ABSRACT

Ericsson Mobile Platform (EMP) is responsible of the development of a software platform and also to some extend responsible for related hardware parts. EMP is developing the data communication parts of the platform which is used by EMP customers. The platform development is done in large development programs and each program span over a quite a long time period. However, as we see every day in the shops mobile phone manufactures are launching new models more or less every month. EMP has many different mobile phone manufactures on its customer list and in order to meet their requirements/wishes, they need to add new functionality to the platform. The platforms are going to be released in the very near future or are half way in the development or in the start up process. The simple image that summarizes the problem is that adding new functionality to projects, which are in the conclusion phase, is not easy without delaying the project.

Thus, our intention in this thesis is to study how new functionalities in datacom platform could be added without jeopardizing the existing architecture of the running development projects of Ericsson. A solution that achieves fast process with high quality output and handles a customer requirement (CR) has been proposed in an efficient way. This is done by looking into current processes, working methods, tools, etc... Afterwards, a new mechanism has been investigated based on Test-Driven Development (TDD) as a main practice in Extreme Programming (XP). TDD is a method that recommends writing the tests at the same time, or even before the function to be tested. Verification of the proposed solution that explores an improvement in software development process is done by analysing, designing and implementing a new functionality related to Wireless Local Area Network (WLAN). The performance has been demonstrated first by spending less time in development phase. Thus, customer satisfaction can be reached with reducing time to market. In addition, writing high quality code within minor errors and bugs has been noticed. Consequently, improving product quality and reducing the cost of project can be achieved.

PREFACE

This thesis concludes two years Master of Science program in Information and Communication Technology (ICT) at the University of Agder (UIA), Faculty of Engineering and Science in Grimstad, Norway. This project has been carried out from January to May 2008 at Ericsson Mobile Platforms (EMP) of Grimstad, Norway.

I would like to send my gratitude and thanks to my internal supervisor Ole-Christoffer Granmo for his excellent guidance and precious advices during the fulfillment of this project. I am deeply grateful to my external supervisor Ole Dag Svebak for his brilliant supervision and useful advices during the technical part of this project at EMP.

I would like also to thank several people at EMP. First, I am grateful to Svein Thorstensen who gave me the opportunity to work with this thesis. I would especially like to thank Gjermund, Lars Olav and Andreas for pleasant and helpful technical discussions due to the challenges of the project.

I would sincerely like to thank the head of master's studies, Andreas Prinz, and the administration of the international office Stein Bergsmark, Sissel Andreassen and Tor Erik Kristiansen for their contributions.

