Evaluation of Test Driven Development

by Hans Wasmus

Evaluation of Test Driven Development

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

SOFTWARE ENGINEERING

by

H. Wasmus
born in Dordrecht, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands

EPCOS, Inc.
186 Wood Avenue South
Iselin, New Jersey
USA

Cover picture: East Village, New York City where a part of this thesis has been written

Evaluation of Test Driven Development

Author: Hans Wasmus
Student Id: 1047957

Abstract

This thesis is about a development practice that will help to create clean code that works, called Test Driven Development. Test Driven Development (TDD) is part of the Extreme Programming development methodology and is based on the principle to test a program before you write it. By adding a new test every time a feature has to be developed, the functioning of the rest of the application is guaranteed by the previously written tests. Developing in these small steps will help developers to conquer complex problems and create cleaner, bug-free code. Several claims concerning Test Driven Development are evaluated in this thesis by executing a case study. Focus of this evaluation is on the usability of TDD, satisfaction of the stakeholders and the quality of the product. Finally, this thesis will provide some solutions to problems which are found during the case study.

Thesis Committee:

Chair: Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor: Dr. H.-G. Gross, Faculty EEMCS, TU Delft
Company supervisor: K. Bauman, EPCOS, Inc.
Committee Member: Dr. K. van der Meer, Faculty ISA, TU Delft

Contents

ACKNOWLEDGEMENTS .... **IX**

PREFACE .... **XI**

1. INTRODUCTION **1**

1.1. SOFTWARE DEVELOPMENT 2
1.2. PREDICTIVE APPROACHES 3
1.3. ADAPTIVE APPROACHES: AGILE DRIVEN DEVELOPMENT 5
1.4. PROBLEM STATEMENT 7
1.5. OUTLINE 8

2. EXTREME PROGRAMMING & TEST DRIVEN DEVELOPMENT **9**

2.1. EXTREME PROGRAMMING 9
2.1.1. Values, Principles and Practices 10
2.1.2. Class, Responsibilities, and Collaboration cards 12
2.1.3. Unit Tests 13
2.2. TEST DRIVEN DEVELOPMENT 14
2.3. CLAIMS 17

3. CASE STUDY: PRODUCT DESCRIPTION **20**

3.1. WHY A CASE STUDY? 20
3.2. EVALUATION ENTITIES 20
3.3. FORECASTING WEB APPLICATION IN .NET 22
3.3.1. Current situation 22
3.3.2. Project vision and scope 23
3.3.3. System Architecture 24
3.3.4. The client 24
3.3.5. Web service and Database 25

4. CASE STUDY: DEVELOPMENT PROCESS **27**

4.1. PROJECT PLANNING 27
4.2. REQUIREMENTS ANALYSIS 28
GLOBAL REQUIREMENT – G1 29
4.3. TEST STRATEGY AND FAULT MODEL 29
4.4. FAULT MODEL 30
4.5. TEST STRATEGY 32
4.5.1. Unit testing 32
4.5.2. Use case testing 33

4.5.3. Security testing ... *35*
4.5.4. Integration testing ... *38*
4.6. IMPLEMENTATION AND BEYOND ... 39

5. EVALUATION ... 41

5.1. USABILITY - GENERAL ... 41
5.2. USABILITY – FULL LIFECYCLE ... 43
5.3. COMPONENTS: DATABASE TESTING ... 47
5.3.1. Possible Improvements/Recommendations ... *48*
5.4. COMPONENTS: WEB SERVICE ... 48
5.4.1. Web service: WSE ... *49*
5.4.2. Web service: AOP ... *50*
5.4.3. Possible Improvements/Recommendations ... *51*
5.5. COMPONENTS: CLIENT ... 51
5.5.1. Client: Interface ... *52*
5.5.2. Client: 3rd party components ... *54*
5.5.3. Possible Improvements/Recommendations ... *54*
5.6. TESTING INTERFACE COMPONENTS WITH ASPECTS ... 55
5.7. EVALUATION OF THE CLAIMS MADE ... 58
5.8. SUMMARY ... 61

6. CONCLUSION ... 63

6.1. FINAL THOUGHTS AND FUTURE WORK ... 64

BIBLIOGRAPHY ... 66

A FORECASTING WEB APPLICATION IN .NET ... 71

Acknowledgements

This thesis is the result of internship committed at EPCOS, Inc. A previous research brought reliability of software under my attention. For this thesis I came in touch with Arie van Deursen, who suggested looking into Extreme Programming and Test Driven Development. I was immediately attracted by the development methodology, because it aims not to create a software product with a perfect design, but a product which is functional for an end user. This, while keeping code readable and the application working.

After various months of searching for a suitable internship in the US, the internship at EPCOS was perfect for me. EPCOS had a project for me and offered freedom to invest time in my Master Thesis. This project would not have been possible without the help of Gerd Gross and Ken Bauman. Ken was my supervisor during my time at EPCOS, Inc. He provided me the guidance and freedom to complete both the business as the academic goal of the project. After being switched several times, Gerd Gross became my supervisor at the TU Delft. His input was invaluable for the completion of this thesis.

I would like to thank my colleagues at EPCOS, Inc. who helped me during my internship: Dave Gubernat, Nelson Nunez, Denise Toryk, James Florance and Achim Buecklers. Special thanks go to Mandy Kratina for investing time to learn about TDD and taking over my project. Thanks to the interns at Siemens and EPCOS, Inc., and all the family and friends who visited me, for making this time so much more than just an internship.

Finally, I would like to place much gratitude to my family, who supported me continuously during my years at the university. Many thanks for your advice, suggestions and opinions. Also many thanks to Chantal for her patience, laughs and for always being there to listen to my stories. Thank you!

HANS WASMUS
Papendrecht, The Netherlands
May 14, 2006

Preface

This thesis is the result of a case study in Test Driven Development done at EPCOS, Inc. in Iselin, New Jersey for the TU Delft, The Netherlands. The case study is the implementation of a webservice and application replacing a Microsoft Access project. The duration of the case study is 10 months. After the case study the project itself will continue to be developed in the future. The purpose of the case study was to create a way to be able to register estimations of future market demand and to anticipate on these estimates.

The thesis comes forth from a study which was the result of a literature research assignment on the reliability of software. The literature research was in strong relation with a project called TRADER, which is a collaboration of industrial and commercial partners. Among the partners in the TRADER project are; the Embedded Systems Institute (ESI), Phillips, Delft University of Technology, Twente University and the University of Leiden. A result from this study was that testing of software applications is one of the influences in the reliability of a software application.

There is a movement in the software development world which uses Extreme Programming (XP) and Test Driven Development (TDD) as part of XP to increase the reliability of software. With the research carried out for this thesis a detailed analysis of the TDD approach is created, it will verify claims made about TDD and review if TDD is a good basis for creating reliable software.

1. Introduction

Since the creation of the first computer program, scientists have been trying to seek better ways to develop computer applications. This science is called *software engineering*. The way a software program is developed is called a *software development methodology*. Initially software development methodologies were based on methodologies used by electrical engineers to build hardware systems [1]. Gradually, major differences between the development processes of the two domains were acknowledged [2] and new methodologies for software development emerged. The road to the perfect methodology however, seems to be a long one.

A research firm, called the "Standish Group", found that companies report, that only less than a third of the software development processes are considered successful [3]. Much effort is done by introducing new technologies and methodologies to increase the success rate of software projects. This thesis evaluates a recently introduced development practice: *Test Driven Development* [4]. A *development practice* is a day to day activity executed during the development of a software project. Test Driven Development is a development practice in which tests are implemented before implementing features of a software project.

Test Driven Development (TDD) has been invented by Kent Beck [4]. TDD is a development practice which is part of a software development methodology called *Extreme Programming* (XP) [5]. TDD is based on the idea to create tests for the program before you develop the program code. This is the opposite of what is usual in current software development methodologies. In Test Driven Development, tests represent specification rather than validation of the functionalities of a software system. In contrast to traditional development approaches, tests are implemented before any code is written instead of after. The reason to implement tests first is to reduce the fear of a programmer, fear that the developed application is not working correctly. When people experience fear they become more tentative, communicate less and are more afraid of getting feedback. The tests in TDD make programmers confident something works "forever", decreasing the fear the programmer experiences [32].

This thesis will evaluate the claims that are made supporting Test Driven Development. A case study has been carried out using TDD and related software development methodologies, such as XP. In this study a big part of developing a software application has been executed to broaden the scope of the evaluation of TDD. This thesis is purely a usability evaluation of TDD; it is not a comparison between TDD and other development practices. Such comparisons between TDD and other practices can be found in [6], [7] and [8].

Before going into the case study, the related topics will be discussed. First software development and various types of software development methodologies will be discussed. Extreme Programming is a methodology based on Agile Driven Development, which is one of the software development methodology types.

1.1. Software Development

Software development is the total set of activities required for building a software system [9]. Software development is carried out in a process: to achieve an outcome, the software process will have to execute one or more activities [10]. To structure the development process, a software development methodology is used. A software development methodology increases the chances of a good quality product by decreasing the overall complexity of the software engineering effort" [11].

A software methodology can help a software engineer in his attempt to create software systems. A software development methodology is defined as: (1) an integrated set of software engineering methods, policies, procedures, rules, standards, techniques, tools, languages, and other methodologies for analyzing, designing, implementing, and testing software; and (2) a set of rules for selecting the correct methodology, process, or tools [12]. A software methodology also gives us helper processes that assist in the overall framework of system designing[1].

With help of a software development methodology software engineers in organizations are able to translate user requirements into a working application, using various development activities[1]. These activities and the time spent on

---

[1] The term *method* is omitted here due to the confusion between the difference between *methods* and *methodologies*. Some authors consider method and methodology as synonyms. The IEEE standard only defines *methodology* as a body of methods, rules, and postulates employed by a discipline [IEEE Standard for Developing Software Life Cycle Processes]. *Methods* are general guidelines that govern the execution of some activity [Ghezzi, C., Jazayeri, M. and Mandrioli, D. (1991). Fundamentals of Software Engineering, Prentice Hall.].

them are defined within the methodology. More general, a methodology is the infrastructure behind all of the development process in an organization [13].

We can consider two types of software development methodologies: predictive approaches with a large basis on design upfront and adaptive approaches, which focus on a quick start on the implementation of the end product. Predictive approaches are commonly known as *Traditional Software Development Methodologies* [62]. In this thesis the term predictive is preferred over the term *traditional*, because traditional implies that the models are from past time. Another reason to use predictive versus adaptive approaches is that it much clearer displays the difference between the two approaches.
One of the main parts of a software development methodology is the choice of the process model. This is the way the activities of a software process are organized in terms of order and time [10]. The outcome of each activity is a product that can be used in a subsequent activity. Introduced in 1970 the waterfall process model is considered the 'classical model' of predictive software development methodologies [10]. Later, several other predictive process models have been proposed. The distinction of process models in adaptive approaches is a little fussier. Adaptive approaches process models have several variations [14]. Some, like "cowboy coding", have no process at all [15].

Properties of predictive approaches and their deficiencies will be discussed first. After that the focus will be on adaptive approaches and the solutions these methods offer to solve the problems of predictive approaches.

1.2. Predictive approaches

Typical approaches in software development are mostly "one shot models". One shot models are based on the principle 'get it right first time' [10]. Process models from predictive approaches are characterized by their structure which is defined in great detail, having a set of steps and rules.

One of the most commonly applied process models is the waterfall model. Introduced as being a bad example of developing software systems [16], the waterfall model still remains a popular model in the software engineering community. The waterfall is a one shot model in its purest form: going from phase to phase the need for going back to a former phase is minimized. After a phase is completed, the process is moving on to a next phase. An advantage of the waterfall approach is that if it works well, the software project can be planned with precision. However, it is in software development's nature for the stakeholders to change their requirements often. When a stakeholder changes a requirement when using the waterfall model, the developer has to return to the first phase of the development process: the requirements analysis. The requirements analysis and the phases thereafter have to be redone using a considerable amount of resources.

The spiral model was introduced as an improvement on the waterfall model [17]. The spiral model consists of similar phases as the waterfall model, but is based on cycles instead of linear phases. Each cycle represents progression in development and connected cycles represent different levels of elaboration.

An extension of the waterfall model, the V-model, has for each of its development phases a verification phase. The process is followed as a V model as a top-down process, which stresses on testing. After the project is implemented verification starts correcting the development phases. Using the V-model the product slowly evolves into a final product, by means of prototyping. The V-model is adopted by major software developing companies, such as ASML [18].

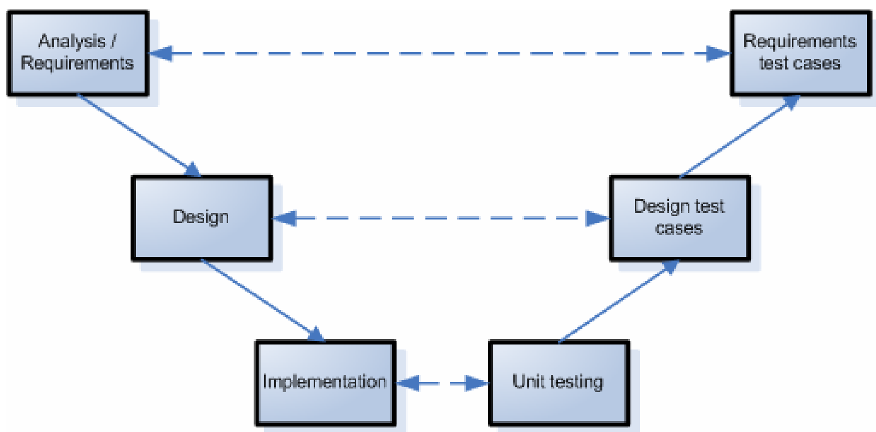

Fig. 1. The V-Model (simplified)

Research shows that the delivery time of software products exceeds the planned time by almost 200% [19]. This is mostly due two changing requirements of the stakeholders [20]. In predictive approaches, changes in requirements are an obstacle in the development process and can slow the process down. The final release is mostly the first software product to contact the end user. Prototypes are used to involve the stakeholders in the software development process. A prototype is an example (which can be a demo or a screenshot) and not a partly finished product. The waterfall model adopts a "build it twice" step embedded in the requirements analysis and design phase [17]. If the user has a very clear mind about what is required, the analysis phase can provide enough information for a successful product. It commonly happens, however, that users can only refine their idea of the functionality of a software product after working with it. This is what adaptive methods try to overcome by incorporating the end user with the production process in an early stage.

1.3. Adaptive approaches: Agile Driven Development

At the other end of the spectrum of development methodologies, we find adaptive approaches. Adaptive methodologies are recently developed and were initially seen as immature approaches. Adaptive approaches now attract a lot of attention in the scientific and business world [21]. They are usually combined under the name *Agile approaches* [22].[2] Agile approaches are born from noticeable drawbacks of predictive approaches:

- inflexibility concerning requirement change requests,
- very formalized process schedule and tasks not adapting to the nature and size of the project [23],
- resulting in long project which can take years to complete a functional prototype [23].

The most important difference between predictive approaches and adaptive (agile) approaches is their focus. Agile approaches place emphasis on people factors, especially on communication between people. Predictive approaches focus on predefining a process in which a software product can be created. This includes communication within the project team and outside the project team [24]. Agile approaches hereby contrast predictive approaches, which focus on processes and tools [25].

Agile model driven development is not a complete methodology in itself, but combines a set of practices which focus on a high customer participation. The main ideas for the Agile model driven development method come from S.W. Ambler [26] who adopted ideas from the Extreme Programming (XP) development methodology and K. Beck, a founder of XP [5] and great supporter of the Test Driven methodology. By observing the software development process, several practices have been combined with high customer participation resulting in Agile development. In 2001, a group of experienced software developers created the “Agile Manifesto” [25], which is a priority list of principles”:

[2] We omit other adaptive approaches such as Ad Hoc coding or Cowboy coding, since they do not have a defined process at all and are therefore not of interest of this thesis

- individuals and interactions over processes and tools,
- working software over comprehensive documentation,
- customer collaboration over contract negotiation, and
- responding to change over following a plan.

An important property of Agile development is the lack of a fixed predefined development process. Instead of a large upfront design process, a high interaction with the customer is preferred. Perhaps this lack of a large upfront process is the reason why Agile development was initially not generally accepted as a mature development methodology. Companies' software development processes are often defined by ISO standards such as the ISO 9000 [53], which require a well defined process structure. It is, however, inadequate incorrect to say Agile development lacks a structure at all. Instead of having a design phase at the start of a project, the design phase is executed in small steps as an ongoing basis during the implementation [15]. The design can be adapted according to change request emerging during the development process and this property of Agile Driven development is considered the key feature in developing successful software products [61][15].

A characteristic of Agile approaches is development in cycles. In most Agile processes the lifecycle of a project is defined as follows:

Cycle 0:
Initial Requirements Modeling
Initial Architecture Modeling

Cycle 1...n
Model storming
(Test driven) implementation

The initial cycle can take up to a few weeks in a large project. The other cycles are done on a daily basis, where the main focus lies on at the implementation and communicating with the stakeholders [27]. Each of the iterations results in a working product which can be shown to the customer. During the execution of the cycles the process can be adapted to customer input.

Predictive and adaptive software development methodologies can be compared with methodologies in other sectors. Predictive approaches are best compared with furnishing houses upfront: all the wishes of the owner are gathered, a design is made and finally the house is furnished. Afterwards the owner has to replace everything, because arrangement is not what it seemed. For example the house should have blue furniture instead of red. Predictive approaches try to make the owner give insight on the end result in the design phase by supplying the owner with images and sketches. This is done to reduce the change of a complete refurnishing. Adaptive approaches have following a different approach: first only a minimum set of desired furniture is bought with the owner. Slowly the home is given a shape by adding new stuff with input of the owner. If something does not fit right, the product is returned and replaced. It could be argued that the products developed with an adaptive approach are never finished. Products are released when they have enough functionality to make it operable, but have to be adapted continuously during their lifetime. Predictive approaches try to minimize the risk of a complete refurnish by showing the owner a part of the real result. For example one room is furnished first. If the owner wants another color, only this room needs to be refurnished.

