and the others are used input values to create different DateTimes. In Figure 2 we can see all the test cases passed.

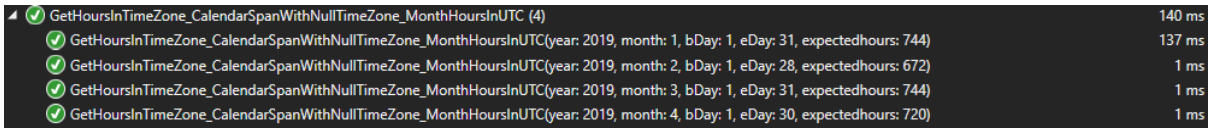

Figure 2. Passing unit test in Visual Studio.

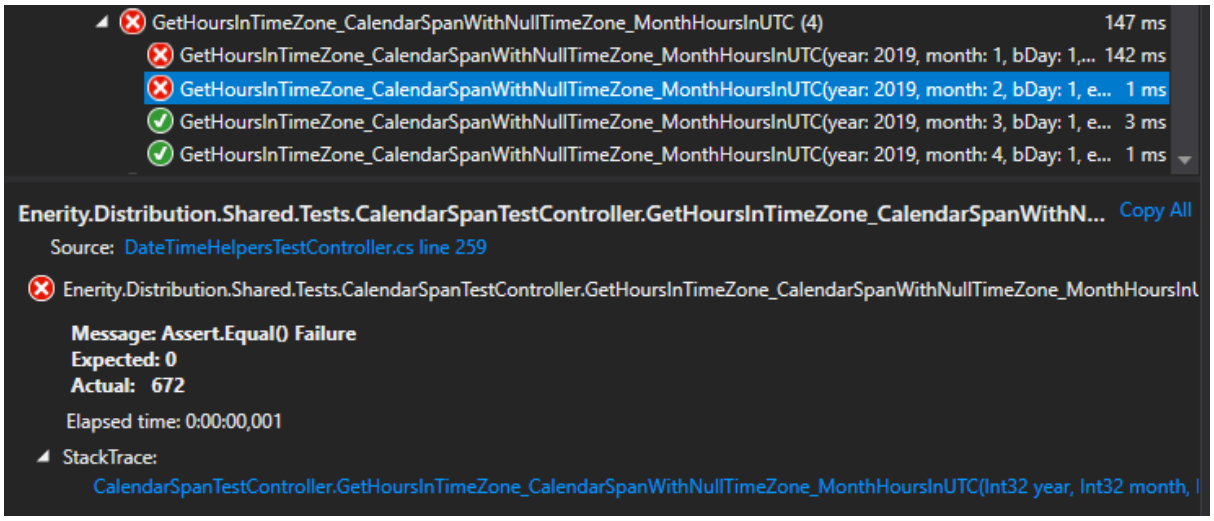

Figure 3. Partially passing Theory with information about failed case.

Figure 3 represent an example of Theory in which some tests pass and other tests fail. The test Theory is exactly the same as used in Figure 2 with four InlineData inserts. The difference with the original Theory example are intentionally changed expected result parameters to be wrong for the first two InlineData (the inspected case has expected value of 0 hours as shown in Figure 3). The point is to demonstrate how failed test information is displayed. If this was a real case, it would mean the test was written correctly but the expected data is wrong. It is important to be sure the expected data and the written test are right. If we assume the test is written right and the expected data is correct, then the next

step would be to check what is wrong with the tested code. Changing working code would be a waste of time so it is better to first double check the tests.

Next example is a case of implemented Fact. We are testing Intersect –method which checks if two Calendarspans overlap. If they do, Intersect –method returns the overlapping time period.

Implemented Fact

```
[Fact]
public void Intersect_FullyOverlappingCalendarSpan_ReturnsContainedCalendarSpan()
{
        DateTime firstSpanBegin = new DateTime(2019, 1, 1);
        DateTime firstSpanEnd = new DateTime(2019, 12, 1);
        DateTime secondSpanBegin = new DateTime(2019, 4, 1);
        DateTime secondSpanEnd = new DateTime(2019, 7, 1);

        DateTimeHelpers.CalendarSpan cs1 = new
        DateTimeHelpers.CalendarSpan(firstSpanBegin, firstSpanEnd);

        DateTimeHelpers.CalendarSpan cs2 = new
        DateTimeHelpers.CalendarSpan(secondSpanBegin, secondSpanEnd);

        var result = cs1.Intersect(cs2);
        Assert.Equal(new DateTimeHelpers.CalendarSpan(secondSpanBegin,
        secondSpanEnd), result);
}
```

3.4.2 Lessons from writing the tests

It is important to have names which are easy to understand. Based on the name, it should be understandable right away what the test is about without having to read the code. Sometimes it might be difficult to figure out a proper name for the test. When in doubt about the naming, it is best to write the test, check what values are used and what is the result.