Walid Trabelsi
Grimstad, May 2008

TABLE OF CONTENTS

ABSRACT ii

PREFACE iii

TABLE OF CONTENTS iv

LIST OF FIGURES viii

LIST OF TABLES x

CHAPTER 1 - INTRODUCTION 1

1.1 Basic Concept 1

1.2 Thesis Definition 4

1.3 Thesis Overview 5

CHAPTER 2 - SOFTWARE LIFECYCLE 7

2.1 Overview of Classical Method in Software Development 7

2.2 Software Lifecycle 8

2.3 Documentation 9

2.4 Software Development Process 9

4.1 Requirements Specifications 9

2.4.2 Design 10

2.4.2.1 The Static Point of View 11

2.4.2.2 The Dynamic Point of View 12

2.4.3 Detailed Design 12

2.4.4 Implementation 13

2.4.5 Modules Test 13

2.4.6 Modules Integration 13

2.4.7 Deployment 14

2.5 Maintenance 14

CHAPTER 3 - SOFTWARE DEVELOPMENT MODELS 15

3.1 Different Models 15

3.1.1 Waterfall Model (W. Royce, 1970) 15

3.1.2 V Model (Mc Dermid et Ripkin, 1984) 16

3.1.3 Incremental Model 17

3.1.4 Spiral Model 17

3.1.5 Iterative Model 19

3.2 Evolution 19

CHAPTER 4 - AGILE SOFTWARE DEVELOPMENT 21

4.1 Introduction 21

4.2 Values of Agile Manifesto 22

3.2.1 Individuals and interactions over processes and tools 22

4.2.2 Working software over comprehensive documentation 22

4.2.3 Customer collaboration over contract negotiation 23

4.2.4 Responding to change over following a plan 23

4.3 Principles of Agile Manifesto 23

CHAPTER 5 - EXTREME PROGRAMMING 27

5.1 Background 27

5.2 XP Practices 27

5.3 XP Development Cycle 31

5.4 Evaluation....32

5.5 Summary....32

CHAPTER 6 - XP PRACTICE: TEST DRIVEN....33

DEVELOPMENT (TDD)....33

6.1 TDD Overview....33

6.2 TDD Rules....33

6.3 TDD Implications....34

6.4 Test Driven Development Cycle....35

CHAPTER 7 - CASE STUDY....37

7.1 PART 1: Software Development within EMP....37

7.1.1 Platform Software Structure....37

7.1.1.1 Platform Group....39

7.1.1.2 Platform Layer....39

7.1.1.3 Platform Module....39

7.1.1.4 Software Backplane....39

7.1.2 Software Development Process....40

7.1.2.1 Current Software Development Process....40

7.1.2.2 Interpretation and Requirements....41

7.1.2.3 Proposed Development Process....41

7.2 PART 2: Datacom Platform (802.11 WLAN Specific Area)....43

7.2.1 802.11 WLAN Networks Generalities....43

7.2.2 General Architecture....43

7.2.3 WLAN Functionality: Power Management....45

7.2.3.1 Overview....45

7.2.3.2 STA Disconnection 45

7.2.3.3 BSS Connections 46

7.2.3.3.1 802.11 Power Save (PS) 46

7.2.3.3.1.1 Traffic Information Map and Delivery Traffic Information Map 46

7.2.3.3.1.2 Null packet data retrieval 47

7.2.3.3.1.3 Power Save Poll 47

7.2.3.3.2 Unscheduled-Automatic Power Save Delivery (U-APSD) 47

7.2.3.3.2.1 Trigger-enabled and delivery-enabled 48

7.2.3.3.2.2 Delivery-enabled 48

7.2.3.4 IBSS Connections 48

7.3 PART 3: Pilot Project Development 49

7.3.1 Project Management 49

7.3.2 Requirement Specifications 49

7.3.2.1 Optimizations for Periodic Traffic using Power Save-Poll 50

7.3.2.2 U-APSD with Local Wake-up Interval 51

7.3.3 Module Design 52

7.3.4 Module Implementation 53

7.3.5 Integration and Testing 59

CHAPTER 8 - DISCUSSION AND IMPLICATIONS 61

8.1 Theoretical Implications 61

8.2 Technical Implications 62

CHAPTER 9 - CONCLUSION 65

ACRONYMS AND ABBREVIATIONS 66

REFERENCES 68

LIST OF FIGURES

Figure 1.1: Software lifecycle 2

Figure 1.2: Test Driven Development mechanism....................4

Figure 1.3: Wi-Fi Network architecture....................5

Figure 1.4: Thesis structure....................6

Figure 2.1: Classical software development process 8

Figure 2.2: Construction of requirements specifications of the software from the existing needs 10

Figure 2.3: Example of hierarchical modules 11

Figure 2.4: Example of dynamic decomposition of an application....................12

Figure 3.1: Waterfall model. 16

Figure 3.2: V Model 17

Figure 3.3: Spiral model....................18

Figure 3.4: Evolution of software development model....................20

Figure 5.1: Extreme Programming practices....................28

Figure 5.2: XP development cycle 31

Figure 6.1: Test Driven Development steps....................34

Figure 6.2: Testing via xUnit Framework....................35

Figure 6.3: Test Driven Development Cycle 36

Figure 7.1: General example of Platform Software Structure....................38

Figure 7.2: Current Software Development Process....................40

Figure 7.3: Proposed Software development process 42

Figure 7.4: Wireless LAN architecture 44

Figure 7.5: TIM and DTIM intervals. 46

Figure 7.6: Power Save-Poll..47

Figure 7.7: Ad-hoc Traffic Information Map (ATIM) window 48

Figure 7.8: Power Save-Poll and Local Wake-up Interval...50

Figure 6.9: U-APSD with Local Wake-up Interval > 0 51

Figure 7.10: U-APSD with Local Wake-up Interval = 0 52

Figure 7.11: Ericsson Host Driver Architecture..54

Figure 7.12: Software Backplane and Platform Module Architecture...55

Figure 7.13: Simple Service Interface Implementation Overview with Iterative Process 56

Figure 7.14: Basic Use of Unit Test Framework 57

Figure 7.15: Core parts of Unit Test Framework 57

Figure 7.16: Example of console interface with some tests and suites failures 58

Figure 6.17: Example of console interface with successful tests and suites 58

Figure 7.18: Different targets of the application build..59

Figure 7.19: Generic Moses Window 60

Figure 8.1: Illustration of effort estimation and real spend time on developing activities............64

LIST OF TABLES

Table 1: Mapping between effort estimation and real spend time on developing activities…….63

CHAPTER 1 - INTRODUCTION

Whereas computer equipment made and continues to make a very fast progress, the software and the other ingredient of data processing, passes through a factual crisis. This crisis was detected probably forty years ago and it always proceeds until nowadays. The software crisis can be perceived through the following symptoms. The cost of software development is almost hard to predict and the delivery dates are rarely respected. Consequently, we focus a real problem to satisfy the customer requirements. Moreover, the delivered software quality is often defective. The product does not satisfy the customer requirements; it consumes more resources than predicted. This is the origin of breakdowns and change the customer requirement or adding new functionality to the projects. One more aspect, the maintenance of software is difficult, expensive and sometimes source of new faults. However, in practice, it is crucial to adapt the software because their environments of use change and the user's requirements are always in state of evolution and improvement. It is rare also to reuse existing software to make a new system, even if it contains similar functions. The main reason of the software crisis lies in the fact that it is much more difficult to create software than suggest our intuition. As the data-processing solutions primarily consist of immaterial components such as the programs, the data to be treated, the procedures and documentation, we underestimate easily their complexity. In fact, the programs size shows that this complexity is often quite real: "several packages and interfaces with a million lines of codes for each software development".

To manage the complexity of the software systems, it is very important to proceed with a good defined step. This phase must be based on principles and methods by using powerful tools. Thus, we reach to manage complex projects and figure out scientifically improved solution, which is of interest in this thesis, in software development process. Software engineering seeks to establish several engineering principles within certain goals for software development. At this point, it seems important to define a several inherent concepts in the software engineering related to software development process as described in [1].