There are several process models[3] available such as Adaptive software development, Crystal development, Scrum and most prominent: Extreme Programming [28]. TDD is one of the practices of Extreme Programming; therefore, both topics will be covered in this thesis. To evaluate the usability of TDD, I have performed a case study according to the Extreme Programming process focusing on TDD. The case study consists of the development of an application containing a wide range of new technologies such as web services, aspect oriented programming and specific capabilities of the .NET framework. These technologies are used to provide a good basis to evaluate TDD. Since there is an emphasis on social factors in XP the non-technical details of the development process will also be investigated. This concerns the communication with the stakeholders, introduction of new members in the development process and the satisfaction of the developers. It is argued in Agile processes that feedback and communications ultimately lead to a better product.

1.4. Problem statement

This thesis will evaluate the following claims concerning Test Driven Development (TDD) and Extreme Programming (XP):

1. TDD handles changes in requirements better than predictive approaches,

[3] Kent Beck refers to Extreme Programming as a development style [Extreme Programming Explained]. Extreme Programming clearly describes a model for a process to develop software.

2. Developers will be able to look further with TDD, think of better solutions while programming,

3. The XP/TDD process is more enjoyable for the stakeholders,

4. Except for delivering a better suited product, the quality of the code will also improve with TDD,

5. It is possible to apply TDD in a project anywhere requirements are implemented. With minor adaptations TDD will be suitable for testing databases, interfaces and other technologies.

The evaluation of the claims is done by executing a case study using TDD. The case study is used to gather information about developing software with TDD. The claims made are evaluated according to the results of this investigation. Finally, problems with TDD are analyzed and possible solutions are proposed.

1.5. Outline

This thesis is divided into five main parts. In the next chapter, several topics which are connected with the study are described. The chapter is finalized with a list of claims which are to be verified in this study. Chapter 3 and 4 describe the case study which is used to verify the claims made and to investigate the possibilities and limitations of the Test Driven Development methodology. The claims made are evaluated in Chapter 5. This chapter will elaborate on the findings from the case study and will describe in which situations TDD is useful. The study is finalized with a conclusion and possible future points which can be investigated.

2. Extreme Programming & Test Driven Development

2.1. Extreme Programming

Extreme programming (XP) is a software development approach which led to numerous other software methodologies and practices, such as Test-Driven development and paired programming [5]. It is created based on observations on what made software development faster and what made it slower [5]. Despite the many arguments for and against this kind of methodology, Extreme programming has been embraced by the commercial sector during the last ten years [21].

XP is primary about giving up standards and principles and moving towards a more adaptive way of working. These standards are more a protecting shield for the liability of the developers than that they do any good to productivity [5]. XP focuses not merely on technique, but also emphasizes on human relations and their requests in a software project. More attention towards human relations in a software project leads to *higher customer satisfaction* [60], which in turn generates more business. One of the observations made by the creators of XP is that software requirements are often vague and/or rapidly changing. XP is designed to adapt to changes in user requirements and is a lightweight approach. Being lightweight means that only the minimum set of features is developed to create value for a customer. This creates flexibility to change features in a software project, since there is less to change.

One of the core elements of XP is change and the ability to change things in the architecture of the application. To quote Ron Jeffries:

*"Face it: the requirements will **not** be known at the beginning, and they **will** change along the way. To be truly successful in software development, we must recognize this and ride the wave."*

This is the main difference between predictive software development methodologies and extreme programming. Predictive software development methodologies mostly gather requirements at the beginning of the project. When designing and implementing the system, changes are something what cannot be avoided.

There is no full prescribed activity sequence specified in XP. According to the adaptive approaches, implementation of the projected product starts quickly leading to an incremental delivery of the product [5]. The initial iteration of a project in XP starts with a rough analysis of the requirements, by extensively communicating with the stakeholders, for example end users and the customer. The requirements are listed according to their importance. An overall very general architecture is specified by using the requirements. In this phase it is inevitable that global technologies which are required by the architecture are specified. The choice of which technology to use will generally be done by implementing samples to see the performance and the potential of the techniques.

Because of the adaptive property of XP, the establishment of one legal contract is difficult. In XP, large contracts can be split up in several sub contracts including optional contracts, to be executed when both parties agree. Another way to handle the financial side of software projects is to use a pay-per-use scheme. Equal to the cell phone/message system, the customer only pays when it is actually using the system. Since XP delivers working products in early stages of development, money can be earned earlier too.

When comparing XP with predictive approaches we can see substantial differences:

1. Short iterations (weeks) versus long (months) or no iterations,
2. Design mainly created while implementing versus design upfront,
3. Practices versus process time planning,
4. Focus on people versus based on processes,
5. Test implementation first versus test implementation last,
6. Heavy customer involvement during the whole process versus customer involvement during the initial and final phases.

2.1.1. Values, Principles and Practices

Extreme programming minimizes the time spent on initial design, because it assumes the customer will change the requirements frequently. Instead, XP makes use of development cycles, called iterations. Each iteration delivers working products and last mostly a few months. The purpose of small iterations is receiving feedback, keeping track of the development progress and to catch defects early. XP is built on several values, principles and practices. Practices are tasks a software developer executes to create software, for example writing a test.

A value is something we like or dislike in a particular situation or what we think is important. Values are a high level criteria used to judge things in a particular situation. For example team meetings can be judged positive according to the value communication. Principles connect values with practices. They use values to select practices. The value "communication" can be implemented in several practices, for example: use email, have a conversation or write a book. A principle is used to select the way the communication is done by measuring the effectiveness for a particular situation depending on the context.

The (main) values in Extreme programming are [5]:

1. **Communication.** Models promote communication between your team and your project stakeholders as well as between developers on your team.

2. **Simplicity.** It is important that developers understand that models are critical for simplifying both software and the software process — It is much easier to explore an idea, and improve upon it as your understanding increases, by drawing a diagram or two, instead of writing tens or even hundreds of lines of descriptive text.

3. **Feedback.** By communicating your ideas through diagrams, you quickly gain feedback, enabling you to act on that advice.

4. **Courage.** Courage is important because you need to make important decisions and be able to change direction by either discarding or refactoring your work when some of your decisions prove inadequate.

5. **Respect (added later).** People in a project should care about each other and about the project.

On the other side, there are various practices in XP. A selection of practices which distinguish XP from other approaches are [5]:

1. **Sit together.** People in a project should spend time together working on a project, preferably in an open space.

2. **Whole Team.** A project team should be established having clear boundaries of who is a team member and who is not. A team should include all the skills and perspectives to make the project succeed. Team members should have a sense of identification with the team.

3. **Pair Programming.** Sit with two people behind a computer and switch regularly. Pairing has many advantages such as keeping each other on track and improvements in brainstorming

4. **Test-First Programming (Test Driven Development).** Write a test before coding anything, which is the main topic of this thesis.

5. **Incremental design.** Make small iterations in the design which allow testing and keep the project under control.

Connecting the values to the practices, there are many principles defined within XP. For a full list covering many principles which are being used in XP can be referred to [5]. One of the principles is 'Reflection' which, for example, connects the value feedback and communication to the practices pair programming and incremental design. Teams think about how and why they are working. Another principle which can be identified is 'Baby steps'. With use of very small steps, complex problems can be solved incrementally, reducing the risk of changing faster than people can. The people working in the team can only change slowly; making too big changes can cause resistance and confusion.

Two practices "Class, Responsibilities, and Collaboration cards" and "Unit testing", which are related to the main topic of this thesis will be discussed next. In Section 2.2 we will describe the most important XP practice for this thesis: Test Driven Development.

2.1.2. Class, Responsibilities, and Collaboration cards

The overall design of the system is done with Class, Responsibilities, and Collaboration (CRC) cards. These cards represent objects and can be shifted around to find an optimal architecture of the system. CRC cards are papers of around 15x10cm. They contain the class name, and two lists of responsibilities and collaborators. A CRC card looks as follows:

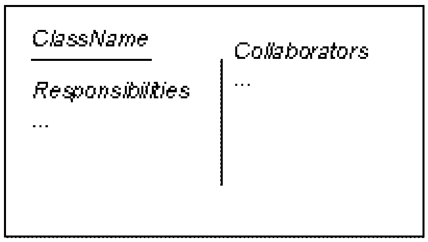

Fig. 2. CRC card [29]

CRC card design is started by making cards for the known entities in a software project, then slowly progressing towards the unknown entities. The contents of the cards are filled by going through possible scenarios. Cards are added or changed when necessary. When there are too many changes written on a card to be useful, the card is replaced by a new card [29].

2.1.3. Unit Tests

To ensure that all components are working correctly, unit testing is required. Unit testing is the sole testing of one or more modules of a software application [30]. In Extreme programming, unit tests are made on the lowest level. The implementation under test (IUT) comprises methods in classes, a class or a cluster of related classes. A unit test starts with a test case, a specification of the input, pre- and post-conditions and expected result. Unit tests can be combined in a test suite which is often containing tests with the same test goal [31]. The actual result of a unit test can either be 'pass' or 'fail'. Failures of test can have various sources: a failure (output is not within the specified limits), an abend (abrupt end, such as a crash), an omission (systems lacks the capability to produce an expected output) or a surprise (system has a ability which was not specified in the specifications).

The unit tests created, are also the main part of the documentation of the application. It is suggested that no one is going to read a 200 paged book about the application; people rather look at the code. The unit tests assess all the features of the system and are a good way to understand the system. If the function of the test is known, the function of the IUT is easily derived. A document with a few pages can be used to describe the requirements and design choices made during the development.

The typical approach to unit testing consists of three phases: test planning, test acquiring and measuring the tests. In the first phase, a general plan is created called a 'test plan'. Tests are designed according to the test plan in the second phase. In the last phase, the tests are executed and the results are evaluated. Depending on the outcome, code in the application is changed, or tests are adapted. The unit testing in Extreme Programming is preferably done with the Test-Driven development methodology, which uses a different approach to the one just described. The next section will elaborate on this approach.

2.2. Test Driven Development

Test Driven Development (TDD) is a new approach for developing software which tries to solve the mentioned shortcomings of predictive development methodologies. TDD is one of the core practices in Extreme Programming (XP).

Test Driven Development is quickly becoming more popular and it is claimed that, once seriously tried [32], as a developer you do not want to go back to traditional development methods. The Test Driven Development approach tries to solve problems with implementing change requests, and tries to prevent overly long development and maintenance times [32], [33], [34].

The motto of Test-Driven development, according to R. Jeffries pithy phrase [32], is "clean code that works". This means that we are not trying to get a perfect model for our system, totally analyzed and thought through before the implementation starts. Instead, an iterative approach is used to expand the project's features. This makes Test-Driven development fit perfectly in the Extreme Programming and Agile Modeling methodologies.

Test Driven development is based on unit tests, which test small pieces of code such as methods. The main difference between normal unit testing and testing via the Test Driven methodology is that unit tests are implemented *before* implementing instead of after.

Test-driven development consists of three phases for every task which need to be implemented in the system. The first two phases are marked by a color, representing the outcome of the test when executed:

- Red – Make a test and see it fail,
- Green – Make the test work quickly, committing whatever sins necessary to get the test to pass,
- Refactor – Eliminate all duplication which was caused by getting the test to work.

As a rule, for every line of code written, the previous scheme should be used. 'If you can't write a test for what you are about to code, then you shouldn't even be thinking about coding' [35] is one of the basic rules for TDD. In extreme programming this is prescribed as "Test everything that could possibly break" [36]. Since the work for testing every single line of code seems astronomical a programmer should increase the size of the IUT every time. On the other hand, the size of the steps are to be decreased when a test is expected to work, but does not. The ideal test size is not fixed and depends on various factors, such as the difficulty of the code to be written, experience of the programmer, time of the day etc.

The Test-driven development is based on the idea that people do not test because people experience stress during programming [32]. The more stress people have the less they test, because testing requires work not related to features of a program. By testing, you have to prove your program wrong, which is causing extra work. In stressful situations, this is the opposite of what developers want. But, testing less also means that you are more likely to make errors, which results in a more stressful situation. The Test-driven methodology reverses this chain by creating tests before you code. By creating these tests, it is less likely that you make errors, which results in less stress. In this way, the descending spiral is inverted.

With predictive development, tests are created in the late phases of the development process, usually called testing phase. Due to approaching deadlines, the pressure on the people gets higher towards the end of a project. Commonly, tests are not functional requirements of a software application. This results in tests being dropped from the project implementation actions. Since a test adds no visible features to a software system the psychological value for the end customer seems to be lower.

As can be noticed, the Test-Driven approach to the development of software is largely developed based on psychological reasons. The Test-driven methodology does not try to improve the software quality by adding extra procedures to the software process. Instead, it creates an environment in which people do things, which increase the quality of the software process and product. Although the TDD process is not predefined, there is a software process as explained in Section 1.1.

R. Mugridge [37] found a similarity between the Test-Driven Development steps and the Scientific Method (in brackets) (Fig. 3).

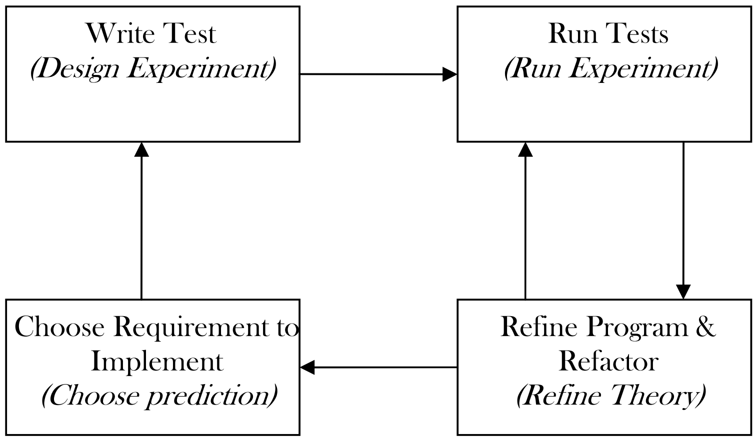

Fig. 3. Test Driven Development and the scientific method

Similar to the Test Driven methodology, scientists applying the Scientific Method use experiments to test if their prediction is right. The Scientific Method consists of experiments which are used to test predictions of a theory. The theories undergo series of experiments to see if the theory holds ground. The theory is changed to keep its consistency when an experiment fails.

The process is started by choosing a prediction, for example that a feather falls as fast as stone from the same weight. This is equal to choosing a requirement to implement in TDD. The next step is to design an experiment, for example stopping both a feather from 1 meter and measure the time. In TDD a unit test is written so test a method in this phase. Now the test is executed and fails. Now the theory is refined by saying that a feather falls as fast as stone from the same weight in vacuum. For TDD the implementation under test is refined in this case.

As seen in figure 3, the basis of the two theories is equal. However there are also noticeable differences if we compare both theories. The theory refinement in the bottom right step basically changes the output of the theory in a particular part, and program refactoring does not change the output of the program. We can only speak of similarity when a scientific theory is simplified, receiving the same result with less steps or calculations. The comparison can be a good help in understanding the success of TDD.

TDD is said to have several shortcomings or disadvantages. Lack of design [38], problems with applying unit tests, lack of documentation [39], reliance on refactoring, dependence on the skills of the programmer [40] and problems with scale [41] are commonly noted problems.

In combination with XP, TDD is said to have numerous advantages over predictive approaches. This thesis will verify some of the advantages of XP in combination with TDD. To make clear what are considered advantages, Section 2.3 gives a list of claims made concerning the advantages of XP/TDD. Furthermore a selection of the claims is made which are evaluated in this thesis. The result of this evaluation can be read in Chapter 1.

2.3. Claims

When Extreme Programming and Test Driven Development are combined, it is claimed that there are numerous advantages over predictive development approaches.

1. Despite all the changes in the development process, XP/TDD will not take more time than a predictive development methodology.

2. It will be easier to fix bugs when they are found. It is far less difficult to locate a bug which occurred after writing ten lines of new code than after adding a thousand lines. This will result in a relatively smaller time per bug fix.

3. The lower number of fixes and workarounds to be done in the final product will have a positive effect on the predictability of the development process' duration and less bugs in the final product.

4. TDD is supposed to be suitable for any software project.

5. TDD handles changes in requirements better than predictive approaches. In XP/TDD there is no full design made upfront, which has to be revised every time a change is requested. It is argued that this makes the development process more suitable for changes in requirements. Because of an iterative based process, which results in working prototypes, possible change request can be identified earlier. In predictive development methodologies a working product is only delivered to the end user when the product is finished. At that point, simple change can require immense amount of time. A result is that once a product is delivered, developers will try to avoid or ignore change requests. XP in combination with TDD

makes changes easier to implement with a better suited final product as result.

6. Developers will be able to look further with TDD, think of better solutions while programming. It is argued that you can only really think on better solutions for a particular situation in a program, when the code is being written.

7. The XP/TDD process is more enjoyable for the stakeholders. For the end users, the incremental delivery will stimulate the user in evaluating the project early on. In XP/TDD the end user is invited to request changes, which can be seen in the next release. This will stimulate the participation of the end user in the project.

   The shortened requirement analysis and development cycle will be more enjoyable for the developer. This is because developers mostly prefer starting with the implementation as soon as possible, since results can be seen quicker. Developing in XP/TDD will be more enjoyable for developers since every addition of code results in a fulfillment: a green bar is displayed which means that the test works.

   Since the tests in TDD will provide the necessary documentation, no bookwork of documentation has to be written. Tests are a replacement for documentation [39]. It is argued that such documentation is always out-of-date and that new programmers start with looking at the code instead of going through a large amount of documents.

8. Except for delivering a better suited product, supporters of TDD say that the quality of the code will also improve with TDD. Due to creating test first, a nearly 100% code coverage of tests will be acquired automatically. Bugs will be found sooner in the development process, decreasing the necessity for workarounds and fixes in the final product.

9. Finally, the main claim which is investigated in this thesis is a usability claim. It is claimed that it is possible to apply TDD in a project anywhere requirements are implemented. With minor adaptations TDD will be suitable for testing databases, interfaces and other technologies.