It helps to organize the tests in groups, e.g. group together tests for the same method (tests which are made one after another). In this case there are several class libraries. One test class library corresponds to program's project (class library). The name of the test project was the same as the tested project with added word "test" to the end. If the name of the project to be tested was "ApplicationName.Logic", the name for the test project would be "ApplicationName.Logic.Tests". Each test project would then have separate test controllers per class file to be tested. This way it is easier and clearer to maintain the tests.

One case caused confusion where there was a class inside of a class. Both classes contained a method which was named exactly the same and even did similar things. The solution how

to differentiate the tests between those two classes was to create the tests inside the same test controller with their own test classes. The other option would have been to separate the tests in different files for each class. It felt easier to choose the first option.

All in all, writing the tests is not the hard part, the hard part is refactoring. If entire code was already in a testable format, it would be quite fast to implement tests.

3.4.3 Challenges with implementing unit tests

The code has to be created in a way that it can be tested. Quite often the code that needed to be tested could not be unit tested or it was challenging to do that without refactoring the code. First major problem is the functions were database dependent and second, the functions were long and did many different things. On top of that, long functions are not only hard to unit test but also difficult to read and understand.

3.4.3.1 Solutions to handling long functions

First the function had to be studied thoroughly. It is necessary to understand what the function really does, what does it affect and how is it dependent to other functions. If the function is refactored without understanding it, the function might end up doing some things differently, change the end result and possibly create other faults.

How to start breaking down a long function? Firstly, clearly group similar things to the same place. When organizing the code, those parts should be commented. Once this are done, check if the function still works as it should. Next, break the big function into smaller ones. Ideal situation would be to create pure functions. Pure function is a function that always gives the same output with the same inputs. Pure function never has side-effects and does not rely on side effects like changing or using global variables (Haverbeke, 2018). Pure function would be the easiest to unit test as it will always work in the same way and has no side effects to worry about. Easiest way to refactor parts of code is to use the **Visual Studio's build in refactoring tool**. Not only it makes it faster, but it also does it in a safe way without breaking anything.

3.4.3.2 Solutions to dependencies

Mocking frameworks can be used to mock the needed data. The point is to fake the data that would usually come from a database. It is not the goal to test the database connection as that falls under integration testing and not unit testing.

3.4.3.2.1 Static modifiers

Static modifiers are troublesome for unit testing. The problem is that for example static methods cannot be instantiated. Which means that static method's return data cannot be mocked (at least not with constrained framework). However, static method can be unit tested by calling it and writing an assert for it. But what to do, if the return data needs to be mocked? If constrained mocking framework is used, refactoring is needed. Easier option would be to just use an unconstrained mocking framework. However, these frameworks usually come with a cost (Osherove, 2013). As an exception, MS Fakes is free to use as a part of the Visual Studio Enterprise and this is why MS Fakes was chosen for mocking (Microsoft, 2016). MS Fakes can be added to the project by right clicking on the chosen reference and then click "Add Fakes Assembly" as shown in Figure 4 (some parts of the Figure are censored to keep the company and its products anonymous). As seen in the Figure below, adding Fakes Assembly also creates the corresponding .dll files in the automatically created Fakes folder. MockingTest-class file has a temporary shim example which will be demonstrated at the end of this chapter.