1.1 Basic Concept

The software development must be seen like a process, so called a software process. This process is composed as well of sub-processes which communicate between different stages. Each process can be characterized by its release conditions, deployed activities, used resources, resulting products and extinction criteria. The activities, which belong to each process, can be broken up into more elementary activities and tasks. The software process can be the object of observations (what is done in practice?) or the object of theoretical studies (how is necessary to proceed?). There is not only one software process, neither in practice nor in theory for multiple reasons: the diversity of data-processing applications related to the software, the disparity between the methods, techniques and tools (including the programming languages) used to develop a software, and finally the differences in "culture" and habits for both organization and people. The term of software process is relatively recent. It is almost synonymous with software lifecycle. Thus, software lifecycle is the time spreading out between the start and the end of software process as shown in figure 1.1. When the lifecycle is subdivided, we obtain a several phases, specific stages within a quit many steps. The part of lifecycle devoted to the development is called software development cycle and corresponds to the software development process. The software development cycle starts with the decision to develop software and finishes with the delivery of the software surrounded by successful installation. The development is preceded by a preliminary phase which knows by concept exploration. It is called also pre-study phase. This phase has the aim to answer the following questions: Why developing the software? How to proceed to make this development? Which tools are necessary for the implementation?

As illustrated in [1], the result of the preliminary study is consigned in a document called project specification. It states the technical, administrative and financial specifications, the requirements, the limits of the project and its relationships to other projects. Since the software was developed, it will be put in exploitation. Between the two ranges a transitional stage, called installation. It is in general concerned by the team of software development. After the installation, it follows the operation and maintenance as a final phase of software development cycle. The software is now employed and its behaviour is supervised. This last activity called the maintenance of the software can be necessary to modify the software, to correct defects, to improve its performances or other characteristics, to adapt the software to new environment or to meet new requirements.

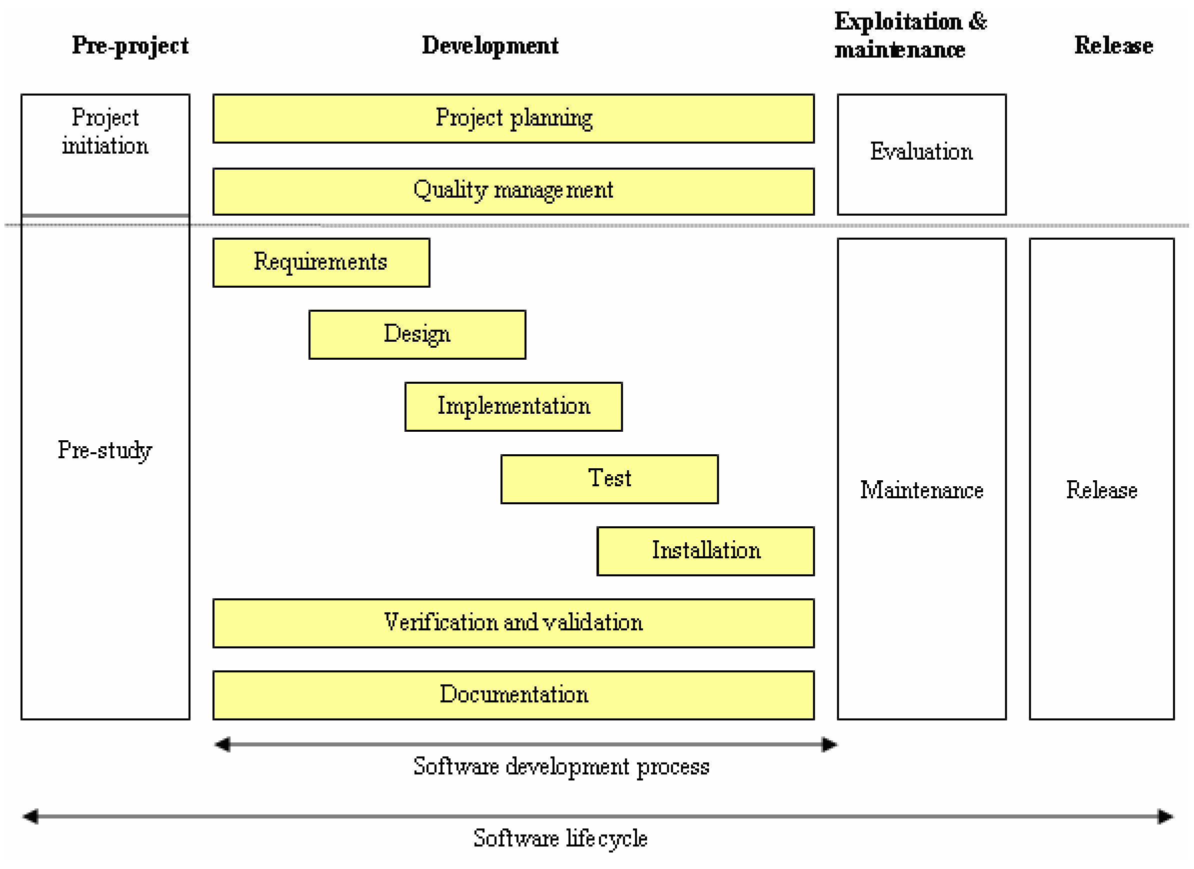


Figure 1.1: Software lifecycle

After having analyzed the main activities that support the whole development cycle, it is quit interesting to come across the different model for software development process. The primary function of software development process is to find out the order of the implied steps in software development and software evolution. During the history of software development, different models and approach were recommended to attempt the uncertainty in software development. The classical model is known as waterfall model. Not all the models of development cycle are the same. However, the idea is similar in most cases and the main phases always exist in different models with a specific mechanism. Thus, the software is developed in discrete phases. Each one has a definite result and a defined criterion of termination. Moreover, each phase is completed before starting the following one. The classical model in software development process includes the following phases: requirements, design, implementation, integration, test, and deployment. Each phase has a certain number of input and output. The output of each phase will be used as an input for the next phase. Some deployed activities are specific only for a precise phase and others are intended to prepare the following phases.