The main goal of this thesis is to verify these claims made by Kent Beck. In this thesis we will be looking at the last five claims made (claim 5 to 9), by executing a qualitative and quantitative analysis. The claims are evaluating the usability of TDD, which is the focus of this case study. With the last claim - declaring that 100% test coverage can be achieved - it is of particular interest if a programmer is able to continue to use TDD for the duration of the project. It is possible that as the project proceeds, programmers tend to skip TDD to continue with a test last methodology. In the next chapter an elaboration on how the claims are evaluated and what points are of interest is given. A clear product description of the case study follows. Then the development process is described and evaluated.

3. Case Study: Product Description

3.1. Why a case study?

The goal of this thesis is to verify if the claims about XP/TDD are justified. The evaluation of the XP/TDD methodology is conducted by means of a case study. A case study seems an ideal test-bed for the evaluation of the various claims which have been made. This is because a case study includes the requirements analysis and design phases, so valuable information from stakeholders about the whole process can be acquired. A case study can be split up in multiple iterations, which allows evaluating at intermediate times during the project. Various techniques can be applied giving a more complete view by broadening the investigation area.

3.2. Evaluation entities

After noting the assumed advantages of TDD over predictive development techniques, we need to specify how we can evaluate the claims made. The claims will be evaluated by using:

- Direct observation: Observation of the researchers during the development process,
- Interviews: Interviews and responses from other stakeholders during discussions and presentations,
- System requirements and test cases: These where made before starting with the development and used to verify requirements,
- System repository: A full repository with release notes was maintained during the development process,
- Schedule: A time schedule was set up before development was started.

Reflected onto the claims, this will lead to the following evaluation structure:

1. Direct observation and interviews are used to evaluate the usability claim. In the case study, several aspects of implementing a software application are applied. The goal is to get an overview of where TDD is usable and what can be done in the areas TDD is not or less usable. Is, for example, extra tool support a solution, or do we need an entire different way of implementing? If there are proposed solutions provided by others, they are also discussed with noting advantages and disadvantages.

2. Incorporating changes should take a short time in TDD. Change requests which emerge during the implementation should have limited impact on the source. This can be measured by evaluating how much work has to be done, considering the expected impact of the change. For each change request it is checked when the change request is issued. Questions are asked such as: Is it possible that this change request could have been detected earlier? What was essential to the detection of the requirement change?

   Except for looking at the change request itself, I will look at the impact of the change on the system. For example the time required to implement the change is estimated and compared to the time the same change would have cost in a later stage of the project.

3. The project lifecycle will be analyzed to evaluate if programmers can look at better solutions with TDD. I will look at which solutions are found to solve technical problems on an architectural level and on low level programming. The overall project lifecycle has to be evaluated: the direction the project goes and the changes to this direction. How does an early delivery affect the decisions of the stakeholders?

4. It is claimed that the XP/TDD process is more enjoyable for the stakeholders. For the case study, several types of stakeholders can be identified. Three stakeholders are considered in the case study: programmers, new team members and customers. Via interviews and feedback from each of the stakeholders a conclusion is drawn about if TDD/XP is more enjoyable for the stakeholders than predictive approaches. Also, direct observation will be used to evaluate the programmer's opinion. After finishing this research and my involvement in the conducted case study, the project will continue to move on and will be taken over by new developers. This creates the opportunity to see how quickly new developers can

understand the project without having an extensive project documentation at their disposal.

5. Except for delivering a better suited product, supporters of TDD say that the quality of the code will also improve with TDD. Due to creating test first, a nearly 100% code coverage of tests will be acquired automatically.

   Since we do not have another parallel project running which could be developed based on a predictive approach, it is not possible to get a clean comparison for both development approaches. However it is possible to compare own written software with components used from other vendors to get an idea about the differences in code properties and program structure between the two approaches. Another way to measure the quality of the code is to use code metrics. Code metrics are static information extracted from the software product. They are used to get a notion of several software properties, such as code complexity, dependability and size. The code metrics can be extracted from the case study and compared to other software applications. It is not possible to draw hard conclusions from this comparison, but it can give us a general idea of the quality of the code written with the TDD approach.

   Code coverage has to be measured with a code coverage application to find if coverage of 100% by the written tests is achieved. The case study supplies two separate software products which can be measured and compared regarding code coverage.

3.3. Forecasting web application in .NET

This section gives a brief description of the application which was built during the case study. A full elaboration of the project which is developed is given in Appendix A.

3.3.1. Current situation

EPCOS is a manufacturer of passive components for electronic devices with more than 40,000 products on sale. External sales representatives work together with customers on the design electronic products which embed EPCOS' products. The products are often low priced (less than 5$ per unit) and sell in enormous quantities. Due to the number of product types and product prices, the number of projects EPCOS is involved in is very high compared to other industries.

EPCOS uses Excel sheets and E-mail to exchange information between sales representatives and the sales department. Sales representatives have to send an annual budget update. This is done via a large Excel sheet which is sent to the different regions of the NAFTA countries (Canada, USA, and Mexico): west, middle and east. Information concerning product forecasting is gathered via phone conversations and email. The marketing department makes a decision and sends an order to the factories. The factories investigate what they can produce in which time span and decide in accord with the marketing what is produced and when orders are ready.

For budgeting, a system called OPAL (Orders, Planning, Analysis and Logistics) is used. OPAL is a tool used for sales, marketing and logistics. Before the start of the requirements analysis, a pilot application is created in the form of a Microsoft Access application. The Microsoft Access application is used for budgeting and the original plan was to extend this application with forecasts. The problem with the Access application is that it is not easily usable via the internet without access to the EPCOS internal network. This is the reason why a new project is defined to allow external parties to input information.

3.3.2. Project vision and scope

Forecasting application will be used to budget and forecast orders. At the moment, forecasting is done on a yearly basis, which makes it inaccurate. The goal of the forecasting application is to increase forecasting accuracy by increasing the frequency representatives update the forecasts. Another goal of the forecasting application is to organize all the forecast information to enable to track forecasts and to provide easier access. After this is done, the marketing department of EPCOS can make decisions on the number of parts to be ordered at the EPCOS factories. The factories should be able to view and confirm the forecasts placed by the marketing department.

By forecasting in advance, production can already start before a real order has been placed. This lowers the delivery time. Currently, production starts after an order is received or when a sales representative sends an email about an expected order. This communication is unstructured and often important information is not communicated to the sales division. An important improvement is to make information more transparent inside the company. This should be done by standardizing communications processes between sales representatives, sales headquarters and production facilities. Sales representatives can indicate how many orders will come in the near future. The plan is to make an application where representatives can input forecasting data after visiting a customer. The marketing department can anticipate on this data and instruct the production facilities to start production, before the real order is placed. When a customer finally places the order, production is already running or finished.

Forecasts have to be assigned to projects with several properties, adding another requirement. The application should also be able to create new projects, maintain running projects and save the history of project changes. Projects have to be filtered on project properties such as customer name. During the development, several other project managing requirements where added, hence the application was named “EPCOS Project tracking and Forecasting (EPF)”. This name will be used in the rest of this thesis to identify the application.

3.3.3. System Architecture

The system to be developed consists of three layers: a client layer, a web interface layer and a data services layer. In software architecture this model is commonly known as a three-tier architecture. A tier is a partitioning of functionality and can be deployed at distinct psychical machines [42]. The layers in the system are implemented in the EPF application as follows:

- A client application, which interfaces the end user and executes operations concerning project tracking and forecasting,
- A web service, interfacing the database for the client application and providing authentication features,
- The database, providing necessary storage.

These components will be evaluated as separate entities with the various claims made in section 3.2. This is done because it can be expected the development and runtime properties differ, which can be of influence of the result of the evaluation.

3.3.4. The client

The architecture of the client application is displayed in figure 4. The blue arrows represent the GUI thread and the red arrows the data loading thread. The controller object is the initiator of the application, creating the threads. The data loader thread starts with creating a User Manager object, which loads the rights of the user using the web service. After the right list is received various container objects are initiated depending on the right of the user. These cache objects receive information from the web service, which can be used by the GUI components. The connection set up and encryption is facilitated by the Connection Controller.

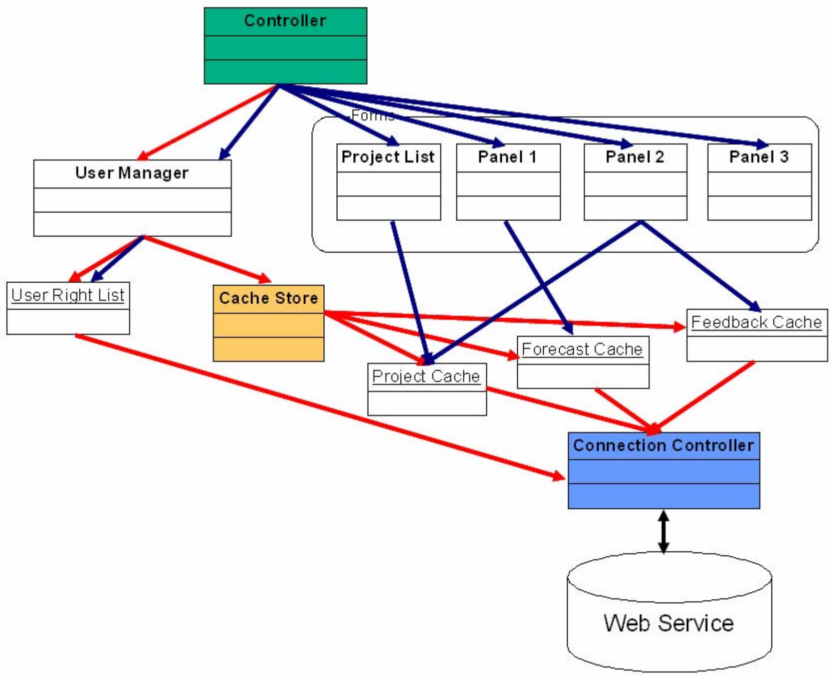

Fig. 4. Client architecture

The GUI thread starts the main form and looks at the rights from the user. Depending on the rights of the user, various panels are created. Each panel knows which rights are required to be able to launch the panel. If the rights do not match, the panel is not loaded. The panels receive their data by calling the cache objects. When the cache objects are filled they return the data received from the web service, else the cache object waits until it is finished loading the data from the web service.

3.3.5. Web service and Database

The web service, depicted in figure 5, consists of three layers. The first layer listens to requests, (un-) marshals objects and encrypts messages. The most prominent layer is the operations layer, which includes the logics to receive information from the database and make it suitable for returning it to the client according to the data model. The third layer, the database manager, facilitates the database communication by creating the database connection and executing queries.

Each of the layers is secured by its own security module. In the first layer this is done by using a custom implementation with help of a component from Microsoft named WSE (Web Services Enhancements). The security in the second layer is applied by using AOP (Aspect Oriented Programming). Before each function in the web service is invoked, the security clearance is checked by the aspect. The method intercepting is done with help of an AOP framework. If granted, the method is allowed to execute, if not a Security Exception is thrown.

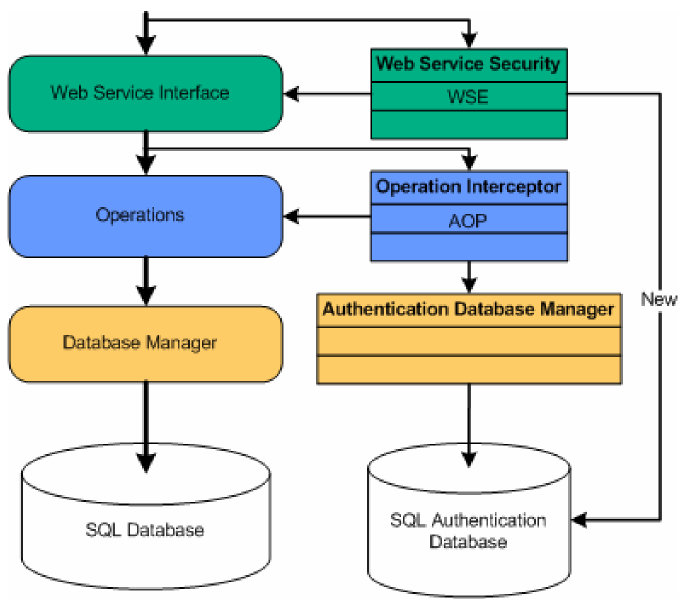

Fig. 5. Web service structure

4. Case Study: Development Process

In this chapter the development process will be described. The development process in XP/TDD consists of an initial phase and several iterations. The project planning will be elaborated in the next paragraph. It follows a description of the initial phase, which consists of several additional activities when compared to the other iterations. The additional activities include requirements analysis, creation of a test strategy and fault model and the definition of a global architecture. The other iterations include the reevaluation of the existing requirements, requirement selection and implementation. This chapter is an example of a possible TDD development process, with the purpose to give an inside look on of what a TDD development process compromises. Note that the initial definition of a test strategy and fault model is not purely a TDD practice. The decision to include this in the case study was to support the evaluative study of TDD.

4.1. Project Planning

The project development is separated into three iterations. Before the first iteration, a requirements analysis is done. This incremental approach is a common methodology in the field of software engineering. For each iteration, specific testing methodologies will be applied. The overall requirements analysis includes study into the Test Driven approach and other technologies which are to be implemented in the system. This phase also includes orientation and project planning. In each iteration around 3-10 requirements are implemented. The first iteration took around four months to complete, because it includes the initial set up of the programming infrastructure. Furthermore, the first iteration consumed up extra time, because the framework of the program had to be developed. The other two iterations took around two months. In these iterations the basic framework was extended to accommodate more requirements. Finally an overall evaluation was performed during the last three months of the project. The total project planning is displayed in Table 1.

Phase	Duration	Period
Overall Requirements Analysis / Study	1 month	March
First iteration	4 months	April - July
Second iteration	2 months	August - September
Third iteration	2 months	October - November
Overall evaluation	3 month	October - December

Table 1. Project planning

4.2. Requirements Analysis

The requirements analysis started with identifying the users of the system. After interviews, the following users were identified:

- Sales representative users, outside of EPCOS inputting forecasts and creating new customers,
- Area managers, top sales managers (later removed due to changing requirements),
- Marketing users, working inside EPCOS inputting forecasts and creating/changing projects,
- Factory users, working inside the EPCOS factories confirming production,
- Administrators, maintaining the system.

For each of the user types, functional and non-functional requirements are determined. Also, there are several requirements which apply for all users called “Global requirements”. Following the TDD approach, each of the requirements has been supported by a test case (marked as TC). Similar to implementing according to the TDD approach, the test case is specified before any implementation is done. The test cases are used to clarify the requirement for both the programmer and customer and are used to specify a full use case. For the complex components some basic screenshots were made and were presented to the user. Since there is no full upfront design process in TDD, the test cases can avoid starting of in the wrong direction. In practice, the test case is quite equal to a simple use case diagram. An example of a requirement is listed in figure 6.

Global Requirement – G1

Login: All actors using the system should be authenticated with a username and password.
TC: Enter username, password; confirm || If password is correct: the login entity in the software system equals the entered username. If password is incorrect: the user should be notified and not allowed further access to the system

Fig. 6. Requirement with test case

After gathering the main requirements, a basic architecture was specified. The technologies were discussed with the customer, giving an understanding of the pros and the cons of each of the technologies. Initially there was a tension to use web pages for the interface, but later it became clear that there was too much data to be sent over the internet every time the end user changed his selection to make this a successful strategy. Other reasons were that the network infrastructure in the company was not sufficient for that and that offline working to some extent should be possible. The customer also liked the possibilities provided by web services, such as caching and selective up- and downloading of data so save connection bandwidth.
Before the selection of the requirements to be implemented in the first phase, a test strategy and a fault model are specified. A test strategy and fault model help to identify possible fault types and specify possible ways to test on these faults. After a global test strategy was created requirements had to be selected and ordered per iteration. To prioritize the requirements, several interviews with future users where held. As soon as there was a mutual understanding of the requirements for both, the users and the programmers, the implementation commenced.

4.3. Test Strategy and Fault Model

Based on the architecture and the data flow in the project, we can establish that the data displayed to the user is going through various components. Some components restrict which information is sent, depending on the user. The application has to deliver the data to the user correctly and restrict where needed. To guarantee the correctness of the application, tests will be executed on different levels with different goals. Strictly spoken, everything what could be tested should be tested with TDD. Due to the enormous amount of time required to test all what we can possibly think of, only the most important/likely fault scenarios will be considered in the next section.

4.4. Fault Model

According to [31] and based on the use cases, there are some general faults which can occur when creating applications:

- Undesirable feature interactions,
- Incorrect output,
- Abnormal termination,
- Inadequate response time,
- Omitted capability,
- Extra capability (a surprise).

Use case specific faults are:

- Domain faults: code that deals with a boundary condition is incorrect,
- Logic faults (*and/or* switches and bracket mistakes),
- Incorrect handling of do not care conditions,
- Incorrect or missing dependency on states established by prior use cases.

Furthermore, each layer in the application can generate its own environment specific faults.

In the first layer, the OPAL and SAP files are imported in the database via stored procedures. Since these files are updated every day and have influence on the database content, conformity has to be tested. This concerns both the input and the output of the stored procedures. The stored procedure has to check on the validity of input files and take measures when this file is invalid. If the file is imported, the database can be left in an inconsistent state causing faults in subsequent operations.

The next layer concerns the connectivity with the database. Problems can occur with the login procedure, requiring a username and a password. Even if the user is logged in correctly, it might not have the right permissions to select or modify information in the database. Since the database is on a different machine as the webservice, connectivity problems can occur. This can happen when the database is not available at all, or when it cannot process transactions. It is assumed that the data transfer itself is without failures, since TCP error correction is used in transferring the packets over a LAN connection.