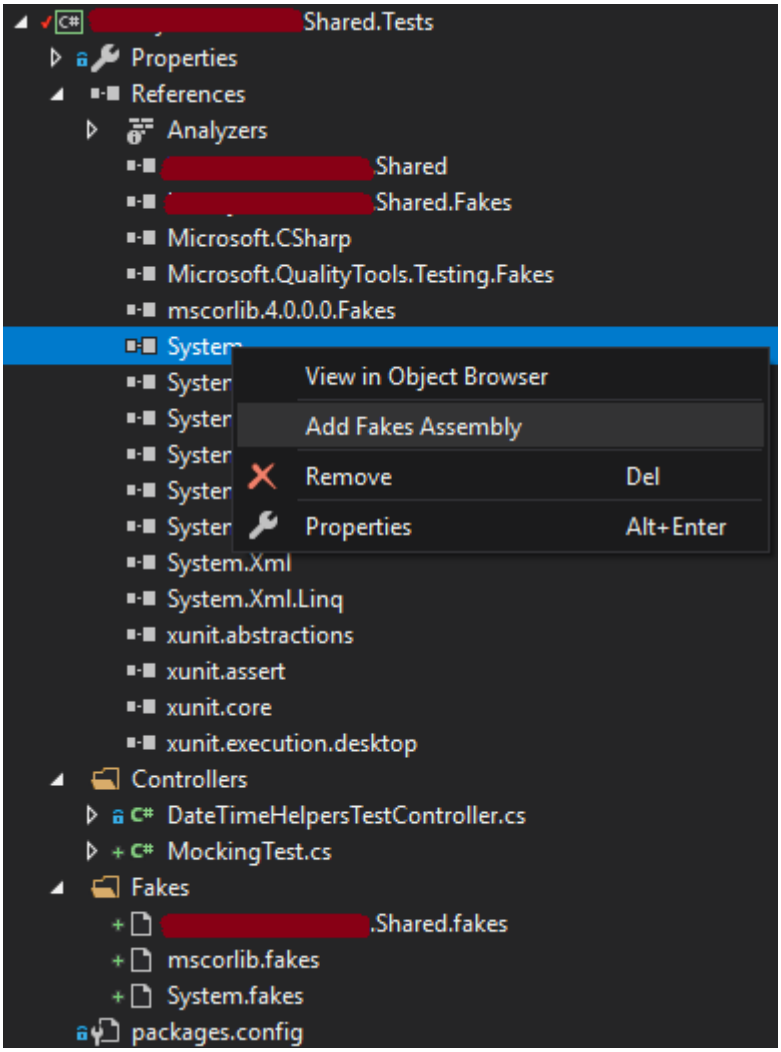

Figure 4. Adding MS Fakes Assembly to a reference in Visual Studio.

MS Fakes has stubs and shims. Stubs can be used for substituting classes depended on interfaces (Microsoft, 2016). Shims can be used for replacing calls to assemblies that cannot be modified like DateTime component (Microsoft, 2016). Shims are also handy for mocking static methods or properties. Table 3 shows what can be and what cannot be handled with stubs and shims (Microsoft, 2013). Stubs are always recommended to be used first as they execute faster and encourage creating more testable code. Shims are useful for parts of legacy code that might be difficult to refactor.

Table 3. Use cases for Stubs and Shims (Microsoft, 2013).

Objective | Consideration	Stub	Shim
Looking for maximum **performance**?	✔	✔ (slower)
Abstract and Virtual methods	✔	
Interfaces	✔	
Internal types	✔	✔
Static methods		✔
Sealed types		✔
Private methods		✔

The example below shows how to shim a property with MS Fakes. *DateTimeHelpers*-class has a property *ThisMonthStart*, that gets the current year, month and the first day of the month. If the current date would be 20.5.2019, the return date would be 1.5.2019. Fakes autogenerates the names and the shim code (compare the naming of class and property in the example: *ShimDateTimeHelpers* <-> *DateTimeHelpers* and *ThisMonthStart* <-> *ThisMonthStartGet*). With the help of the shim we can set the return value to be anything as it is done in the example. This example is simple to show the syntax for shims which only verifies the shim itself works correctly.

MS fakes shim example.

```
using (ShimsContext.Create())
{
	Shared.Fakes.ShimDateTimeHelpers.ThisMonthStartGet = () => { return new
	DateTime(2015, 1, 1); };

	var thisMonth = DateTimeHelpers.ThisMonthStart;

	var result = new DateTime(2015, 1, 1);
	Assert.Equal(result, thisMonth);
}
```

CUT used for shim example

```
public static DateTime ThisMonthStart
{
	get
	{
		return new DateTime(DateTime.Today.Year, DateTime.Today.Month, 1);
	}
}
```

3.4.3.2.2 Testing function that calls other functions

In the best case when a function calls other functions, it is possible to test the return value of the function under test without any problems. This is how private methods can be validated as well. However, if the "child" functions are using external interfaces e.g. database connections, mocking tools can be used to fake the return data. This can be done with stubs in MS Fakes. Problem with creating stubs, is that the CUT has to be built in a way the class or classes are only dependent on the interfaces, not from other classes directly (Microsoft, 2016). In other words, **stubs cannot be made directly for class objects**. Instead of class objects, stubs can be made for interfaces and this allows us to do dependency injections (Microsoft, 2016). If interfaces are not being used already, this once again leads to refactoring.