However, in practice, it is hard to adapt one model in software development because their environments of use change. Thus, improvement in software development process is always in state of evolution. According to [2] and [3], several alternatives of development cycle were made which constitute as many software development models .First of all, the elementary model has been used in data processing based on “code and fix”. It should well be admitted that it is still used very often today. This model includes two main stages: write a little code, then improve the code and correct the errors. Thus, we start by coding and afterwards we thinks about the requirements, the design, the tests and maintenance. This model involves many difficulties. First, after a certain number of corrections, the code becomes much badly structured that the later corrections become very expensive. This fact highlights the need for preliminary design phase before coding. Frequently, if the software did not satisfy the user’s requirements, it is either rejected, or developed again with large expenses. Thus, it perceives the need for a requirements specification phase before making the software design. As the tests and modifications are not prepared, it is expensive to correct the code.

According to these difficulties, the waterfall model came into sight. The software must be developed in successive phases, a phase being able to finish only when well-defined requirements are satisfied. Afterwards, The V model appears as an improvement of the waterfall model, each phase of the project has a testing phase that is associated with it. For certain types of software, this model is very well adapted, but not for all. That leads to new model so called evolutionary model. In this model, one frequently applies the classical development cycle within certain number of iterations. With this model of development, the users have quickly an operational system and a realistic base to determine the desired improvements of the product. Within this type of development, several models have been evolved focusing on the iterative process. The most know was the spiral model due to Boehm [4]. It is a complex model that integrates an evolutionary sight into the waterfall model based on risks analysis and management. The innovators continue always to seek improvement in software development process. [7] and [8] prove that the state of art in software engineering is so called by agile software development. Agile is an iterative and incremental approach to software development that promotes iterations development throughout the project lifecycle. Agile development encloses a set of values and principles that clarified the new concept in software development process.

1.2 Thesis Definition

This thesis has been carried out at Ericsson Mobile Platforms (EMP). EMP is responsible of the development of a software platform. EMP is developing the data communication parts of the platform which is used by EMP customers. Nowadays, a big progress has been achieved in mobile phone manufactures within new functionalities. Consequently, in order to meet the costumer requirements, EMP need to add new functionality to the platform. In this context, we have, as a first challenge related to software engineering, to study how software development process is preceded inside EMP. We should look into the current processes, working methods, tools, etc… Afterwards, our intention is to investigate how new functionalities in datacom platform could be added without affecting the existing architecture of the running development projects. A solution that achieves an improvement in software development process shall be proposed within an efficient way. In this context, agile software development, as considered in the previous section 1.1 that is the state of art in software development process, has been concerned. Agile development encloses a certain number of values and principles. Hopefully, almost of these values and principles are applied inside EMP which encourage us to follow this new concept. In literature [10], several agile methods are defined as Scrum, Feature Driven Development, Crystal Clear and Extreme Programming which is concerned in this thesis. Extreme Programming (XP) can be considered the most known of agile methods. XP is a discipline of software development based on values of simplicity, communication, feedback, and courage. It is a software development process which encloses a certain number of practices. In [17], Kent Beck has suggested Test-Driven Development (TDD) in association with XP, one of several practices in software development process. It is also the main interest carried to agile methodologies which contributed in the software process improvement. Although TDD is one of the keystones of XP, it can easily be approved to be used in any iterative software development process without adopting any of the other practices of XP. Thus, we have motivated to investigate TDD method that could be introduced in EMP process. Especially, this practice was not used before at EMP in software development process. Therefore, we have the opportunity to test TDD practice in order to check its performance.

TDD method presents the unit tests, called "programmer's tests", in the heart of programming activity. TDD recommends writing the tests at the same time, or even before the function to be tested. Figure 1.2 shows the main mechanism of TDD: write a test that fails and then write the code to pass it. However, it seems a better way to create software. Thus, writing a test is a quick means to find out if you appreciate the requirement specification.