The webservice interface is the most complicated layer in the application. The data communication over this line has to be authenticated, private (encrypted), reliable and secure. The testing of the security of this layer will be discussed in a separate chapter. General faults are connectivity faults, same as for the database. The difference with the database is that problems, like limited connectivity, are more likely to occur. This is because the sales force is using the application from the Internet, which is a more unreliable medium than an intranet. Another problem can be caused by the web service communication message itself. Apart from the consensus that has to be made between the client and the server concerning the encryption, the client has to be able to interpret the data structures sent over the link. C# provides checks for parameters and return objects. However, in this application we make use of Datasets, which can contain different tables. The C# language does not provide any check to see if the Dataset contains the data we expect. This is a potential source of failures, which can go as deep as the database level. It is difficult to avoid this problem, since some C# controls, such as the dataGrid, only support linking with Datasets. Creating a custom control will take much more effort and is beyond the scope of this project.

The authentication can also generate problems. When the client logs in with an incorrect password, the server can throw errors. These errors need to be handled by the client. Other problems will occur when the client calls for a method on the webservice, while the user does not have access to it. This should not be possible since the functionality of the client is based on the rules retrieved from the client. However, an incorrect rule set can lead to incorrect calls. Even if the use should have access to this method, there can be something wrong with the permission checking denying the user access to the specified method.

The interface can generate another kind of problem. For response time improvements, the data displayed in the interface is buffered. This can only be done for viewing information about projects. When updating information, other users might be in each others way. The exceptions which can be expected are (in this phase):

- Updating/Changing a forecast while someone else is changing the same forecast,
- Changing a forecast which is just about to be more than one week old (this is not allowed).

More problems can be expected with features such as project addition, changing and deletion.

4.5. Test Strategy

Testing is the paramount paradigm in TDD and is not only used as validation technique but also as specification and documentation. In this section the strategy on how to assess the application is described in detail. The described test strategy has two functions: it is a product specification and technical product documentation. The first function, product specification, describes the requirements of the application by specifying test cases and focuses on potential risks related to the fault model, presented in section 4.4. The second functionality of the test strategy is to supply technical product documentation, for third party components for which a comprehensive documentation is not available. The test suites represent the most important documentation for their usage. For our own components, unit tests supply the necessary documentation.

The general test strategy consists of two parts: low level testing with the Test Driven methodology and high level testing. High level testing concerns testing the functions of a software application on a system basis and check if the functions match the requirements. Because of the high priority of security in the application, additional measures for testing the security of the application have to be taken. For example, the communication between the client and the server has to be monitored to evaluate if messages are securely sent.

4.5.1. Unit testing

Unit tests are the core of Test Driven Development. In object oriented programming unit tests evaluate the classes of a program [50]. Automatic unit testing is done with help of a unit testing framework. The most commonly known and applied unit testing framework, xUnitTest, is implemented in various languages. For the Microsoft .NET framework version 1.1, NUnit is available as a unit testing framework. Due to the increasing popularity, the new 2005 release of Microsoft Visual Studio (the development environment of Microsoft .NET) contains an embedded unit testing framework [51]. Since at the time the development started, the new development environment was still in the beta stage, decided is to test with NUnit. For quick testing a plug-in for Visual Studio, called "Test Driven .NET", was used. This plug-in allows compiling the project and executing tests within a single click and debugging tests by stepping through the code. Tests can be created either as inline methods of a class or in separate test classes. The advantage of having inline methods is that the test cases can access private variables; the advantage of having test classes is a better separation of implementation and tests. An example inline test is as follows:

```
[Test]
public void testGetCustomers()
{
     DataTable customers = getCustomers().Tables[0];
     Assert.IsTrue( customers.Rows.Count > 0, "There are no customers re-
turned" );
     Assert.AreEqual( "Hans", customers.Rows[0]["CustomerName"].ToString(),
"The first customer is incorrect" );
}
```

The [Test] attribute describes that the method is a test and thus recognizable by automated testing frameworks. In the test first the action is executed and the result is verified. The NUnit framework provides several classes to assess variables. The most important one is the class "Assert", which is used in the code sample.

Visual Studio offers a possibility to mark certain regions of code which are not compiled in a special mode, called "release mode". In this way tests can be left out of the executable and thus do not influence the behavior of the system, which is distributed to customers. Also passwords to databases, which can be used in tests, are not compiled into the application which is released.

To use Unit testing for testing the webservice, the initial plan was to create a tool which sends and receives the web service SOAP messages. James McCaffrey from Microsoft cooperation created a small automated webservice testing utility [54], which uses the SOAP messages as input. After creating the unit tests via the Test Driven methodology, unit tests which verify the SOAP messages, were written. The automated testing application had to be adapted to support encrypted messages in a later stage. Later in the project, there seemed a more obvious solution available: testing the web service in the client. This is described in paragraph 5.5.4.

4.5.2. Use case testing

The use case testing is done via the Extended Use Case test, mentioned in [31]. The use cases were specified with use of test cases in the requirements analysis. This section elaborates on the behavior of the application concerning the test cases. The operational relation of the use cases are displayed next

Use cases:

1. Server side,
2. Internet Login (As Sales user),
3. Intranet Login,
4. Get an overview of projects.

Use Case	Username	Signature	Access	Expected Message
All	DC	Invalid	DC	Communication with the server failed: Reason invalid signature.
All	Valid	Valid	No	This user doesn't have access to this method.
Login only	Invalid	Valid	DC	This user doesn't have access to this application
Others	Invalid	Valid	DC	Display: your account is invalid.
All	Valid	Valid	Valid	Execute use case

Table 2. Extended use case test, use case 1 to 4

Use case:

5. View actuals from a project

	As above	Project #		
	All Valid	Valid		Show the actuals
		Invalid		The project selected is not available.
		Null		Please select a project

Table 3. Extended use case test, use case 5

Use case:

6. View forecast

	As above	Project #	Forecast avail.	
	All Valid	Valid	Yes	Show latest the forecast
		Valid	No	Show white fields, ready to fill in
		Invalid	DC	The project selected is not available on the server.
		Null	DC	Please select a project (should not happen)

Table 4. Extended use case test, use case 6

Use case:

7. Insert/edit forecasts from a project

	As above	Project #	Forecast > month	
	All Valid	Valid	Yes	Create new forecast in DB, show latest
		Valid	No	Update latest forecast
		Invalid		The project selected is not available on the server.
		Null		Please select a project (should not happen)

Table 5. Extended use case test, use case 7

4.5.3. Security testing

Due to the priority of the security of the application, various security measures have been applied to the system. When we analyze the application from back-end to front-end, the first security measure applied is the Aspect # code. This aspect-code is enabled before and after the methods are called. This provides a level of security. While implementing the classes which are called via the web-service, the Aspect is always called first to check on permissions. However, it needs to be guaranteed that the Aspect is always called. So for each test for the web methods it needs to be checked if the Aspect is invoked. It is possible to do this by invoking the method with an invalid username, a username which doesn't have permissions and by invoking it with a correct username. This can be done at two levels: testing the method via the Test Driven methodology and by testing via the webservice testing tool. By following the exact same way by implementing a new class, we can test with help of the Test Driven methodology, if there are no easy ways to bypass the security added by the Aspects.

Another security issue necessary to check is the encrypted SOAP messages. After implementing the encryption in the SOAP webservices, it must be assessed that the webservice does not support unencrypted messages anymore. Moreover, the encrypted messages need to be tested by omitting viable information about the security. The key elements needed to pass the security check for the web service are as follows:

- Encrypted Username token,
- Encrypted method request,
- Signature,
- Signature is valid concerning the message, user token and password (hidden).

And on the return (client) side:

- Encrypted response,
- Signature.

All these elements are mandatory and can be tested by omitting or invalidate one of the elements at a time. The return side can be analyzed with the testing tool, to see whether the response is valid and encrypted.

The security tests which have been implemented according to the Test Driven Development methodology are listed in the table below. The column headers have the following meaning:

- NT Test: Test which applies to NT users as well as external users
- SQL Security: The user is correctly added to the SQL users
- Signature: Valid signature is sent
- DB, Username: The username matches the password in the database
- DB, User Right: The user has permission to execute the function
- DB, Password: The user password is correctly added to the database

Test Name	NT Test	SQL Security	Signature	DB Username	DB User Right	DB Password
RightsUsername		X	X			
RightsPassword	X	X	X	X	X	
RightsSQLSecurity	X		X	X	X	X
SendHashLogin		X	X	X	X	X
PlainTextAndSignature		X		X	X	X
PlainText		X	X	X	X	X
UserRightLogin		X	X	X		X
Correct	X	X	X	X	X	X

Table 6. Security tests permission grid

These security tests should all fail (except for the last one) and are used to test all the aspects of the security applied in the web service and in the database. The marked test, "UserRightLogin", is a critical test accidentally omitted by the Test-Driven methodology and was added later. The checks in the table mean that this property or variable is correctly supplied to the webservice. The checks in the NT Test column display if there is a test implemented which uses NT Authentication. All tests have been implemented on the web client and have been executed every time a new feature was implemented.

4.5.4. Integration testing

The following diagram will assist explain the deployment testing:

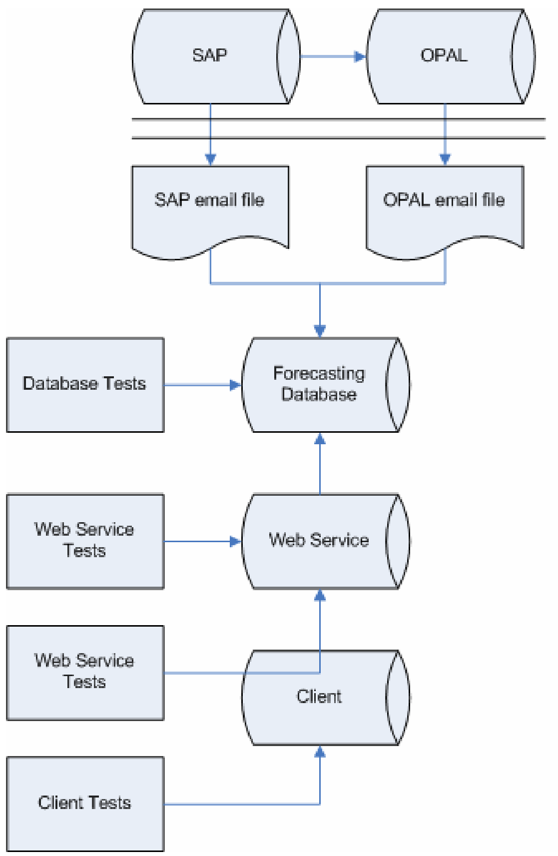

Fig. 7. Basic testing scheme

As noted before, the application will be deployed from top to bottom. The data which is originated from the SAP and OPAL databases is funneled through several gates and operations before they arrive at the end user. For the last two gates, the web service and the client, strict security has to be applied. The main idea is to test the application in phases, comparing input and output data while testing. The original plan was to test the database with TDD, but this was not easily possible, as will be explained in chapter 5. The web service is tested twice using TDD: first in the web service itself and when that works on the client side.

In this way the scope of the implementation under test slowly increases, to view if changes are applied according to the requirements and to view if the security measures still work accordingly. The tests are used to check if the contents of the information transported, is equal to the data in the database. Furthermore it will also look out if the security permissions are correctly set. This is done with methods described in sub-section 5.5.2.

4.6. Implementation and beyond

After a global test plan and a basic architecture are created, the development process continues with the implementation. As mentioned, the implementation was done according to the TDD approach: implement a test, make the test work as soon as possible and refactor the system. In the beginning small refactorings are made, later they are sometimes closer to redesigns. New technologies are tried out when necessary and results are reported to the end users.

While implementing, various statistics are gathered to monitor the use of TDD during the development lifecycle. For each change in the source code of the application, there is noted what the purpose of the change is. By use of a revision control system, it is also possible to get an estimate of the time spent for committing this change. This will help get an idea of the use of and the time spent on TDD during the development process.

Every time an iteration is finished, the application is deployed and shown to the users. In the first iteration the application was only delivered to one user to review the basic features. It became clear after this iteration that the forecast panel did not contain the minimum set of features in order to distribute the application. Also several project management functions were not available to allow marketing users to manage their projects. These functions were originally not part of scope of the project, but would deliver valuable features for end users. Because of these reasons the requirements for the second iteration were reprioritized and implemented. The same process was executed for the second iteration. In the second iteration, the application was deployed on a platform of another user. The deployment consisted of the installation (and testing of the installation) of the application and the importing of the current projects of the user into the new application. To import the projects we created a template which had to be filled in by the user and was uploaded to the database. During the third iteration more users were added and project deliveries became more frequent, mostly after the completion of each feature. To be able to let end users make use of the application, several small features had to be added. This diminished the need of creating a requirements priority list, because requirements emerged which were crucial for the acceptation of the application. Basically, the iteration was completed but some essential requirements were forgotten during the development process. This considered mostly small extensions to current use cases, such as displaying a small descriptive text when hovering over a control on the screen.

Besides functional requirements, there were also non-functional requirements implemented. While working with the user we noticed that the application had a high response time on slower systems while executing some actions. In the third iteration some major speed improvements were implemented that improved the timely behavior by more than 300%. After the completion of the case study the project was taken over by other developers. There are still improvements possible, but the focus is again on functional requirements.

5. Evaluation

The case study discussed in the previous two chapters represents the basis for the evaluation of the claims made in section 3.3. Before the claims are evaluated, we will look at several aspects and impacts of TDD which were noticed during the performance of the case study. This chapter establishes a theory regarding the effect of changes in requirements or changes in the implementation on the time spend implementing changes in TDD (section 5.1). The theory presented has an impact on several of the claims made, in particular the claim concerning changes in requirements. This is one of the spear heads of TDD and XP: the first book about XP even encloses “Embrace change” as subtitle [5]. The second part of this chapter is about the lifecycle of a TDD process (from section 5.2 on). During the implementation, data about the development process was recorded. The recorded data is summarized and analyzed and compared to what was expected before the case study was performed.

The main part of the evaluation is a summary of the experiences while developing and implementing the components of the software system. The focus here will be on the usability of TDD for testing the components. While developing, we came across problems to apply TDD with some of the technologies used in the case study. After the description of the problems for each component, I will discuss the lessons learned during the development and the possible solutions for the problems encountered. In particular, I will discuss a technique to perform unit testing by using aspect oriented programming [55] which can provide solutions for various problems with TDD. This chapter is then finalized with the evaluation of each of the selected claims, which are given in Section 2.3.

5.1. Usability - General

Before going into the case study itself, some general theory came to the attention while working with TDD in combination with unit tests. TDD proposes to test everything what one can think of. After making a test, we have to make it work as soon as possible. Finally the implementation under test (IUT) is refactored.

Tests should be testing atomic pieces of implementation to facilitate changes of the implementation. Figure 8 shows the effect of changes in the IUT on changes in tests. The left side of the figure displays three tests checking three functions of two IUTs. The right side displays the situation after one IUT is changed. In figure 8 it can be observed that multiple tests have to be updated if they test the same IUT and a change is made in the IUT.

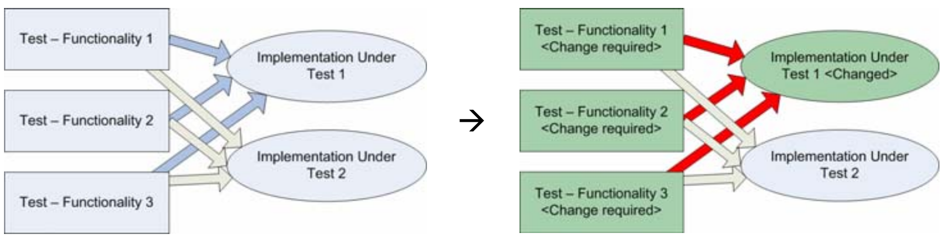

Fig. 8. Test dependencies

To optimize the flexibility of the tests when changes occur, tests should be independent from each other. This means that one IUT is only assessed by one test, so that when this IUT is changed, only one test has to be adapted. Now imagine the two IUTs and two unit tests, with each test assessing one of the IUTs. In this way, if one IUT is changed, only one test has to be changed. At this moment in the implementation, the first IUT is changed to call the second IUT. For example, the first IUT retrieves a value from the other IUT which is provided by the user. Now the first test checks functionality in both the implementations. The second test still only assesses the second IUT. This caused that, when a change in the second IUT is done, both the tests have to be updated. However, this does not mean that the second IUT has to be adapted if the first IUT is adapted. The problem is that if in any way the result of an implementation changes, the test fails and the test has to be adapted, since tests use instances to check a function. However other IUTs, which call IUT$_y$, do not necessarily have to be changed.

This can be illustrated as follows: In many cases the adding of a record in the database has no impact on calling IUTs. This is because TDD enforces us to refactor. The tests however, can be using the record count or number as a test entity. This states that given the TDD practice, the higher the coupling between IUTs and the more entities we test on, the more tests we have to change if we change an IUT. The result is that tests can slow down changes. It is to be expected that testing first can worsen this, because there is no pre design for a test suite. Therefore, we can not predict on how a test suite should be designed and which entities should be used for testing and which should not.

In practice it occurs often that multiple tests fail because of a change in one of the IUTs, especially when using a layered system such as a web client. Most of the tests in the client test functions use data from the web service to verify functionality or a method in the client. The web service retrieves the data from the database, which can be either a real or a test database. A change in either the database or the web service leads to a different output of a method in the client, which leads malfunction of one or more tests in the client. This makes it very difficult to maintain all tests, especially when many changes are implemented and is something to be considered when creating tests.

We will now go into the usability of TDD during the lifecycle of the software development process. Statistical data is used to analyze the time spent for various development activities. After that, the various components developed in the case study are evaluated.

5.2. Usability – Full lifecycle

The first question to answer is if developers are able to use TDD and are able to continue to use it. Since there is not a predefined development process enforced, the focus is on using the TDD as a practice. But if it is too much effort to use TDD or the stakeholders make the use of TDD difficult, developers will eventually let go of the development practice, unless strictly enforced. This is especially the case in small projects, where mostly no strict standard processes are defined.

To get an idea of usability of TDD over the lifetime of a project, every time a phase finished the changes were committed to the source repository. The commits in the repository have been monitored for six months in the development cycle.