In Figure 5 we can see a high-level example of how stubs work. This example is taken from Microsoft's documents. When a component (StockAnalyzer) depends on another component (RealStockFeed), it is possible to create a stub (StubStockFeed) for the component (RealStockFeed) that is being used by the main component. Instead of using the original component, the created stub component will be used instead by the CUT and the stub can be given any values for the test.

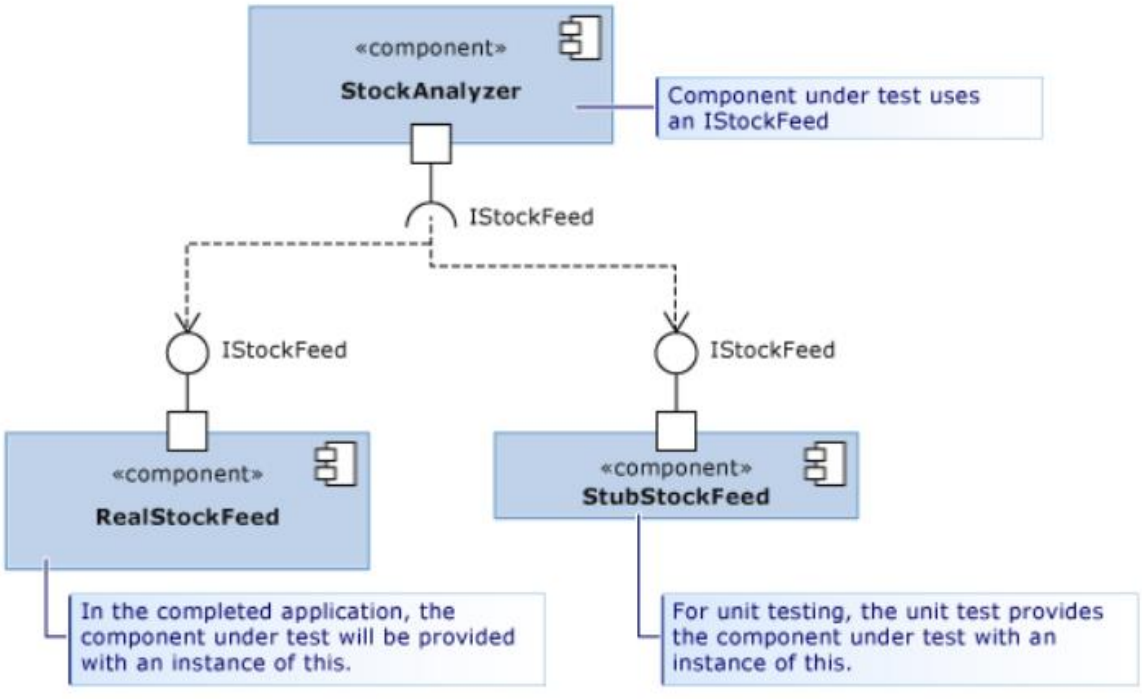

Figure 5. Example case for how stub works (Microsoft, 2016).

Figure 5 demonstrated how stubs work in general and what's the point of them. Below is a separately made up example of how to use Stub in practice. First there is CUT with DI implementation and after that is the unit test made with Stub.

```
public interface IInfo
{
	AnimalInfo GetInfo();
}
public interface IAnimal
{
	string Dog(int age);
}

public class AnimalInfo
{
	public AnimalInfo()
	{
		name = "Woofer";
		weight = 1;
	}
	public AnimalInfo(string n, int w)
	{
		name = n;
		weight = w;
	}
	public string name;
	public int weight;
}


public class InfoClass : IInfo
{
	public AnimalInfo GetInfo()
	{
		AnimalInfo ai = new AnimalInfo();
		return ai;
	}
}
public class Animal : IAnimal
{
	private readonly IInfo _info;
	public Animal (IInfo info)
	{
		_info = info;
	}
	public string Dog(int age)
	{
		var i = _info.GetInfo();
		return "Name: " + i.name + ", Weight: " + i.weight + ", Age: " + age;
	}
}
```

In the example CUT the idea is that the method Dog method returns a string which contains data gathered from GetInfo method from IInfo-interface. In real life case GetInfo could be

retrieving data from a database. Because of this stub should be used to make the test execute much faster and this way we are not doing integration testing.