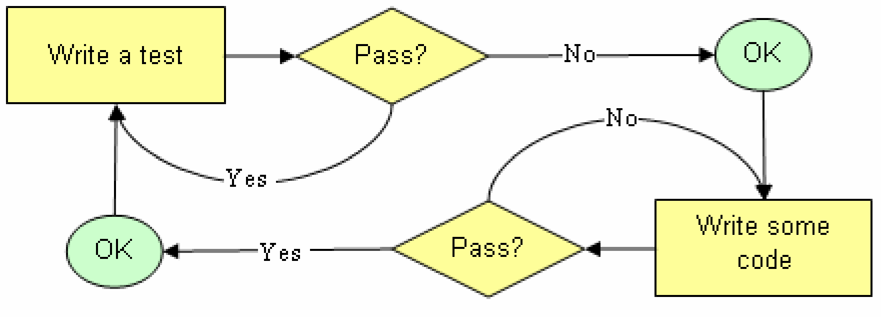


Figure 1.2: Test Driven Development mechanism

After fixing the proposed solution using TDD expected to achieve an improvement in the software development process, it is time to tackle the practical part intended for applying the proposed solution in specific environment. This specific environment will be related to data communications technologies presented by EMP. This is our second challenge of research that encloses the technical part of the thesis. We have to gain knowledge of Wireless Local Area Network (WLAN) as main trends in data communication technology in EMP. Thus, we will apply what we find out in software engineering research to an embedded software development as a pilot project. WLAN presents certain major functionalities such as connection management, scanning, power management, roaming, security, ect… In this thesis, we are interesting only in power management functionality and particularly related to Wi-Fi Multimedia (WMM) power save aspect. WMM power save is a set of features for Wi-Fi networks that help conserve battery power in small devices such as phones, PDAs, and audio players. It uses mechanisms from the recently ratified IEEE 802.11e standard, and is an enhancement of legacy 802.11 power save. It allows devices to spend more time in a "dozing" state, which consumes less power, while improving performance by minimizing transmission latency. In this context, the client dozes between packets to save power which an application picks time to wake up and transmit uplink trigger for downlink packets. Despite the fact that the access point buffers downlink packets while client dozes. Thus, the access point waits for client to wake up and send trigger frame. In this context, "Local wake-up Interval" as a new functionality must be added in WMM power save.

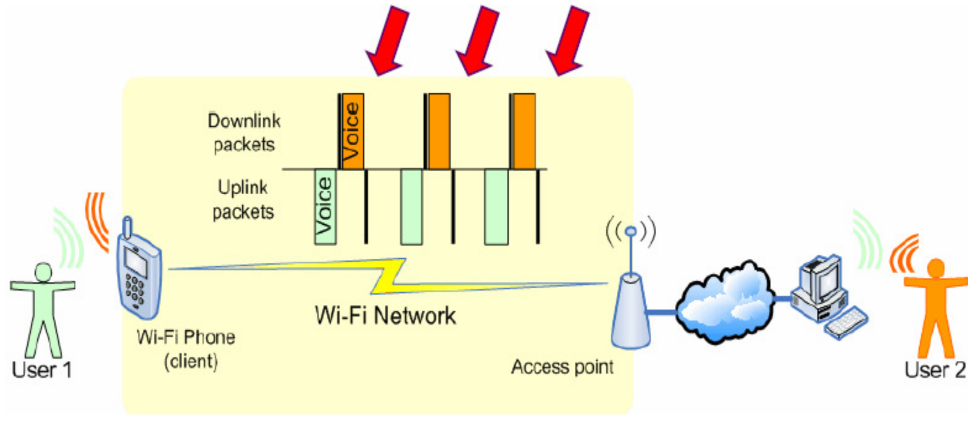


Figure 1.3: Wi-Fi Network architecture

Throughout this thesis, we gain knowledge from both software engineering and data communication technologies.

1.3 Thesis Overview

This thesis is organized as follows. In Chapter 2, the software lifecycle is established and the different phases of software development process are briefly described. The most important models of software development are investigated in Chapter 3. Chapter 4 constitutes the description area of agile software development process. First of all, the main values are described and then the principles are derived. The extreme programming is introduced with a brief description for each XP practice in chapter 5. The test-driven development is investigated in Chapter 6. First, we present the rules and implication of this concept of development, after that, the general TDD cycle is inspected. In Chapter 7, we describe the different steps of our pilot development project as a case study. First, we illustrate the architecture of EMP platforms which include the existing software development process. Afterwards, we propose our solution for software development process improvement. Moreover, we describe power management as WLAN functionality in datacom platform. Finally, we explain the main stages occurred during the development project. Discussion and implications are introduced in chapter 8 to evaluate the work and to present the limitation of thesis. First, the theoretical implications are investigated. The latter is the starting point for technical implications happened during the project. In chapter 9, we draw the conclusion.