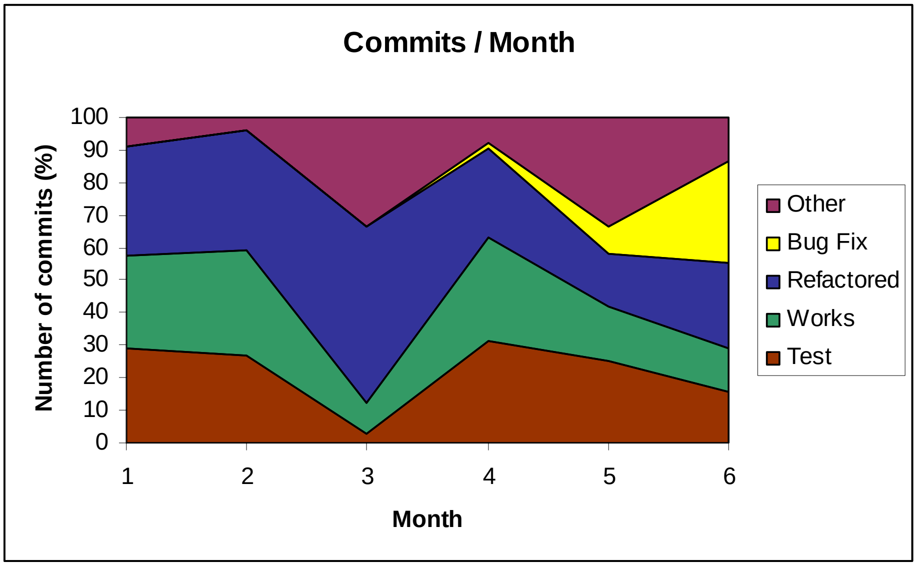

Fig. 9. Commits per month (percentage)

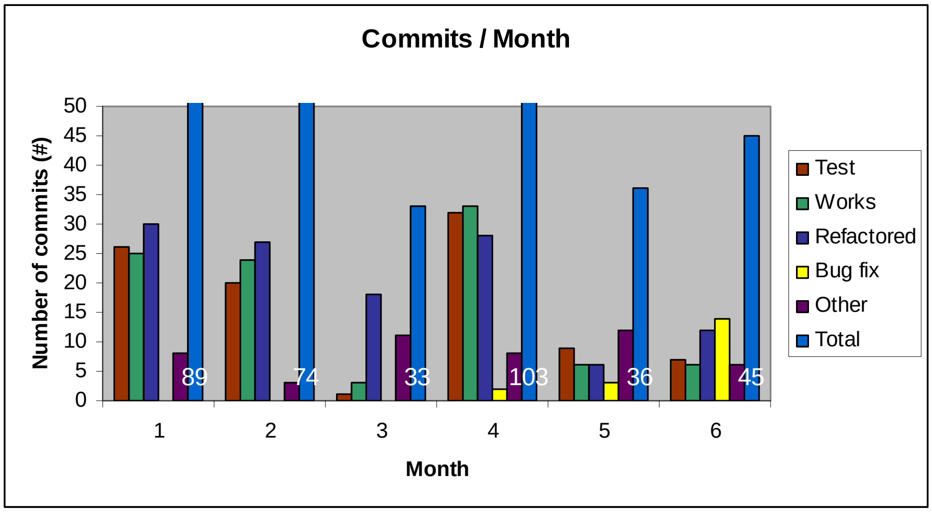

Fig. 10. Commits per month (quantity)

Figure 9 and 10 display the number and the percentage of commits per category. Please note that the 3rd month is not a very reliable source, since it was the beginning of a new iteration and time was invested for planning and researching for this thesis. As can be seen in the charts, the Test/Works/Refactored slightly go down making place for bug fixes. It can be seen that in general the TDD practice is still being adapted, but more time has to be invested to fix bugs. From a programmers perspective the charts give a good image of the reality: during the lifecycle the Test/Works/Refactored paradigm is selectively omitted. This is depending on the implementation under test, for example interfaces which are hard to test. This can also be seen in Figure 11 in which compared with Figure 9 the time spent for testing went down from 18% to 6%. In comparison, the tests commits only went down from 29% to 21% resulting in a lower time per test implementation. This strengthens the idea that later on in the project fewer tests are made for those parts of an application that are more difficult to test. In the next sections (section 5.3 and on), the parts which are hard to test will be identified. Another, but unlikely, cause of the lower time per test implementation could be that the programmers got more experienced making tests. The case study did not provide enough information to be able to verify this. To draw conclusions, more research has to be done on the time spent on creating tests when programmers gain more experience.

The other two categories, time spent on fixing bugs and on miscellaneous activities, display an expected flow. Time spent for bug fixes is increasing during the life time of the project. The final months are largely impacted by the "other" category which includes adding/removal of projects, adding of comments and testing of possible technologies to embed.

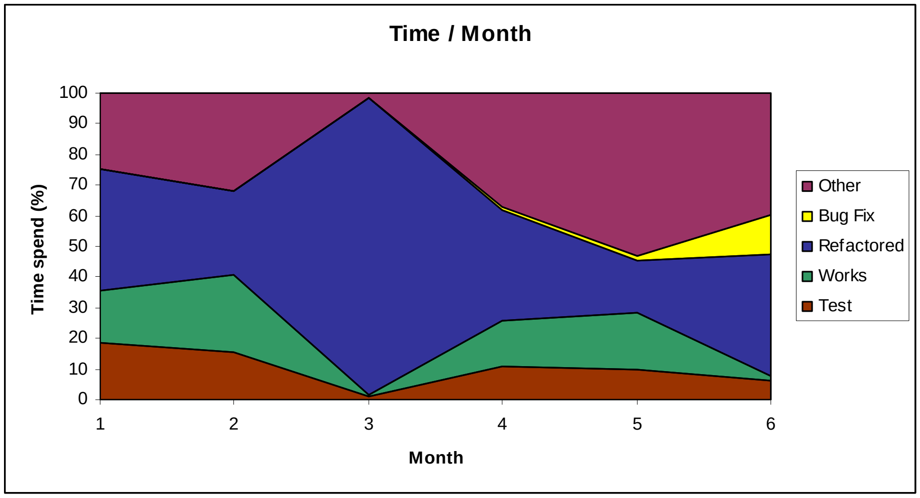

Fig. 11. Commits per month (percentage)

Overall the TDD practice seems to be used continuously during the duration of the case study. To verify if the decrease of the time spent in Test/Work phases is caused by selective use of TDD by the programmers, the TDD usage will be evaluated per component. The following paragraphs will also discuss the usability of TDD for each of the three main components specifically (Database, Web Service and Client). It is assumed that TDD works in every part of the development process. Described are the parts were TDD fails or has to be adapted to cope with the technique or code part. In some cases TDD might not be a good solution at all. We consider TDD to fail, when: (1) it is technically impossible/extremely difficult to test an IUT using unit tests, or when (2) the time required creating a test is not in relation with manual verification of a feature, in respect to the complexity of the implemented feature.

This is for example the case when testing the temporal visibility of a dialog which is shown at startup of an application. In the case study this situation occurred when a loading screen is temporary displayed to the user. After the loading was completed, the screen is hidden again. Manual testing of this feature is very easy, but verification by creating a unit test is very difficult. It either requires a lot of time to write a suitable unit test or it is not possible at all (when the dialog is shown and hidden in the same method).

In the next sections the experiences with developing the components in the case study are described. We start with the database component, followed by the web service component and finally the web service client. For each component, problems concerning TDD are described and possible solutions/recommendations are given.

5.3. Components: Database testing

The first step in the implementation of the case study was the development of the database. According to the Test-Driven development procedure, we should first develop a test which tests some feature in the database and we should test all features. This can be for example: testing if it is not possible to add an unknown customer to a project (foreign key test). In this way it is possible to test all foreign and primary keys and to generate a similar database design as developing a database by starting with a database design tool. By using the Test-Driven methodology for database design, we can utilize the advantages of the Test-Driven methodology. The small steps in the development and the quick results satisfy and stimulate the programmer.

While experimenting with using the Test-Driven methodology on database development, some problems were uncovered. The tests used to assess the database were written in C#, one of the .NET languages. The standard database adaptors require making a connection with the database, writing the SQL statement and storing the result in a 'Dataset'. After that, the result can be compared with the expected result or exceptions can be analyzed. This is the usual way for creating unit tests. However, for the database design, this is a time consuming operation. For example for every foreign key test, an "Insert" has to be done and the generated exception has to be analyzed, since C# only throws exceptions of type "SqlException". In practice, this means that the developer spends around 90% of the time on creating tests and only a few percent on creating the actual database design. This is largely caused by the lack of support of C# for testing databases. Even two datasets have to be compared manually, since a pre-implemented function is not available. With a proper tool the tests could be created without much effort. In the current state, it is not advisable to test primary and foreign key constrains using C# simply because the risk/effort ratio is too low.

A final problem concerns testing with a database as back-end. Most of the tests are based on accounts or other data stored in the database. The data for these test cases have to be added beforehand, because it takes a lot of effort to add and remove records in the database (within a test) every time a test is run. It can also be a dangerous job to remove records with tests, once the database is filled with real data. Adding test data to the database implies, however, that there is non-valid data in the database that can result in a security threat. Furthermore, when the database is filled with the official data, the tests will fail. If extensions are added later on, the omission of these tests can cause serious risks concerning the Test-Driven methodology and the security and stability of the application.

A solution could be using a special test database, which is switched if a test runs. The disadvantages mentioned above disappear by using this scheme. However, investments need to be made to create and maintain the database. If the real database changes, the test database has to be adapted too. Mostly it is convenient to add some example records in the test database, so the test framework does not have to create the same records for every single test case. The maintenance of the test records should be addressed in team discussions, since every part of the program uses these records. Defining and extending the default test record set is a main priority.

5.3.1. Possible Improvements/Recommendations

It is advisable to create an easy interface to the database for tests with an easy setup. Setting up database connections for every test class is a time consuming operation and it does not make the test code very structured. If possible the database interface of the tested application can be used to setup database connections. A variable can be used to switch between the real and the test database.

Secondly, it can be a help to wrap or modify the SqlException object to throw more specific errors which can be caught in tests. When communicating with the database, tests often differentiate between various exceptions. To test security or error behavior, for example, tests can explicitly send faulty information to a database and it required to investigate what type of SqlException is generated.

Finally, discuss database test records with your team. For large projects it is advisable to reserve series of Ids for categories of tests. When using auto increments, reset them to the right category value to test the addition of records to the database. Try to make several sample records with special properties, such as NULL values and create records which are similar to allow choice boxes in the user interface to select options.

5.4. Components: Web service

TDD is very suitable for testing the web service's general functions. The problem with functions in web services is that they cannot be verified with executing the web service application itself. The only way is to create a client application to partly test the service. Using unit tests, we can assess the full range of inputs and outputs can be tested. TDD works very well with the web service: it is easy to think of suitable tests as a programmer. Code coverage of nearly 100% can be achieved without much effort, new technologies can be assessed and exploited while developing and changes are implemented quickly. The web service seems to be a very good environment for TDD. However, there are some problems especially concerning security testing.

The webservice has to cope with environmental variables. This can be, for example, the identification of the client by IP number or network ID. A test for setting the IP number is difficult to construct, so this can only be tested on a computer on a different network. The assessment of web services which use static variables is also problematic. Web services are programs that run continuously, keeping the values of static variables. When running a test, that causes a change to such a variable, the behavior of the program is influenced. If multiple tests are executed unexpected behavior can occur. For example, it makes checks such as “x == null” in the application fail or pass, because the “x” variable is set by a former test. The behavior of those variables is hard to track and are sometimes discovered when the test sequence is changed.

5.4.1. Web service: WSE

The security for the web services in the Microsoft .NET framework [63] is arranged by WSE 2.0, which stands for Web Services Enhancements. WSE uses SOAP as communication standard, which stands for Simple Object Access Protocol. The authentication of the user and password is done automatically by using the username and password according to the accounts created on the Windows computer. The Windows authentication is very difficult to test via the Test-Driven methodology: a Windows account cannot be created and removed on-the-fly using a test, so the authentication cannot be tested properly only using unit tests. The WSE library is a closed source library, which sets several attributes while reading a SOAP message. Most of these properties are read-only and thus can not be emulated with tests within the web service. A clear example is the usage of the UsernameToken, which contains the username sent along with the SOAP request. To test the security fully, two computers are required with different Windows NT user logins. In the case study conducted, a major security flaw was discovered when having multiple sessions at the same time. It was designed that the web service for each user opened a new database connection. This worked well with the tests and when only one user was using the system. However, when two users logged into the system the database connection was reopened and the first user could see data which was only available for the second user. This was not visible with the unit tests and was not considered when writing the test with TDD.

WSE reads the username and password from the SOAP requests and stores both in an object called "SoapContext", which is defined as a static variable. As has been noted in the previous paragraph, it is not easy to emulate this object. The solution to this problem is to put these unit tests in a different component, which calls the web service via SOAP. In this way the objects will be generated automatically by the WSE library so that it becomes possible to test the back-end and authorization functions. In this way, it is possible to apply the Test-Driven methodology on webservices. This approach has also some disadvantages. Because there are two applications now, compiling and running tests is much slower. This is strengthened by the fact that the calls need to be processed by the web server and the WSE libraries on both sides. Furthermore, when adding new functions or when refactoring, all tests are run to check on compliance. With the tests split up into two pieces, it requires more effort to test every time. When to all the tests are not executed, after implementing multiple features or refactorings, it can result in having multiple tests fail at once (also refer to paragraph 6.1).

An advantage of using TDD for security tests is that the security permissions are strict. By default, only the tested features are allowed to be executed. When a test is first executed on the client, a permission denied will be returned because the method is not in the database yet. By slowly releasing the security permissions, the function can be executed. In this way, every time a new function is created the security framework is actively retested. This makes the developer more certain about the functioning of the security mechanism.

5.4.2. Web service: AOP

By using the Aspect# library, there were some different problems concerning the Test-Driven approach. In this webservice, the Aspects are used to intercept method calls. Because this is a security measure, this code has to intercept current calls, but should also intercept calls to functions not implemented yet. To test an interception function a new function should be implemented in the webservice to check if the interception happens correctly. The test created for the Test-Driven methodology should thus contain code, which belongs to the implementation of the program. This is not desirable since these tests can influence the stability of the program. This is not always a problem. For example the Aspect# library requires the intercepted methods to be virtual to be intercepted. This means that if a method is not virtual, it is not intercepted which breaches the security. Therefore, the implementation of a check, which looks if a class contains only virtual public methods, is necessary. The test to assess this checking method can be made by using a sample class with a non-virtual method.

This sample class is indeed part of the real program, but does not have a real function. However, it still blurs the border between test and implementation.

5.4.3. Possible Improvements/Recommendations

TDD works well with testing the web service, but functional unit tests as used in TDD are not sufficient to test the security with concurrent logins of web services. It is advisable to develop/adapt a small test suite at the end of an iteration, which is deployed at multiple clients to test sessions and concurrency. This allows programmers to think of possible security threads after developing an iteration with TDD. It is easy to overlook these tests, since they are often not a requirement on the requirement list.

For security tests without focus on concurrency, copy the internal tests to a client application. The easiest way is to compile the tests into a library on a shared drive which can be accessed by both the server and the client. TDD works really well with slowly opening the web service, to allow the user "just enough" to operate the application. In case there is a different authentication method for users inside the company network and outside the network, be sure to make an user with equal rights for both networks and use a dialup to test the access to the application. It must be noted that adding such functionality nearly doubles the required testing time.

5.5. Components: Client

The logics of the client are where TDD excels, also in the client. The methods with a specific technical task are easy to test and easily adapted when there are change requests. At a certain point during the second iteration, there were comments of some users internally, that the application should be faster than it was. The design available at the time, loaded all the data from the web service when it was requested. With the accumulation of new projects, this became slower. Leaving the tests the same, a speedup, by redesigning parts of the client, could be applied. The test were used as verification that the data was still loaded correctly. There was however no case, in which a test failed on the functionality it was built to test. When tests failed - and they did fail - tests failed due to a new way of handling variables. For example, a hashtable variable was changed to store objects instead of strings, which made a test fail. The tests expected to compare strings from the hashtable with, for example, a string from the database and now failed with an invalid cast exception. In some cases the test and the implementation were not updated to cope with the new structure. In other cases the project was correctly adapted, but the tests had to be adapted too. This slowed the developing process down, but filtered a lot of simple bugs from the product. There was for example a change in the data source type, which is a property of a standard component in .NET. The object changed from handling a data set to handing a data table. With help of the tests, it was easy to find where this variable was used throughout the application. In the end this saved time because bugs were captured sooner. This kept the project on schedule. Also change requests could be received earlier, because the products which were released were incomplete but not buggy. These early releases definitely caused a major requirement shift to a Project Management system instead of a forecasting system. The project management functions got the attention of the users as an important feature and were quickly extended. Requirements were reprioritized and even total new requirements, such as adding feedback and speeding up the application were added to the requirements. The speedup had a big impact on the structure of the system, but could be implemented quickly because of the initial stage of the application. The allowed responses of an application were very hard to estimate since different response times are acceptable in different cases and for different stakeholders. Delivering a more or less bug free application allowed the users to effective testing and helped guiding the project into the direction which was requested by the customer.

5.5.1. Client: Interface

Human input and the events generated by humans are difficult to test, especially in a Test-Driven manner. This is because it is very hard to register, create and catch events by tests, which is a lot of effort just to see if an event is caught. Furthermore, results of human events can simply be seen by the human eye when running the application. Another difficulty with human input is, for example, the displaying of screens after the user has taken an action. Since this is time dependent, it is impossible to write unit tests which check whether the dialog really popped up at a certain time. This happens, for example, if an exception is generated by the webservice. The application should present this exception in a dialog to the user. However, there is no way to check if this dialog is presented after the exception, since this may be a local variable of the constructor for a form, for instance.