Below is the implemented unit test for Animal-class's Dog -method. In the test, first a stub is created for IInfo -interface. The return data of GetInfo -method is altered in "`new AnimalInfo("Doge", 5)`" –section and works as injected data. This way the return data now matches the expected string content.

```
[Fact]
public void Dog_StubAnimalInfoAndGiveDogZero_ReturnString()
{
	var infoStub = new Fakes.StubIInfo
	{
		GetInfo = () => new AnimalInfo("Doge", 5)
	};

	Animal a = new Animal(infoStub);
	var result = a.Dog(0);
	var expected = "Name: Doge, Weight: 5, Age: 0";
	Assert.Equal(expected, result);
}
```

While creating the example, one problem occurred. The stub would not generate the stub code properly for the interface if an interface was declared inside a class. This situation was found in the production code which otherwise works fine but it could not be tested like this. Solution was to simply move it outside of the class. Below there is a simple example of before and after CUTs.

CUT before

```
public class Class1
{
	public interface IInterface
	{
		string SomeString();
	}
	public class SomeClass : IInterface
	{
		public string SomeString()
		{
			return "text";
		}
	}
}
```

CUT after

```
public interface IInterface
{
	string SomeString();
}
public class Class1
{
	public class SomeClass : IInterface
	{
		public string SomeString()
		{
			return "text";
		}
	}
}
```

3.4.3.3 Testing methods that take class objects

To test a method with class object parameters, an object instance of the required class needs to be created and given to the method in the method call (Act phase). From testing point of view this can however make the tests very long if the class itself is big with lots of variables.

3.4.4 Refactoring CUT later on

When maintaining the system, also unit tests will need to be maintained. If the tests use stubs or shims to mock some values and CUT will be refactored in some way, most likely test will also need to be changed. Stubs and shims auto generate some code. Names of stubs and shims come from the combination of CUT's name and the parameters it takes. In addition to the name changes, it effects the updating of new parameters or return values that are used or tested.

4 DISCUSSION

When unit testing is mentioned, it is common to find it together with TDD. The relationship of unit testing and TDD seemed to be a bit misleading by reading different literature sources. Unit testing can be done without practicing TDD. Another common view is that TDD is a successful and useful way of developing. However, in this thesis it was ruled out as it seemed to be an even bigger project and the goal of the thesis was to focus more on unit testing.

4.1 Project evaluation

Changing the development process was fairly easy to figure out and was the easy part of this project. The difficulties occurred when adding the unit tests to the legacy application. More problems arose when the unit tests could not be applied straight away to the legacy code. Adding tests to simple parts of the system was easy because in those cases there was no need for refactoring and tests could be written without any issues. But when a so called "monster" –method/function needed to be tested, it always caused problems. The usual problems of the "monster" method/function was its length, it did a lot of different things, included database dependencies and called to many other functions. Using the Visual Studio's build in refactoring tool helped a lot as it was fast to create smaller functions in safe way.

With the mocking frameworks there was a setback caused by the lack of research about different options before choosing the mocking framework. At first, RhinoMocks was chosen but it turned out that it cannot mock static methods which would lead to extra work (more refactoring). Only at a later stage mocking frameworks that can do this were discovered (unconstrained mocking frameworks, discussed in chapter 2.5). The problem is not all of them are free. If free framework is the only option, then a sole free alternative seems to be MS Fakes which is part of Visual Studio Enterprise.

All in all, I learned a lot about unit testing while the research phase and the case study. I believe unit testing is very useful as it can save time and money once it's properly integrated to the organization and its applications. However, it seems it still is not widely used in companies. My opinion is it could and should be way more widespread.

At this point it is hard to judge if unit testing is worth the effort for legacy applications. In this case more time is needed to see how the coding practices are received, if good practices will be kept and unit testing will continue. For sure the results and benefits will not show

fast. Unit testing is a long term investment which will eventually payback in the future. Ideally, there will be less bugs found at later stages and the entire system will be more maintainable.