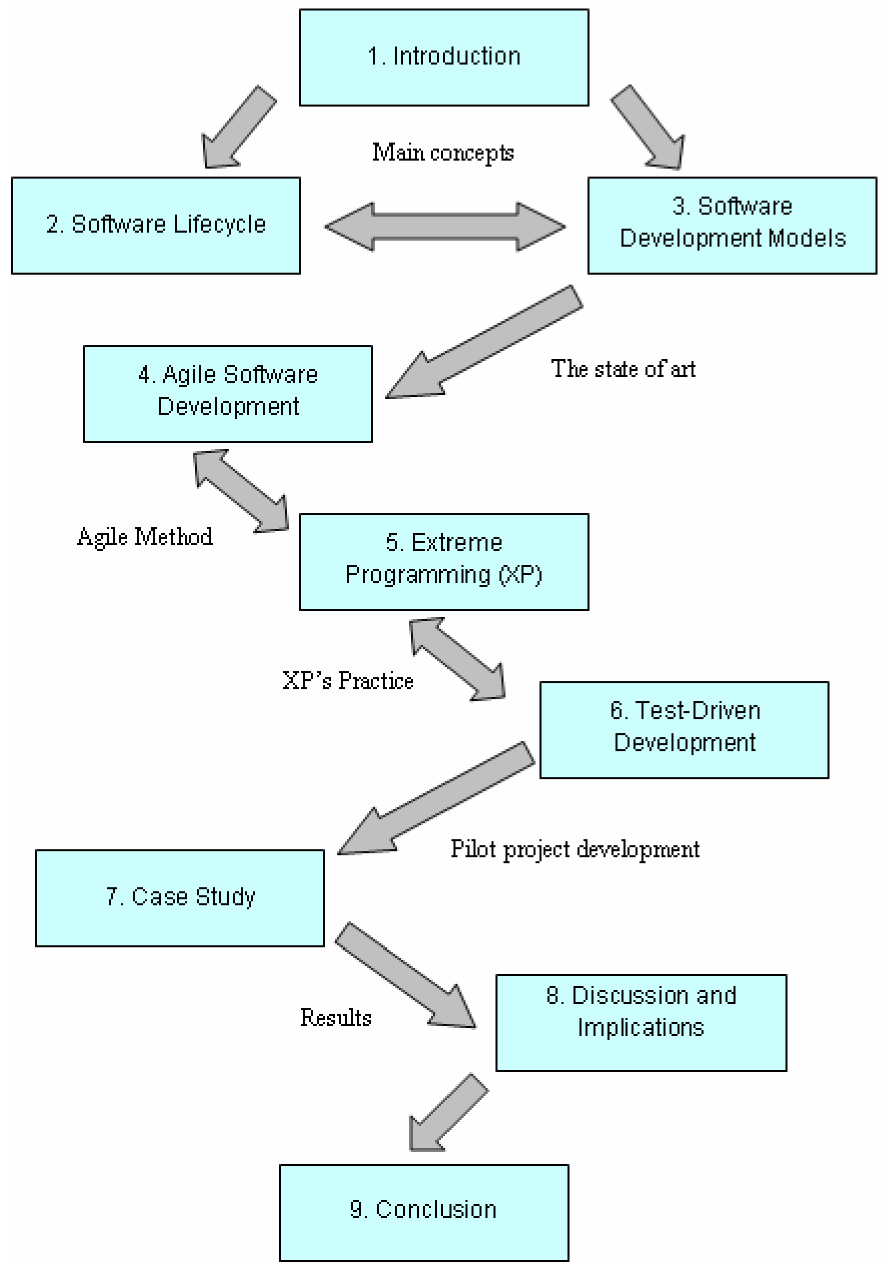

1. Introduction
Main concepts
2. Software Lifecycle
3. Software Development Models
The state of art
4. Agile Software Development
Agile Method
5. Extreme Programming (XP)
XP's Practice
6. Test-Driven Development
Pilot project development
7. Case Study
Results
8. Discussion and Implications
9. Conclusion

Figure 1.4: Thesis structure

CHAPTER 2 - SOFTWARE LIFECYCLE

Software lifecycle indicates all the stages of software development process. All these phases of software development start by specifying the customer requirements until the software achievement as a commercial product. The objective of such a plan is to define the various intermediate phases required to validate the software development and conform the requirements specification for an explicit application. This lifecycle contains a variety of activities in different development parts. We focus on improving the software development process at EMP in this thesis as a main objective. Thus, we start by studying how is classical software lifecycle and which activities are included. In this chapter, we present the different phases of software lifecycle. Afterward, the general architecture of software development process is presented. Moreover, we describe briefly each phase step in software development process.

2.1 Overview of Classical Method in Software Development.

The software lifecycle is the cradle to give existence of a software product or software intensive system. Management of the entire software lifecycle requires a deeper knowledge than based in the small development intuition and experience. Software development process project can distinguish two major stages which it is necessary to dissociate in order to understand this chapter.

The first can be considered as a research or a feasibility study of the project. The user has an unclear idea of the requirements and especially does not know how to accomplish these requirements. It does not know also which solution is feasible. There are only a several ideas that should be implemented and validated. In many companies, research departments carry out this kind of work. The expert systems are in general used because they are adapted for that. The second is more concrete development which does not have a problem of feasibility. It is clear and possible to define the software requirements. It is known in advance that there is a feasible solution to carry out the software. Inside such organization, persons of development departments of operational software are investigated this type of work.

A traditional method of development applies at the second stage without being interdict to use some principles in the first stage. We present next sections the principles stages of a classical development process.

2.2 Software Lifecycle

Software development is done according to specific cycle called the software lifecycle. Boehm suggests in [3] that the main stages of software lifecycle for development are:

- Requirements Specifications,
- Design,
- Detailed Design,
- Implementation,
- Testing
- Integration,
- Deployment,

These phases are spread out in time. One phase finished in the handing-over of one or several documents validated by both the user and the developer. Development phases come to an end when the review of this phase is made. Next phase can start only when the previous one is finished. Both of the user and the developer agree at the end of each phase. The decomposition in phases of development allows the user to follow the different steps of project. Figure 2.1 presents the classical software development process:

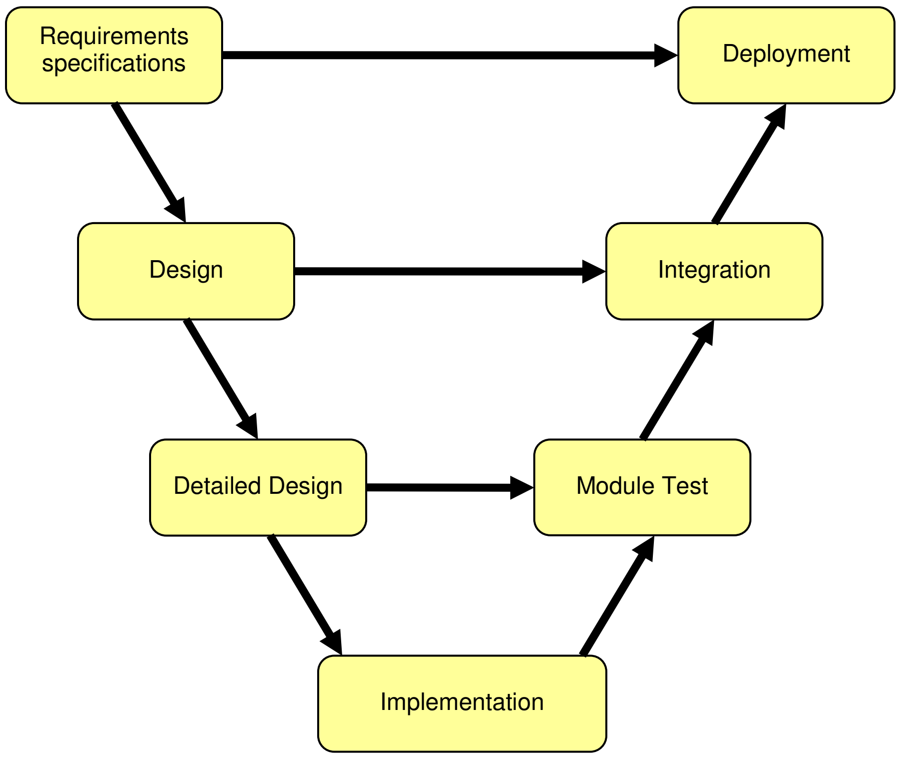


Figure 2.1: Classical software development process

The first phases make it possible to decompose the whole of the project to simplify the implementation phase (Top-Down). The following phases recompose the whole software which is tested in each stage (Bottom-Up).

2.3 Documentation

During a project, communication is very important as first quality between people who interfere closely or by far in this project. If all the speakers are animated by this common, the success of the project is very largely favoured. If not, the failure is certain. The information exchanged between the participants contains two parts. First, one is an abstract exchange of information which does not need to be structured and which is used to create an environment of work, a motivation, etc… This information in general is transmitted orally face to face. The other part is a more technical exchange of information which must be structured. Documentation makes it possible to meet this requirement.

The objective of documentation is to allow the transmission of information, to make visible the software product throughout the lifecycle. In particular, at the end of the project, the documentation, which interests the users of the software, must be ready. Documentation is used as support of work. It ensures the conservation and the transmission of knowledge. In addition, it facilitates the management and the control of the project [1].

Documentation is progressively produced during the project development phases. The documents validated at the end of each phase are used as a basis of work for the different activities of the following phase. In particular, that quickly allows external people to be familiar with the project in middle phase.

2.4 Software Development Process

4.1 Requirements Specifications

It is essential to establish the software requirements during the first phase. The requirements can be converted in several forms:

- General specifications: a set of objectives, constraints (use of materials and existing tools) and general information which will be necessary to respect during the development
- Functional specifications: the description of the software functionalities in a way as detailed as necessary.
- Interface specifications: the description of the software interfaces with the external world (persons, other software, hardware…) in detailed manner.

The requirements specifications are used to define what the software must do and not how it is made. This is described in the document of requirements specifications. Figure 2.2 shows that the goal of the first phase of development is to specify the software requirements starting from the existing needs.

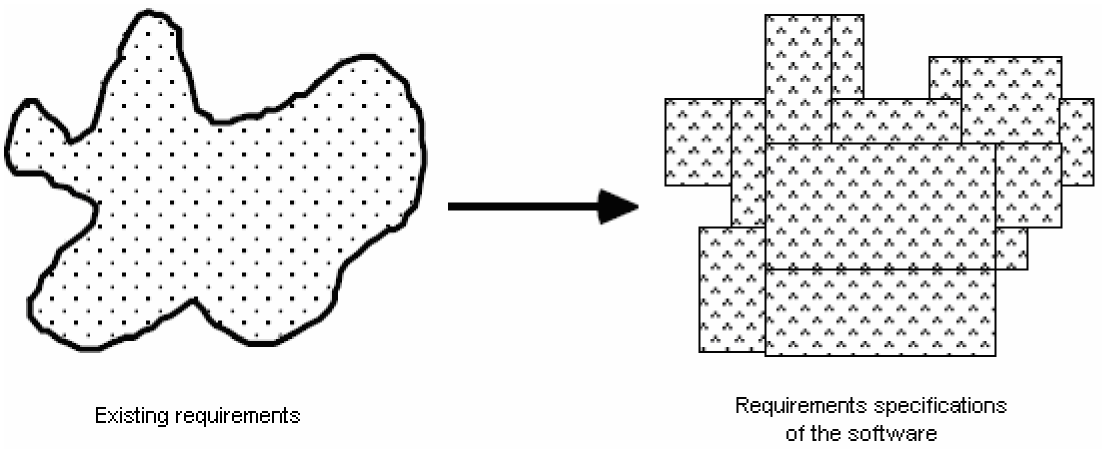


Figure 2.2: Construction of requirements specifications of the software from the existing needs

The methods and techniques used to produce the software are studied in parallel. There are described in the development plan of the software that exposes the organization of the project (decomposition in tasks, structure of the teams, planning, documentation, tests, and quality evaluation…). This phase which constitutes approximately 15% of the total time of the development finishes by the review of the functional specifications. This phase can be also called "external specifications" or "Requirements analyze" according to the vocabulary employed. The documents produced during this phase are software development plan, software requirements specifications and the receipt book.