A proposed solution for these limitations is the use of a mock object framework such as DotNetMock. Mock objects are fake interface objects which act like real interface objects. The best way to use mock objects for the interface is to split the interface components via the Model-View-Controller [56] design pattern. In this design pattern, the interface is separated into two classes: a controller and a view class. The view class' responsibility is only to display the controls and passing through other information and events. The controller class handles the input and output of the view class. In this way the view class can be faked using a mock-up class. This fake viewer can be loaded via the constructor of the controller. The constructor of the controller accepts an interface (interfaces as in UML [57]) of the viewer. The application loads the real viewer into the controller; a test loads a fake viewer (MockObject). In this way, the controller logic can be tested extensively without having to display the interface every time. However, MockObjects are not particularly fit for changes: if a new control is added to the viewer, the UML interface and the fake viewer need to be revised. In the case study, the human interfaces were changed most often. In a test last environment, using these MockObjects would have been easier since the methodology does not allow so many changes. After several tries, the MockObjects were removed from the project. The separation of controller and view was maintained, because the division of tasks made the code much more organized.

Most of the difficulties were coming up with unit testing environmental Microsoft Windows variables. They are hard to emulate, especially in the Microsoft Windows system, which doesn't support many text based commands to create users, set variables, etc. A clear example is the detection of whether a user is logged in into a particular domain or not. In the webservice client, different decisions are made based on the domain. It is not possible to influence the domain with a test case. The only way to test this via the Test-Driven methodology is to create a test on a computer which is not member of the domain and implement the code. This conflicts with the principles of the Test-Driven methodology, because it is hard to test quickly. To increase the speed of testing on multiple computers, a remote desktop client can be used.

The unit testing in the case study was facilitated by NUnit, a member of the xUnit testing framework. A shortcoming of NUnit in testing client objects and interfaces is that when a test is placed in a class to be able to evaluate private variables, only the default constructor can be initiated. Constructors with arguments cannot be initialized and if there is no default constructor available, the test will simply not run. As solution, an init function was used to execute the tasks of the constructor. The constructors only assigned objects, for example (pre- and post conditions are omitted):

```
public AClass( int i1, string s1 )
{
     i = i1;
     s = s1;
     B = new B(i);
     …
}
```

Becomes:

```
public AClass( int i1, string s1 )
{
     i = i1;
     s = s1;
     init();
}

private void init()
{
     B = new B(i);
     …
}
```

5.5.2. Client: 3rd party components

Despite being time saving, COTS (commercial off-the-shelf) components are a problem for Test Driven Development. If they are closed source, which means that testing can be done, but fixing is not possible. Neither are change requests. Open source components are testable and fixable, but another problem arises: how to test them Test Driven? At the start of the project, it was decided that these components were not to be done with TDD. Later on, it was found to be a major flaw. The COTS control components generated bugs during the rest of the life time of the project. After a while, most were fixed but it was still suspected that the controls contained bugs which would emerge later in the project. With the components which were extensively tested, this fear was a lot lower.

5.5.3. Possible Improvements/Recommendations

A major improvement for testing applications which communicate directly with end users is to create tests by using Aspect Oriented Programming. This will be explained in section 5.6. Testing with Aspects can make it possible to use information from the user interface with much more ease.

Because of current limitations with NUnit to create objects with overloaded constructors, it is helpful to use init operations in objects, which execute the basic object initialization operations. The parameters can still be supplied to the object by passing them via the constructor, but the object initialization is done by the init function. This is already common with user interface objects in .NET, since the Initialize() method is automatically filled with actions to initialize the user interface. The test initialization operation now only has to set the variables which are assigned in the constructor of the object and then call the init operation.

The use of MockObjects is generally not recommended, because it slows down making changes to interface objects. The interface objects in the case study where most liable to changes. Developing with MockObjects would require almost twice the changes and, if the controller-view separation is used, not so many advantages.

The last recommendation is to implement unit tests for open source components or they can become the major source of bugs of your product. The best way to avoid having to invest a lot of time into creating a full test set is to create test only for the functions which are used. Finding all the methods which are called when a function is invoked requires a stack trace analysis. This does not result in a guarantee that all executed code is covered by a unit test, but tries to cover most common execution paths.

5.6. Testing Interface Components with Aspects

The test executes one or more methods from the application and after that it checks the results. These results are mostly hard coded in the test. Especially for database and interface operations, some of the results of the test are harder to get. Another problem occurs when you want to test whether a piece of information is set in the interface. This requires evaluating a component in the interface, which is supposed to have a private modifier. There are two ways to solve this. Either change the visibility of the component into public (which requires the change of real code for a test) or find another way to test the value (which is hard and sometimes not even possible).

Finally, there is a problem with checking for events occurring *during* the tests. It is for example not possible to test if a message is displayed during a series of events and later changed back in the same method. There is no way to recognize if this event occurred, since the action was executed in the middle of a single execution flow.

Summed up the following limitations are visible:

Disadvantages and limitations (using NUnit)

- Cannot evaluate variables or events *during* tests,
- Cannot evaluate private variables. Making them public would require the program to be changed for tests. This is not an optimal solution.

Using the Model-View-Controller design pattern with mock objects can remove some of the limitations of user interface testing, but not all. For example, the test classes still need to have access to various variables, like error messages. This means mock objects can not eliminate the need of making attributes public.

But in my opinion, the largest problem with mock objects is that they take a lot of time to develop. The Test Driven methodology is set up to have quick results and make quick changes. With mock objects, there are two objects in the project doing the same thing. A change in the view requires a change in the mockup and visa versa. This opposes Kent Beck's vision of the Test Driven methodology, "It feels good to write it" and of XP, "Embrace Change". Having to change two classes for one change is not the most motivating work and requires more effort than just changing one class. Thus, it is less likely that developers will change interface objects easily. While interface objects are most likely to be changed since they interact with users.

Enabling tests to evaluate variables while executing methods, can improve testing of private variables and events during methods. A first step into this direction can be to add a functionality which allows testing before the exiting of a method, allowing access to all existing private variables. In fact, we can say that we no more 'Unit test' software, but test even smaller pieces of software code.

As we look at language tools [58], we can see that there is already a paradigm available which intercepts methods before they exit: Aspects.

An example which displays the limitations of unit testing:

There is a form with a list of projects. When the user clicks on a project, an event is generated with the selected project number. The current status bar text is changed to "Loading project". The form contains other controls which need to be updated if a new project is selected. The project number is used to gather new information from the database and display it. Finally, the status bar text is changed again to "Ready". A unit test should be able to test this situation by comparing the displayed project information with the database.

Using a testing framework such as NUnit, the event in the example can be generated since it is public. However, it is not possible to check the displayed data. Also, it cannot be checked if the status bar has changed, since the unit test can only assert the status bar text before and after the event. With help of aspects it is possible to test this using the following scheme:

- First the test will be created throwing only the event, since other results cannot be verified due to access levels,
- To make the test compile the event is implemented,
- The program is run and a green bar is displayed,
- Now a "test aspect" is created for the status bar event handler. It must be made sure that the aspect is only invoked for this particular test (this can be done manually or via tool support). The aspect tests the first message from the status bar,
- The status bar function is implemented so that a green bar is received,
- The test is adapted to verify the status bar message for the second time the function is run,
- Test is run and red bar is displayed,
- Status bar event handler is adapted so that a green bar is again received,
- Everything is refactored,
- Etc...

With help of the Aspects it is possible assert variables after a method is called. This is done by marking the aspect as *"privileged"*. We could also put assertions directly in the methods. But as can be seen at the status bar example, the values are dependent on the test which is run. The big advantage of using aspects as test is that aspects are meant to be weaved into the program, grouping different function types of the program together. One of the function types of a program can be testing. With proper tool support it is possible to activate the test-aspects *only* when a particular test is run with ease. It can be imagined that a testing framework could be developed which only executes test-aspects. By using aspects tests are able to evaluate private variables and evaluate events during program execution without compromising the structure of the program.

Ultimately a testing framework can be developed where pre conditions and post conditions are specified for each object operation. The testing framework generates aspects from these pre and post conditions and executes the method with these conditions, verifying if the operation functions correctly. Consider the fictive operation:

```
// Pre: Name should be the name of an existing account
// Post: the Account bound to Name is returned, the number of account re-
quests is increased by one
Account getAccount(string Name)
{
     accountRequests++;
     Return accounts[Name];
}
```

By means of an Aspect which is invoked before the execution of this operation, the accounts variable can be initialized when required. An Aspect which is invoked after the execution of the operation can check on the post conditions: if the accountRequests variable is set to the correct value and if the correct account is returned. This would be a help in maintaining the test set and checking the internal variables.

5.7. Evaluation of the Claims Made

After the discussion of the performance of TDD during the life time of a project and concerning the various components of the case study, the claims made in section 3.3 have to be evaluated. The claims are handled individually and evaluated as follows:

1. It is possible to apply TDD in a software project anywhere were requirements are implemented. With minor adaptations TDD will be suitable for testing databases, interfaces and other technologies.

   This is in not true in general. There are definitely problems with applying TDD everywhere in a project. Some solutions presented in this thesis, such as aspect oriented testing, increase the applicability of TDD, but do not totally diminish the problem.

2. TDD handles changes in requirements better than predictive approaches.

   Accurate claims are difficult to make here, but the case study development process was lead by requirement changes and what was supposed to be a forecasting application, evolved into a project management application. Performance issues could be solved early in the development process and are unlikely to be found early in a predictive approach. In the process, it was noticed that the stakeholders did not have a concrete idea of what should be in the end product and what should not be. New requirements were made while the project evolved. A good example is the feedback plug-in for Microsoft Outlook. In the initial phase of the application, feedback was not even on the requirements list. In the second iteration, after working with the first release, a feedback panel was added as high priority requirement. Later, this lead to the idea of adding emails to a project and thus creating a plug-in for Microsoft Outlook. When several testers believed the application had sufficient features to move from their Excel sheets towards the application, another new requirement was mentioned: an action list panel.

As can be seen on the flow of events, it would be unlikely to receive all these requirements at the beginning. One of the problems with this is that it is very hard to make people see the potential of an undeveloped product. After the first and second iteration, the requirements of the end users got more and more specific and changed several times. It cannot be denied implementing changes is costly; however it saved a lot of time implementing changes after the delivery of the application. With TDD the design of the application is continuously optimized. When changes have to be applied after the development of the system is completed, they are easily 'hacked' into the design. Eventually, this can result in problems with maintaining the application.

3. Developers will be able to look further with TDD, thinking of better solutions while programming.

   Developers are able to look further with TDD, but only for low level programming. The main technologies to be used in the case study were decided in the period from before the implementation until the first weeks of the implementation. Also, the general design was determined in this period. During the implementation, several alternations were made to improve the design or to implement discovered technologies. An advantage of TDD is that programmers will know sooner if their approach is suitable for creating a product which satisfies the end user. By using a pilot the suitability of a technology for a particular situation can be evaluated without consuming a lot of time. This was for example the case with the web service's performance.

4. The XP/TDD process is more enjoyable for the stakeholders.

TDD has advantages when seen from a programmer's perspective: quickly starting with building a real program is for many a relief and not having to write a technical documentation is considered positive because a programmer can do what he likes to do: developing an application. However, making simple tests and having to change those tests can slow down the process. Especially when simple changes make many tests fail, it makes programmers reluctant to create more tests. This would mean a failure in the usage of XP/TDD. The key to solve this issue is to create unit tests with Aspects as explained in Section 5.6. For some features, even testing with technologies such as AOP is not satisfactory. In this case, black box testing techniques as described in [31] should be used to assess these features.

For new team members the lack of a written technical document was not an obstacle. The tests provided necessary documentation and all architectural information was supplied by using small paper documents and discussions. Not having a large written documentation was not at all a hindrance.

From a customer's perspective: For a customer, the TDD approach is ideal, according to various interviews held among customers. Due to the incremental delivery, a total vision about what has to be created does not have to be done upfront. Customers can get used to a new system, ask for migration tools and propose new requirements. Pricing models which requires payments for each change should be avoided, since this decreases the flexibility and does not require the developer to deliver high quality software. Instead, payment can be based on usage (e.g. by using licenses) is a better way. The software developer should be financially stimulated to deliver a product of high quality. During the case study Test Cases were specified to show how a particular functionality is verified. These Test Cases help to verify the correctness of the understanding which the developers have on a certain functionality.

5. Except for delivering a better suited product, supporters of TDD say that the quality of the code will also improve with TDD. Due to creating test first, a nearly 100% code coverage of tests will be attained automatically.

   When we compare the code written in TDD and the open source COTS objects, this is definitely the case. However, these COTS objects might not represent the typical stereotype of an object. What we can say is that there was a working product after each iteration, which was not full with bugs. The product can also be shaped according to the customer's wishes, which means that programming and design time is invested in the requirements most needed by the customer.

To get an estimate of the quality of the code, the final libraries are examined by a code analysis application, called NDepend [59]. NDepend measures several metrics at assembly level and at type level. The metrics at assembly level can indicate how well the design of a project is in terms of maintainability. The type level metrics evaluate the program at a lower level and can detect possible complex classes, which can cause trouble. If we compare the results with NUnit, another project developed using TDD, results are similar. Compared to the CrystalDecisions library, which is part of the .NET framework, there are noticeable differences, especially for the cyclomatic complexity. By creating a control graph, cyclomatic complexity is an indicator for the complexity of a module [52]. In NUnit and the case study the highest complexity in a module is around 90, in the CrystalDecisions module this is more than 650. It must be noted that cyclomatic complexity is just an indication of complexity; it is arguable whether cyclomatic complexity is an accurate complexity measure [64].

Finally the code coverage of the test in both the client and the web service was 93.5%, which is close to the goal of TDD of 100% code coverage. I have to note that these figures can only count as an indication, not as evidence of code quality: a deeper investigation with more samples is required to be able to speak of evidence.

5.8. Summary

It can be summarized from the evaluation of the claims that the use of TDD and XP for developing applications has many advantages. Despite the problems caused by test dependencies, the construction of tests during the development process remained steady. The main problem with creating unit tests which is discovered in the evaluation is: how to make tests for everything you want to test? Sometimes the creation of tests is to time consuming to check if a function works compared to manual verification. This was in particularly the case with testing the database and the user interface. In other cases, it was technically very difficult or even impossible to create a suitable test for a given situation. This is the case with the user interface, build-in features and special technologies, such as Aspect Oriented Programming. Aspect Oriented testing as described in Section 5.6 can provide a solution for some of these problems. However, experience has to be gained with deciding whether to construct a test or to verify a requirement without using a unit test.

The biggest advantage of TDD is that a product of high quality can be developed by maintaining flexibility. The case study provided various indications of a high quality of the source code. By creating tests first, the design is refactored continuously disallowing changes to have a bad influence on the system design. This results in source code which is easier to maintain. By using iterations and active customer participation, problems can be identified quicker which saves having to rewrite various pieces of code. All the previous statements help the programmer to have a smooth development process, which adds to programmer's satisfaction. Work satisfaction can be an important ingredient for success. Customer participation also gives the customers the feeling they can shape the application to their needs and give a better understanding of the problems the programmers are facing and the other way around.

6. Conclusion

In this thesis a case study to evaluate TDD and its software development approach, XP, has been done. The biggest advantage of TDD in combination with XP is the flexibility of the development process. Especially when the exact user requirements or the user domain are not clear, development with TDD is very suitable and delivers stable applications. When we compare the software development approaches, adaptive and predictive approaches, in the end the adaptive approaches seem to deliver a more predictable project planning. This is because the chance that the product has to be revised is smaller.

During the lifetime of the case study, the major problem with TDD and unit tests in general is the maintenance costs due to coupling of IUTs. As explained in section 5.1, the maintenance costs of unit tests will increase as requirement changes are made to the project. A test plan can provide support in keeping this maintenance cost low. At this moment there is no known solution for this problem, because the consequences of changing IUTs on unit tests can not be measured with the tools currently available.

TDD can be used to test most components, but there are still problems with testing user interfaces, layered architectures and database constrains. The complexity of the database constrains is mostly low and predictable so this does not cause major problems. Concerning the problem with unit testing layered architecture care must be taken in setting up a suitable testing environment. By setting up a test database with several test entities and providing a mechanism to easily switch between test and production database, this can be solved. The problem with testing user interfaces is more complex, due to insufficient tool support. This thesis proposed a possible solution for testing interfaces by using Aspect Oriented Programming.

The tool support for developing with TDD is increasing. The xUnit testing framework has been ported to support several programming languages such as Java, C# and C++. For C#, other frameworks have been developed such as TestDriven.NET, which make executing unit tests easier.

The response of the end users and project members on TDD was very good. End users were excited, because they had the chance to influence the direction of the project and see the most important requests working in the next version.

Because of the flexible process bugs could be found and solved quickly. In general TDD caused end user more actively involved in the development of the system. For team members, TDD delivered organized and readable source code, which saved time when new team members were added to the project.

To summarize, TDD delivers a good quality products which is adapted on end user input. By using iterations the stability of the product is monitored along the development process. There are various problems with TDD but these problems do not get in the way of TDD being a good development practice.

6.1. Final Thoughts and Future Work

This section will cover some thoughts based on a personal vision which is developed during the execution of the case study. These thoughts can be a topic for further research and are not covered by hard evidence in this study. The main thought of TDD, using implemented test to specify has its advantages, but requires discipline and skills to do well. Finding ways to develop good tests which can sustain the most common changes in an application calls for practice. It can mean that TDD fails easily when a team uses TDD for the first time on a project which is under a lot of pressure. For projects under pressure, another question is if programmers can take the time to do refactoring well. Since refactoring does not add any new functionality to a program, this seems doubtful. In the case study there was enough time to do refactoring well. Even if there is enough time, it can be difficult because a natural drive of the programmers to create new functionality.

The incremental development by delivering full products is a good idea in general, because it lowers the risk of developing a product which is not right for the customer. It also allows early detection of bugs and problems so that the total cost of build can be estimated. For products with a very small user interface and complex tasks, such as systems for space ships, using TDD is less suitable. This is because the system is not likely to endure many customer change requests and it is more important to agree on an interface that works, over picking one which is suitable for the end user. Projects for space ships seem to be very large too, which is not the ideal territory for XP/TDD. In very large projects it seems hard to develop project with XP/TDD, because of the difficulties with internal communication. It might be possible to adapt and extend XP/TDD to make it more suitable for these projects, but at the current state, XP/TDD is better suitable for smaller systems with a high interaction with a customer.