4.2 Research limitations

This research only focuses on ASP.NET applications and the unit testing frameworks specifically for them. Specific examples might not be applicable to other programming languages, but they do present a general view and idea for how to do unit tests. The information and findings from this research can be utilized with other technologies such as Java, Python etc. TDD was ruled out from the theory part and therefore from the case study. TDD has a big relation to unit testing. It would have been interesting to see what issues it would cause, and what would be the difference between invested time, learning curve and finishing tasks.

4.3 Comparison to existing research

There was not much research done in either one of the research topics *1. Implementing unit testing to an organization* or *2. Actual implementation case study of adding unit tests to legacy application*. Because of the lack of practical implementation research, a lot of the practical coding support and knowledge came from blogs and posts of different professional programmers with knowledge and experience in unit testing.

In chapter 2.8.2 research from Klammer and Kern was explained and presented their problems with applying unit tests into legacy system. The conclusions of their paper states that it is not worth the time and effort to apply unit tests to legacy code. Instead, they will have to compensate the lack of tests with manual testing (Klammer & Kern, 2015).

In this research not much of actual refactoring was yet done. However, it does seem the refactoring might be challenging and could take quite much time as the system was not done from the start with DI in mind.

All in all, there is a fair amount of books and articles about theory of unit testing with some example scenarios (Osherove, 2013), (Daka & Fraser, 2014), (Bowes, et al., 2017), (Klammer & Kern, 2015), (Sommerville, 2011) and the same goes for refactoring legacy systems (Feathers, 2004), (Roock & Lippert, 2006). However, there could be more

information and studies about the practical real-life situations and problems that may emerge.

4.4 Plan for the future with unit testing in the company

It is important to write a testable and maintainable code. It would be ideal to write short functions which only do one operation so it is easy to read and unit test. There are no real rules as to how long functions should be. Therefore, in the past it often occurred to write long functions because it was faster and easier to implement everything at once in the same function. After the research and practical experience from the case study, I can set as a rule of thumb what is considered as a long function. If the whole function cannot be seen in whole on the computer screen, it is too long. Of course this varies because of different settings and screen sizes but I can estimate it accounts around 60 lines of code. Another good rule to keep in mind is the function should only do one thing. This might be hard to follow but in an ideal function, that is how it would be.

Several research papers are mentioning TDD and how good it is. It could be a good idea to try out TDD after unit testing and code refactoring skills are more advanced and people get comfortable with them.

4.5 Future research ideas about unit testing

More concrete and practical research could be done about the effects and benefits of unit tests. This could be the amount of bugs found in testing phase and production before and after having unit tests. Another research subject could be the time effort needed to learn how to do unit tests, write production code that supports unit testing and how much unit tests effects finishing tasks. It would be interesting to compare differences between unit tests without TDD and with TDD. One question that cannot be answered in this research and could be a topic of its own, is to figure out if it really is worth applying unit tests to legacy applications.

5 CONCLUSIONS

This thesis explained what unit testing is, its benefits and how to make them. The case study went through integrating unit testing as a part of the development process in the organization and how to apply unit tests to existing legacy ASP.NET system. In the case study it was noticed there are several issues with applying unit tests to existing legacy system. The main problems to be solved were function dependencies like database connections, "static"-modifiers (e.g. static methods), long functions/methods and complicated parameter types or return types (e.g. class object instances). Solutions for the listed problems are code refactoring and using mocking framework to test isolated functions. Based on the conducted research there are several points supporting the usefulness of unit testing. However, after the conducted research and case study it is still hard to say if applying unit tests to legacy applications is worth the time and effort. If it is a greenfield project, then unit testing is a great tool and benefit for the system. The system should be designed and implemented properly from the beginning to support unit tests.

Main thing to consider if someone should decide for unit testing, is testability, quality and productivity. Some might prefer to focus on one side more than the others. I would guess that for most agile companies the productivity would be one of the higher priorities. The essence of agile development is to react fast to customer needs and deliver fast. In this sense, the time (and therefore money) investment required for unit testing, can seem intimidating and might be the main reason why some would not want to decide for it. Of course, there could be a compromise such as partly adapting unit testing to the organization. Unit testing could be applied for example only on some smaller, simpler and new features of the system. Still, these parts should be the ones that would have enough significance to be unit tested. Unit tests could especially be put in place if the new functionality can be made so it does not dependent on the legacy code and could be created independently. Even if unit tests were only partly utilized in some parts of the system, it is still better than no tests at all and testing everything manually. Manual testing is time consuming and some things can be harder to notice at the first glance.