When outsourcing IT activities, working with TDD can bring some extra problems. Arranging suitable financial service level agreements can be difficult: companies often desire fixed price because outsourcing is often used to prevent going over budget. When using TDD it can be hard to get a grip on the change requests. It is difficult to not get a situation where requirements are change continuously. On the other hand, if well monitored, TDD can lead to superior code quality and products made with TDD are better suited for end customers. This is a reason to try to overcome the difficulties when applying TDD with an outsourcing partner.

Bibliography

[1] P. Taylor, "Designerly thinking: what software methodology can learn from design theory", In *Proceedings of the International Conference on Software Methods and Tools.*, Wollongong, 107-118, 2000.

[2] B. Littlewood, L. Strigini., "Software Reliability and Dependability: a Roadmap", In *Proceedings of the Conference on The Future of Software Engineering*, pp 175-188, May 2000

[3] The Standish Group International, "The CHAOS Report", *http://www.standishgroup.com,* 1994

[4] Paulson, L.D., "Adapting methodologies for doing software right," *IT Professional* , vol.3, no.4pp.13-15, Jul/Aug 2001

[5] K. Beck, *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[6] Maximilien, E.M.; Williams, L., "Assessing test-driven development at IBM," *Software Engineering, 2003. Proceedings. 25th International Conference on* , vol., no.pp. 564- 569, 3-10 May 2003

[7] Geras, A.; Smith, M.; Miller, J., "A prototype empirical evaluation of test driven development," *Software Metrics, 2004. Proceedings. 10th International Symposium on* , vol., no.pp. 405- 416, 14-16 Sept. 2004

[8] George, B.; Williams, L., "A Structured Experiment of Test-Driven Development", *Information and Software Technology*, vol. 46, no. 5, pp. 337-342, 2004.

[9] Rigby, K., *Technical Management: Documentation Standard,* Rigby Publishing Ltd, 2002

[10] Hughes, B.; Cotterell, M., *Software Project Management,* third edition, McGraw-Hill Education, 2002.

[11] Berard, E., "*Fisuys on Object-Oriented Softwure Engineering*", Prentice Hall, 1993.

[12] "IEEE standard glossary of software engineering terminology," *IEEE Std 610.12-1990* , vol., no.pp.-, 10 Dec 1990

[13] Trussell, L., "Essential software development methodology," *Power Engineering Society 1999 Winter Meeting, IEEE* , vol.1, no.pp.357-361 vol.1, 31 Jan-4 Feb 1999

[14] Fairley, R.E.; Willshire, M.J., "Iterative rework: the good, the bad, and the ugly," *Computer* , vol.38, no.9pp. 34- 41, Sept. 2005

[15] Highsmith, J.; Cockburn, A., "Agile software development: the business of innovation," *Computer* , vol.34, no.9pp.120-127, Sep 2001

[16] Royce, W., "Managing the development of large software systems: concepts and techniques, " In *Proceedings of ICSE 9,* 1970.

[17] Boehm, B. W., "A spiral model of software development and enhancement," *Computer* , vol.21, no.5pp.61-72, May 1988

[18] "Lecture notes Embedded Real-time systems", *TU Delft Press*, TU Delft, 2004

[19] Charette R., *Software Engineering Risk Analysis and Management*, McGraw-Hill, 1989

[20] van Genuchten, M., "Why is software late? An empirical study of reasons for delay in software development," *Software Engineering, IEEE Transactions on* , vol.17, no.6pp.582-590, Jun 1991

[21] Ming Huo; Verner, J.; Liming Zhu; Babar, M.A., "Software quality and agile methods," *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International* , vol., no.pp. 520- 525 vol.1, 28-30 Sept. 2004

[22] Sommerville, I., *Software engineering*, 6th ed., Addison-Wesley, 2000.

[23] Fruhling, A.; Tyser, K.; de Vreede, G.-J., "Experiences with Extreme Programming in Telehealth: Developing and Implementing a Biosecurity Health Care Application," *System Sciences, 2005. HICSS '05. Proceedings of the 38th Annual Hawaii International Conference on* , vol., no.pp. 151b-151b, 03-06 Jan. 2005

[24] Cockburn, A.; Highsmith, J., "Agile software development, the people factor," *Computer* , vol.34, no.11pp.131-133, Nov 2001

[25] Fowler, M.; Highsmith, J., "The Agile Manifesto, " *Software Development*, pp. 28-32., Aug. 2001.

[26] Ambler, S.W., "Agile model driven development is good enough," *Software, IEEE* , vol.20, no.5pp. 71- 73, Sept.-Oct. 2003

[27] Ambler, S.W., *Agile Modeling: Effective Practices for Extreme Programming and the Unified Process,* Wiley, 2002.

[28] Martin R.C., *Agile Process: Agile Software Development, Principles, Patterns, and Practices*, Prentice Hall, 2002.

[29] Beck, K. and Cunningham, W., "A laboratory for teaching object oriented thinking," In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, no.pp. 1- 6, 1989.

[30] "IEEE standard for software unit testing," *ANSI/IEEE Std 1008-1987* , vol., no.pp.-, 29 Dec 1986

[31] Binder, R., *Testing object-Oriented Systems*, Addison-Wesley, 1999.

[32] Beck, K. *Test-Driven Development*, Addison-Wesley, 2004.

[33] Janzen, D.; Saiedian, H., "Test-driven development concepts, taxonomy, and future direction," *Computer* , vol.38, no.9pp. 43- 50, Sept. 2005

[34] Geras, A.; Smith, M.; Miller, J., "A prototype empirical evaluation of test driven development," *Software Metrics, 2004. Proceedings. 10th International Symposium on* , vol., no.pp. 405- 416, 14-16 Sept. 2004

[35] Chaplin, D., "Test first programming", *TechZone,* 2001.

[36] Jeffries, R.; Anderson, A.; Hendrickson, C., *Extreme Programming Installed,* Addison-Wesley, 2000.

[37] Mugridge, R., "Test driven development and the scientific method," *Agile Development Conference, 2003. ADC 2003. Proceedings of the* , vol., no.pp. 47- 52, 25-28 June 2003

[38] Pancur, M.; Ciglaric, M.; Trampus, M.; Vidmar, T., "Towards empirical evaluation of test-driven development in a university environment," *EUROCON 2003. Computer as a Tool. The IEEE Region 8*, vol.2, no.pp. 83- 86 vol.2, 22-24 Sept. 2003

[39] van Deursen, A., "Program comprehension risks and opportunities in extreme programming," *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on* , vol., no.pp.176-185, 2001

[40] George, B., Williams, L.A.; "A structured experiment of test-driven development, " *Information & Software Technology*, vol. 46, no.pp. 337—342, 2004.

[41] Constantine, L., Methodological Agility, *Software Development*, vol., pp. 67-69, June 2001

[42] Bass, L.; Clements, P.; Kazman, R., *Software Architecture in Practice*, 2nd ed., 2003.

[43] Meyer, B., ".NET is coming [Microsoft Web services platform]," *Computer* , vol.34, no.8pp.92-97, Aug 2001

[44] Newkirk, J.W.; Vorontsov, A.A.; *Test-Driven Development in Microsoft .NET*, Microsoft Press, 2004.

[45] Stott, W.; Newkirk, J.W., Improve the Design and Flexibility of Your Project with Extreme Programming Techniques , *MSDN Magazine*, April 2004

[46] Moise, D.L.; Wong, K.; Sun, D., "Integrating a reverse engineering tool with Microsoft Visual Studio .NET," *Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings. Eighth European Conference on* , vol., no.pp. 85- 92, 24-26 March 2004

[47] Astels, D., *Test Driven Development: A Practical Guide*, Prentice Hall, 2003.

[48] *W3C Glossary: Web Service, http://www.w3.org/TR/ws-gloss/*

[49] van der Lubbe, J.C.A., Basismethoden Cryptografie, *Delft University Press*, 1994.

[50] Xie, T.; Marinov D.; Schulte W.; Notkin, D., Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems*, 2005.

[51] Michaelis, M., A Unit Testing Walkthrough with Visual Studio Team Test, http://msdn.microsoft.com/library/en-us/dnvs05/html/vstsunittesting.asp, 2005.

[52] McCabe, T.J., A Complexity Measure, *IEEE Transactions on Software Engineering*, vol.2, no.4, pp.308-320, December 1976

[53] Stelzer, D.; Mellis, W., Success Factors of Organizational Change in Software Process Improvement, *Software Process Improvement and Practice*, vol.4, no.4, 1999.

[54] McCaffrey, J., Test Run, *MSDN Magazine*, March 2005.

[55] Elrad, T.; Filman, R. E.; Bader, A., Aspect-oriented programming: Introduction. *Communications of the ACM*, vol.44-10, no.pp.29-32, 2001.

[56] Gamma, E., Helm, R., Johnson, R., Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[57] Rumbaugh, J.; Jacobson, I.; Booch, G., *The Unified Modeling Language Reference Manual.* Addison-Wesley, 1999.

[58] Pronk, C., *Objectgeorienteerde Talen, Technieken en Architecturen*, TU Delft, 2002

[59] NDepend .NET code analyzer, *www.ndepend.com*

[60] Boehm, B.; Turner, R., "Using risk to balance agile and plan-driven methods," *Computer* , vol.36, no.6pp. 57- 66, June 2003

[61] Coram, M.; Bohner, S., "The impact of agile methods on software project management," *Engineering of Computer-Based Systems, 2005. ECBS '05. 12th IEEE International Conference and Workshops on the* , vol., no.pp. 363- 370, 4-7 April 2005

[62] Perrochon, L.; Mann, W., "Inferred designs," *Software, IEEE* , vol.16, no.5pp.46-51, Sep/Oct 1999

[63] What is .NET, *http://www.microsoft.com/net/basics.mspx*

[64] Shepperd, M.; A critique of cyclomatic complexity as a software metric, *Software Engineering Journal*, vol.3, no.2,pp.30-36, 1988.

A Forecasting web application in .NET

A.1. Project vision and scope

In a satisfactory report held under the customers from EPCOS, it appeared that average delivery times for products were too high. The aimed delivery time, was not achieved in all cases. To improve customer satisfactory, this percentage of deliveries on time should be increased. To increase the process customer satisfactory, EPCOS has to improve its supply chain management by lowering the product delivery times. The current environment in EPCOS does not provide the reports necessary to help marketing prepare a forecast in an intelligent way. Nor does it provide a tool to consolidate and simulate customer forecasts. There is no easy way to gather information currently with SAP, which is the ordering processing application used in EPCOS.

Forecasting application will be used to budget and forecast orders. At the moment, forecasting is done on a yearly basis, which makes it inaccurate. The goal of the forecasting application is to increase forecasting accuracy by increasing the frequency with which representatives update the forecasts. Another goal of the forecasting application is to organize all the forecast information to enable to track forecasts and to provide easier access. After this is done, the marketing department of EPCOS can make decisions on the number of parts to be ordered at the EPCOS factories. The factories should be able to view and confirm the forecasts placed by the marketing department.

By forecasting in advance, production can already start before a real order has been placed. This lowers the delivery time. Currently, production starts after an order is received or when a sales representative sends an email about an expected order. This communication is unstructured and often important information is not communicated to the sales division. An important improvement is to make information more transparent inside the company. This should be done by standardizing communications processes between sales representatives, sales headquarters and production facilities. Sales representatives can indicate how many orders will come in the near future. The plan is to make an application where representatives can input forecasting data after visiting a customer. The marketing department can anticipate on this data and instruct the production facilities to start production, before the real order is placed. When a customer finally places the order, production is already running or finished.

Forecasts have to be assigned to projects with several properties, adding another requirement. The application should also be able to create new projects, maintain running projects and save the history of project changes. Projects have to be filtered on project properties such as customer name. During the development, several other project managing requirements where added, hence the application was named "EPCOS Project tracking and Forecasting (EPF)". This name will be used in the rest of this thesis to identify the application.

A.2. System Architecture

The system to be developed consists of three layers: a client layer, a web interface layer and a data services layer. In software architecture this model is commonly known as a three-tier architecture. A tier is a partitioning of functionality and can be deployed at distinct psychical machines [42]. The layers in the system are implemented in the EPF application as follows:

- A client application, which interfaces the end user and executes operations concerning project tracking and forecasting
- A web service, interfacing the database for the client application and providing authentication features
- The database, providing necessary storage

To allow input from the various participating parties inside and outside the local EPCOS LAN, the web interface layer is deployed on two web servers. Both web services have access to the database server running Microsoft SQL server. For internal authentication, the username and passwords are verified with the active directory server. In this case internal users are able to login with their company NT credentials. Credentials for external representatives, who are not added to the Windows NT users, are created on the database server. The system architecture deployment can be depicted as in figure 12.

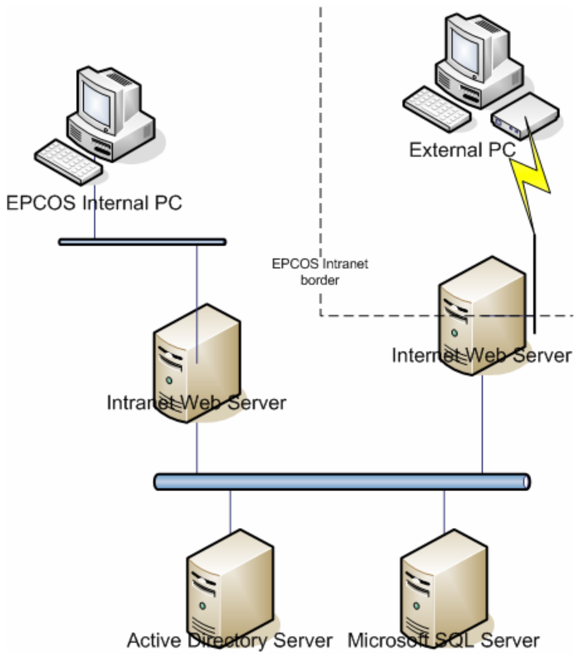

Fig. 12. Overall architecture

A.3. Used technologies

.NET Development

TDD was first used by developers which programmed mainly in Java and Smalltalk platforms [32]. In this thesis the performance of TDD will be analyzed and evaluated by using a project which is programmed in the .NET environment. .NET is an open-language web development platform released for the Windows operating system by Microsoft in 2002. The study concerns the development of a system using web based technologies and web services which are key features of the .NET framework [43]. There is an increasing interest of using TDD in .NET applications [44][45] since the release of a testing framework. A testing framework is one of the requirements for the use of Test Driven Development in a software project.

Released in 2002, .NET is Microsoft's response on the increasing use of Internet in businesses. It is "An open language platform for enterprise and Web development." [43]. Microsoft .NET is composed of two components: the .NET framework and the Visual Studio development environment [46]. The Visual Studio IDE is capable of drawing user interfaces and handling code. Visual Studio provides the ability to create applications for Microsoft Windows and has a focus on applications using the web via web services. Software applications can be coded in multiple languages such as C# (Microsoft's successor of C++), C++, J# (Java style programming language) and Visual Basic.NET. With the introduction of ASP.NET the border between predictive and web development is virtually removed. In .NET a web page is considered as a real program having an html description for the interface. The rest is programmed in one of the languages available for the framework.

The .NET framework allows components of an application to be programmed in multiple languages. In contradiction with Sun' Java, the applications developed in the .NET framework are not interpreted, but are compiled into machine code. The source code of the various programming languages is translated into Microsoft Intermediate Language (MSIL). MSIL is then compiled into machine code by a Just In Time compiler (JIT) [46]. Although the MSIL is platform independent the JIT limits the portability of the .NET framework. To make programs in .NET more portable, there are projects writing a .NET implementation for other operating systems. One of them is Mono, which is an implementation of the .NET framework for Linux. However, there is no support from Microsoft for these kind projects. One of the reasons of the choice to use the .NET framework is because the framework quickly getting adopted in industry [46].

TDD is primarily raised by Java programmers with the use of JUnit, one of the first automated unit test tools available. After the release of NUnit and the publishing of books [47], TDD is now slowly getting adopted in the .NET development community. This recent development is another reason for picking the .NET framework as basis for the case study.

Webservices

There are many different definitions of a webservice. This is caused mainly because a lot of different companies are currently involved in developing webservices. Webservices depend on the web and as the Internet hype only took place a few years ago, webservices are also a relatively new phenomenon. The W3C (World Wide Web Consortium) is an organization, which tries to standardize new web-based technologies. The W3C defines a webservice as follows:

"A Webservice is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Webservice in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards." [48]

Webservices are software applications that communicate by using a common language. This communication is done over the Internet to other distant applications using an Internet protocol, (for example HTTP). A webservice offers a certain set of functions (or methods) that all applications, which connect to it, can use. For this client-to-web-service communication several techniques have been developed. We have seen the most important older communication methods in the previous chapter. They can all be used for the purpose of webservice communication. Recently new XML based techniques have been developed. One of the major advantages of the XML based communication is the independence of language and platform. Secondly, the data is separated from the lay-out.

There are three parties taking part in webservice communication. They all offer different functionality to the client who is requesting a service from the webservice. The webservice architecture is given in figure 13. The service provider is the webservice that is willingly to offer its service to clients. This webservice is an implementation of the metadata describing the service. These metadata are published in the service registry, which is a webservice itself. The client can now find the webservice he is looking for by asking the service registry. Binding is the process that allows an application to connect to a webservice at a particular web location and start interacting with it. As suggested by the double arrow in figure 13, the communication can be both application as well as webservice initiated.

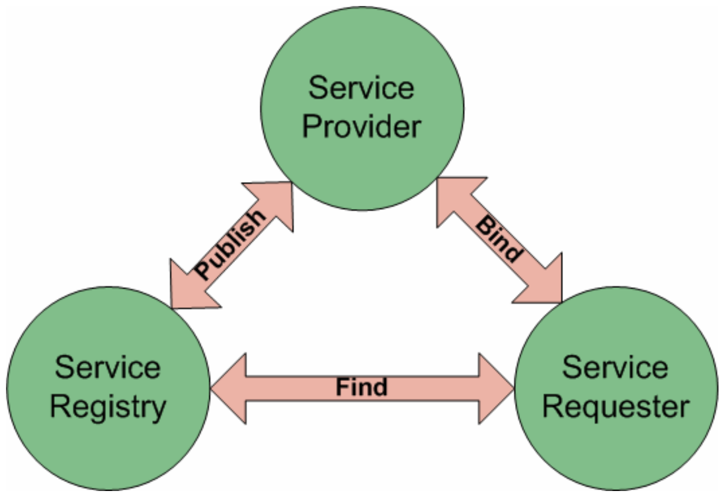

Fig. 13. Web service structure

For the three communication processes three protocols based on XML have been developed. These protocols are SOAP (Simple Object Access Protocol), WSDL (Webservice Description Language) and UDDI (Universal Discovery, Description and Integration).

A.4. Application Layers

A.4.1. The database

The database is a Microsoft SQL database which contains project information and authentication information for the web service. Table operations such as SELECT, INSERT, UPDATE and REMOVE commands are restricted per user. In this way the Microsoft SQL server provides a first level of security. The main users are categorized in four user groups: “Sales”, “Marketing”, “Factory” and “Administrators”. There is one special user called “EPFAuthentication” which is used by the web service to verify user groups and passwords. In Microsoft SQL user permission groups are called “User Roles”.

Access to the functions of the web service are also stored in the database. As for the Microsoft SQL access, groups are created. The tables responsible for containing user information are:

- `UserData` (containing the username and password)
- `User_Group` (linking users to access groups)
- `GroupData` (containing group names and descriptions)
- `Group_Right` (linking groups to web service function access)
- `Project_Member` (linking projects to users, except for administrators users only have access to the project they are member of)

A.4.2. The web service

The web service has two main functions: providing data to clients and authorizing users in- and outside the EPCOS network. For providing data from the database a database manager sets up a connection via a database manager (CDatabaseManager). The database manager read the database server variables from the configuration file (web.config) and provides several functions to execute select, insert, update and remove commands.

For external users, the database manager can login via SQL username and passwords. These users need to be created in the SQL server and the roles should be assigned. For internal users, which are created in the Windows Active Directory (AD), the web service is impersonated into the user. Using the credentials of the Active Directory user the connection with the database is made. To avoid problems with local file permissions on the web server the impersonalisation is reversed after the connection with the database is established. Because the connection with the database is already established, the database permission trust is kept. The web service stores the database manager in the application object which is reloaded when processing new requests by using the username. This scheme achieves higher performance because the database connection is kept alive for each user until the user session expires.

Data is gathered from the database by using C# datasets. These dataset objects are filled by a data adapter directly from the Microsoft SQL server. Datasets can be serialized as XML object which has the advantage that they can be send via the web service without any extra conversion. Database requests itself are protected by a mutex. This is because C# data adapters cannot fill multiple datasets simultaneously. Since multiple users can request data at the same time the use of synchronized functions is required.

Password verification

There are two types of users to be distinguished in the environment: internal (EPCOS) users and external users. Because external users connect to the web service via the internet, the password itself is not sent to the web service. Instead, a signature is attached to the SOAP message on which the password can be verified. This is a so-called proof of possession method by means of one way functions [49]: the signature is a hash from the username, password and timestamp of the message. The web service's Authentication Database Manager has a fixed connection into the database. The Authentication Database Manager only has access to the UserData table and can only get a password of one user: the user which is currently requesting a service. The webservice retrieves the password and a signature hash is calculated and compared with the one attached to the SOAP message. If the two signatures are equal the web service knows that the correct password was used by the client to create the signature and permission is granted. When used on a SSL secured line, this scheme provides protection against connection sniffing.

When it comes to internal users the password is sent in plain text, since the username and password can only be verified at the Active Directory server by using a plain text username and password. An advantage of the internal user verification is that it doesn't require new users to be created in the SQL server and the passwords are not stored in the database. The disadvantage, sending plain passwords over the network, is not a risk because internal users only log into the web service from the EPCOS network or over a highly secured line such as a VPN connection. In both cases the risk of a sniffing attack is minimal. To prevent external users trying to use the internal verification the web service checks the IP of the user to make sure it is on the EPCOS internal domain.

Security structure

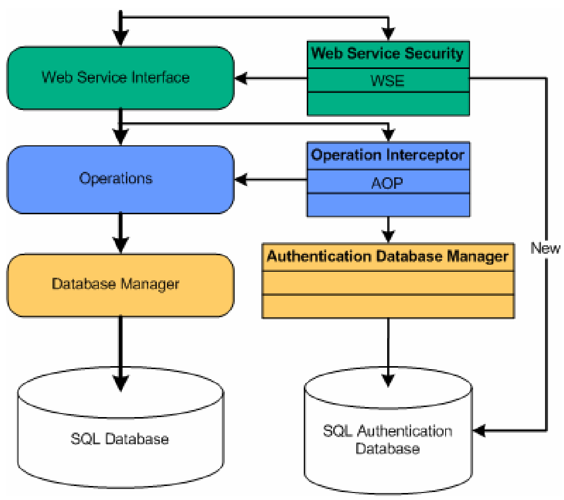

Fig. 14. Web service structure

The web service provides two layers of security (figure 14): the Authentication Manager (CAuthenticationManager, Web Service Security layer) and the Security Manager (CSecurityManager, Operation Interceptor layer). The first to be invoked is the Authentication Manager which uses the authentication methods provided by Web Services Enhancements. For each request to the web service a username and password are sent along. Before the password is checked, the Authentication Manager checks if the user has access to the requested function of the web service. This is done by acquiring the group of the user with the username and checking if the group has access to the function. When the user has access to the requested function, the password is checked by the Authentication Manager and the Authentication Database Manager.

After the password verification is finished, the connection with the database is established using the Database Manager using the user credentials and the function is executed. But not before another security level is invoked: the Aspect# Security Manager. The second layer of security is implemented by an extension of the .NET framework called Aspect#. Aspect# is an implementation of an Aspect Oriented Programming Framework which is originally developed for Java. Aspect Oriented Programming (AOP) tries to group functionality which is normally scattered over a program. A prime example is the use of logging in an application. When a program uses logging during its execution, the functionality has to be implemented in each method which logs events. With AOP this functionality is grouped by intercepting the method calls before and after its execution. In this way the logging is done at a central place: the aspect.

In the web service AOP is used to provide another level of security. Before each function in the web service is invoked, the security clearance is checked by the aspect. If granted, the method is allowed to execute, if not a Security Exception is thrown (CSecurityException). In the .NET framework a class has to be wrapped by the AOP implementation to make it intercept method calls. At the startup of the web service, the functional classes are wrapped in the constructor of the main web service file (ForecastingWebService.asmx). Since the Aspect# implementation only intercepts 'virtual' methods, the Security Manager checks first if all public methods are virtual before it wraps the class to ensure the interception of all methods. The aspect is relatively simple: it states that all methods from the name space which contains the functions have to be intercepted.

When the aspect is invoked, it loads the group of the user from the database and checks the rights, just as the WSE authentication method does. If the user has access the method is allowed to execute, making it invisible that the aspect was ever invoked.

A.4.3. Client

The client part of the software system is used to view and input project and forecast data. Before the client is launched, it tries to detect if the user is on the EPCOS domain. If the client detects the user is on the EPCOS domain, this can be verified when the user is asked to fill in his username and password. After the user fills in his username and password (for inside users this is the NT username and NT password), a login function is executed to see if the user has access to the web service. Then the rights of the user are received from the web service, so that the client knows what functionality is available for this particular user.

The client is built up as a main form, containing a project list (CProjectForm) as child form and several panels. The project list is always loaded and can throw two types of events: the project selection of the user is changed and the project list is updated. The first even is used when another project is selected by the user and different information has to be displayed. The project list updated event can for example be used to re-sum totals.

The main form interprets the rights of the user by asking the various panels what rights they require. When all required rights are met, the panel is initialized and displayed onto the screen. All panels have access to the project list, but to keep panels independent, panels do not have access to other panels.

All forms are split up into two parts: a view and a controller part. This is part of an architectural style separating the interface with the controlling behind it. For tests, the view can be replaced by a mockup which makes testing the controlling functions easier.

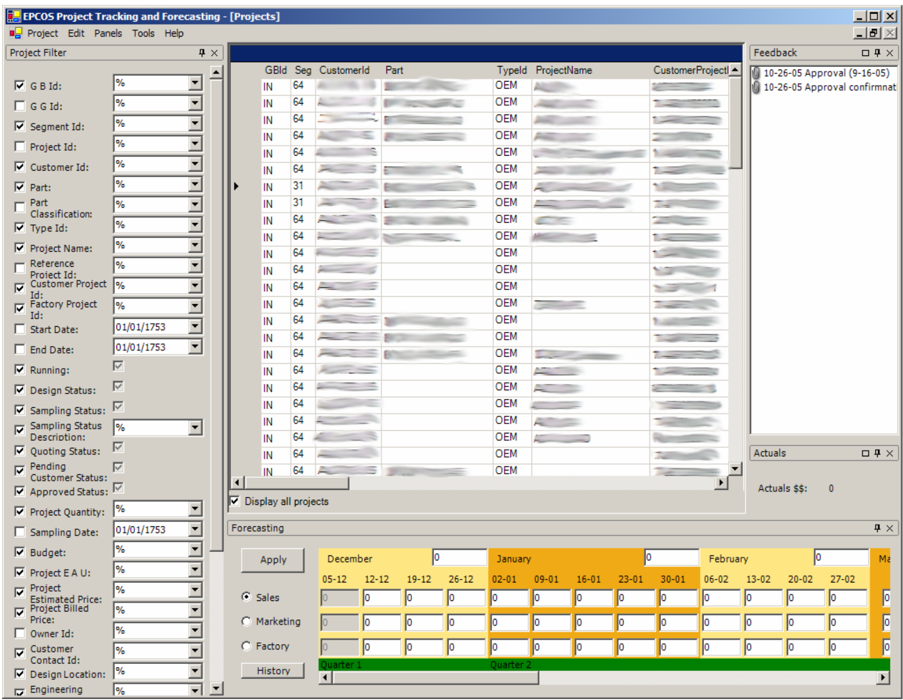

Fig. 15. Client screenshot

The user interface of the EPF application is displayed in Figure 15. The form presented in the middle of the screen is the project listing. Each user is able to view the project he/she is member of. The other windows, called panels, are loaded depending on the permissions of the user. The left panel is a project selection filter and has influence on which projects are displayed in the project list. The checkboxes on the left correspond to the columns in the project list and can be hidden by deselecting a checkbox. After the closing of the EPF application the column selection will be saved on the web service.

In the bottom panel, the Forecasting panel, users are able to input their forecasts in quantity for the selected project. The months are automatically generated and users can update the current months in their own permission category (Sales, Marketing or Factory). In the screenshot an Administrator is logged in and has permissions to change all forecasts. Each month the web service creates a new forecast automatically. When time passes by, weeks will gray out so that users cannot edit forecasts values from weeks in the past. All changes are logged automatically in the database via a stored procedure. Finally it is possible to display a history of forecasts categorized by the type of forecast.

The rightmost panel is used to add feedback to a project. This is a requirement which was later added and directed the project more towards a project management application. For each project it is possible to add a text message. Furthermore, every message can contain an attachment: a binary file. Attachments can be loaded directly from the application. Since users are currently using Outlook for all their project management, a plug-in for Microsoft Outlook was created. The plug-in is integrated into the Outlook environment and makes it possible to select an email and add it to a project. For this purpose a "light" version of the EPF application is started, which has a faster login procedure. The email is then added as attachment and can be opened from the EPF application. In this way project data, such as quotes can be directly stored into the database.

The interface is created by using several components and libraries. To create the docking panels a library called ".NET Magic" was purchased. Despite some small bugs the library worked well and the source is available. For the data browsing it came clear that the build-in controls of C# where not sufficient. The data browsing control (Combobox) is therefore extended with the following features:

- The Combobox automatically finds a matching result when text is typed into the text field (auto-complete, figure 16)
- Support for looking up multiple columns from a table and showing them in the drop down field

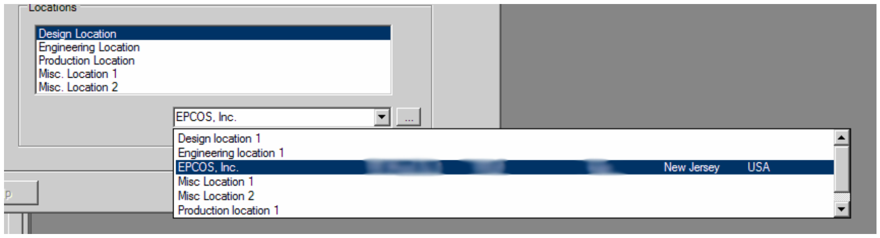

Fig. 16. Extended combobox

For the creation of projects a wizard control is used, guiding the user through the process. Changing projects can also be done via the wizard (figure 9), but can also be done via the project list in the main form. The projects can be changed by selecting values from the Comboboxes in the project list (figure 10). This required an extension to the Datagrid control in C#.

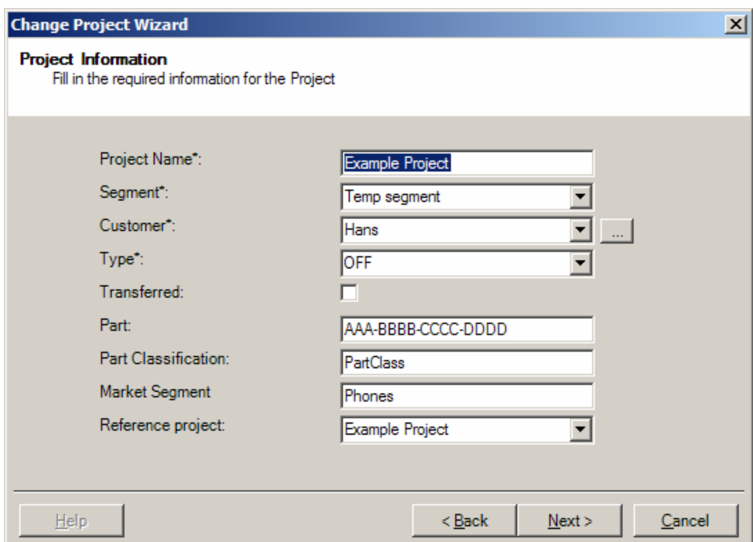

Fig. 17. Project Wizard

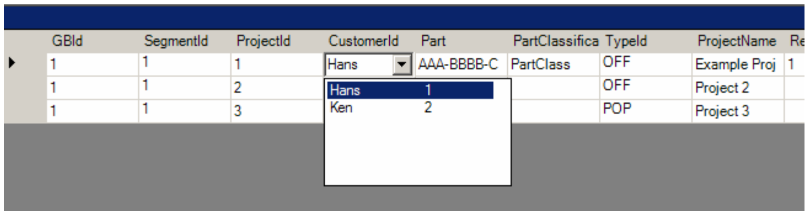

Fig. 18. Project Change in Project Listing

After various refactorings the architecture of the client resulted in a modular multithread application depicted in figure 19. When the EPF Client is launched, a username and password is requested. The NT username, if available, is automatically set in the username field. The client connects to the webservice URL which is saved in the registry by the setup when the application is installed. Given a correct password is supplied, two threads are started to load the main application. The first thread, the "main" thread, loads the graphical user interface. The second thread is a data loading thread, which fills the cache objects with data from the webservice. Only the data is loaded which is needed for the user. This is determined by looking into the right list of the user. The right list is retrieved from the webservice before initiation of the two threads. The main thread uses the right list to specify which panels have to be loaded. When the panels are constructed, they go to the cache objects for data. If the cache objects have already received the requested data, it is returned. Otherwise the main thread is suspended until all the data is received. Cache objects are made for several reasons. First, the cache object can re-supply data to various panels without going back and forth to the web service. Next, for internal purposes the webservice could be too slow for some users and it would be easily possible for change the cache objects to go to the SQL database directly. Another way to improve performance is to let the cache objects only load changed records. This could be done in combination with saving the cache objects to the disk, giving a major performance boost for the startup of the application. This is currently being implemented.

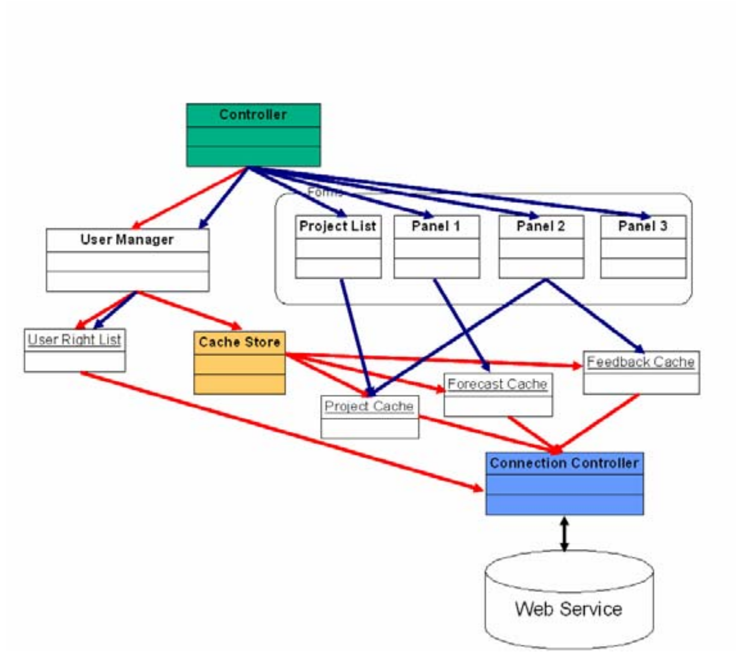

Fig. 19. Client architecture