REFERENCES

Aniche, M. F., Oliva, G. A. & Gerosa, M. A., 2013. *What Do The Asserts in a Unit Test Tell Us About Code Quality? A Study on Open Source and Industrial Projects.* Genova, IEEE, pp. 111-120.

Bowes, D. et al., 2017. *How Good Are My Tests?.* Buenos Aires, IEEE.

Daka, E. & Fraser, G., 2014. *A Survey on Unit Testing Practices and Problems.* Naples, IEEE.

Dietrich, E., 2017. *Stackify.* [Online]
Available at: https://stackify.com/unit-test-frameworks-csharp/
[Accessed 22 March 2019].

Dohnert, C., 2018. *DEV.* [Online]
Available at: https://dev.to/donut87/one-assert-per-test---what-is-that-about-4l75
[Accessed 13 March 2019].

Feathers, M. C., 2004. *Working Effectively with Legacy Code.* Saddle River, NJ: Pearson Education.

Fowler, M., 2004. *Inversion of Control Containers and the Dependency Injection pattern.* [Online]
Available at: https://martinfowler.com/articles/injection.html
[Accessed 21 5 2019].

Ganesan, D. et al., 2013. *An analysis of unit tests of a flight software product line.* s.l., ScienceDirect, pp. 2360-2380.

Haverbeke, M., 2018. *Eloquent JavaScript.* 3rd ed. s.l.:s.n.

Holvitie, J., Leppänen, V. & Hyrynsalmi, S., 2014. *Technical Debt and the Effect of Agile Software Development Practices on It – An Industry Practitioner Survey.* Victoria, BC, Canada, IEEE.

Karac, I. & Turhan, B., 2018. What Do We (Really) Know about Test-Driven Development?. *IEEE Software,* 35(4), pp. 81-85.

Killeen, S., 2015. *SeanKilleen.com.* [Online]
Available at: https://seankilleen.com/2015/06/xUnit-vs-MSTest/#
[Accessed 22 March 2019].

Klammer, C. & Kern, A., 2015. *Writing Unit Tests: It's Now or Never!.* Graz, IEEE.

Meszaros, G., 2007. *xUnit Test Patterns: Refactoring Test Code.* 1st ed. Upper Saddle River, NJ: Addison-Wesley.

Microsoft, 2013. *Better Unit Testing with Microsoft Fakes.* 1.2 ed. s.l.:Microsoft Corporation.

Microsoft, 2016. *Isolate code under test with Microsoft Fakes.* [Online]
Available at: https://docs.microsoft.com/en-us/visualstudio/test/isolating-code-under-test-with-microsoft-fakes?view=vs-2019
[Accessed 20 5 2019].

Microsoft, 2016. *Use stubs to isolate parts of your application from each other for unit testing.* [Online]
Available at: https://docs.microsoft.com/en-us/visualstudio/test/using-stubs-to-isolate-parts-of-your-application-from-each-other-for-unit-testing?view=vs-2019
[Accessed 21 5 2019].

Myers, G. J., Sandler, C. & Badgett, T., 2012. *The Art of Software Testing.* New Jersey, John Wiley & Sons.

Osherove, R., 2013. *The Art of Unit Testing.* Second ed. NY: Manning.

Peffers, K., Tuunanen, T., Rothenberger, M. A. & Chatterjee, S., 2007. A Design Science Research Methodology for Information Systems Research. *Journal of Management Information Systems,* 24(3), pp. 45-77.

Reese, J., 2018. *Microsoft.* [Online]
Available at: https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-best-practices
[Accessed 1 March 2019].

Roock, S. & Lippert, M., 2006. *Refactorings in Large Software Projects.* Chichester: John Wiley & Sons.

Seemann, M., 2012. *Dependency Injection in .NET.* 1st ed. New York: Manning Publications Co.

Sommerville, I., 2011. *Software Engineering.* 9th ed. London: Pearson Education.

TutorialsTeacher.com, 2019. *Dependency Injection.* [Online]
Available at: https://www.tutorialsteacher.com/ioc/dependency-injection
[Accessed 22 5 2019].

Weiskotten, J., 2006. Dependency Injection. *Dr. Dobb's Journal*, 31(5), pp. 10-15.

xUnit.net, 2019. *xUnit.net.* [Online]
Available at: https://xunit.github.io/
[Accessed 22 March 2019].