2.4.2 Design

A first step in the process of software design, starting from the requirements specifications, focuses on the definition of the software architecture. The phase of general design let to consider several solutions with the problem arising and to study their feasibility. For each solution, the choices carried out are noted with their reasons in order to distinguish the real constraints of the project. The solution meeting best the expressed requirement is adopted and solidified. We can make prototypes of the various approaches under consideration in the general design in order to validate them. The document of general design of the software describes the adopted solution. A general presentation of the structure of the software is made.

The structure of software can be seen with two different points of view:

- The static point of view,
- The dynamic point of view.

2.4.2.1 The Static Point of View

The static point of view consists in cutting out the software in modules if we use a traditional language. If we exploit an object oriented language, the objects of the different interfaces should be defined.

A module is a compilation unit gathering a whole of functions or procedures, definition of data structures. A module is composed of two parts:
- A visible interface of the other modules which can use this module
- A hidden and invisible body of the other modules.

A module enables the visibility of its body and its interface obviously, but also the modules interfaces which it uses.

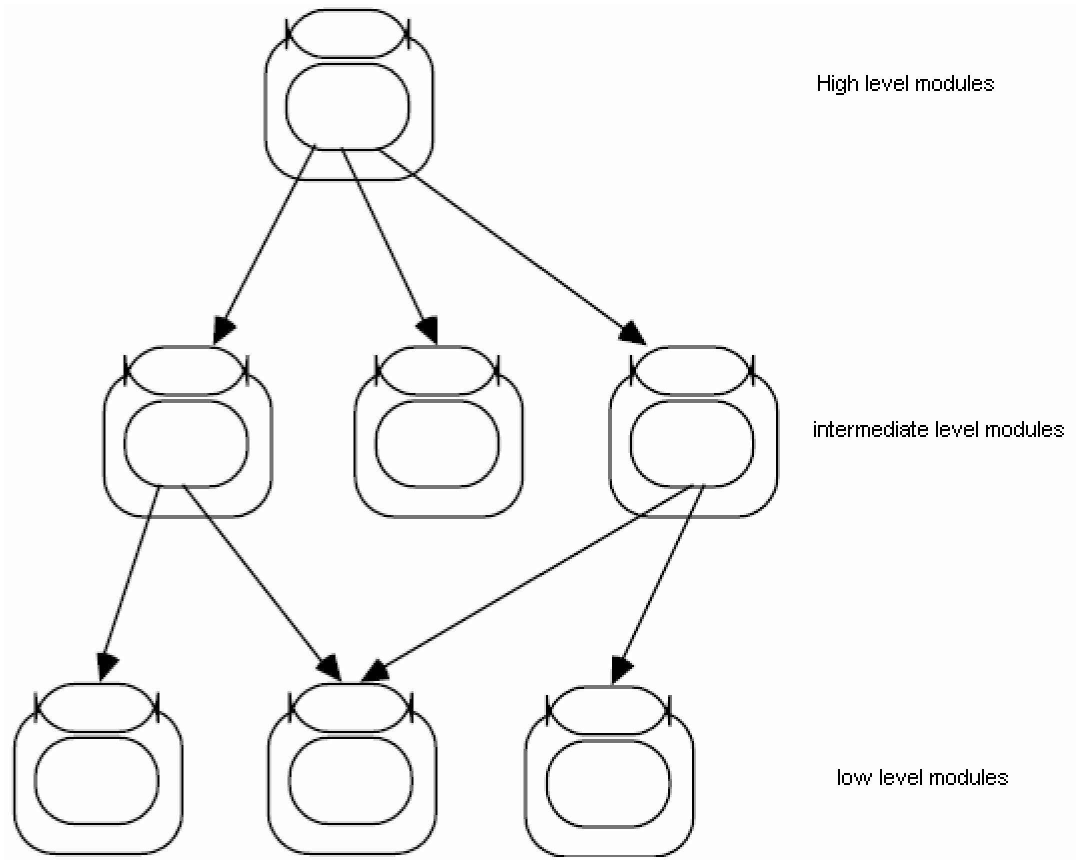


Figure 2.3: Example of hierarchical modules

The module interface constitutes the contract which the module is committed to respect. The body of the module is used to fill this contract. One module tries to gather functions, procedures and data which have common package related to the same concept. An object collects methods and attributes which in common have the object itself.

2.4.2.2 The Dynamic Point of View

The dynamic point of view consists in cutting out the software in tasks which can be carried out in parallel (multi-task if necessary) as shown in figure 2.4. A task is the code execution included in one or more modules. Decomposed tasks enable to separate the independent treatments and to simplify them. Thus, it is guided by the application aspect of the software.

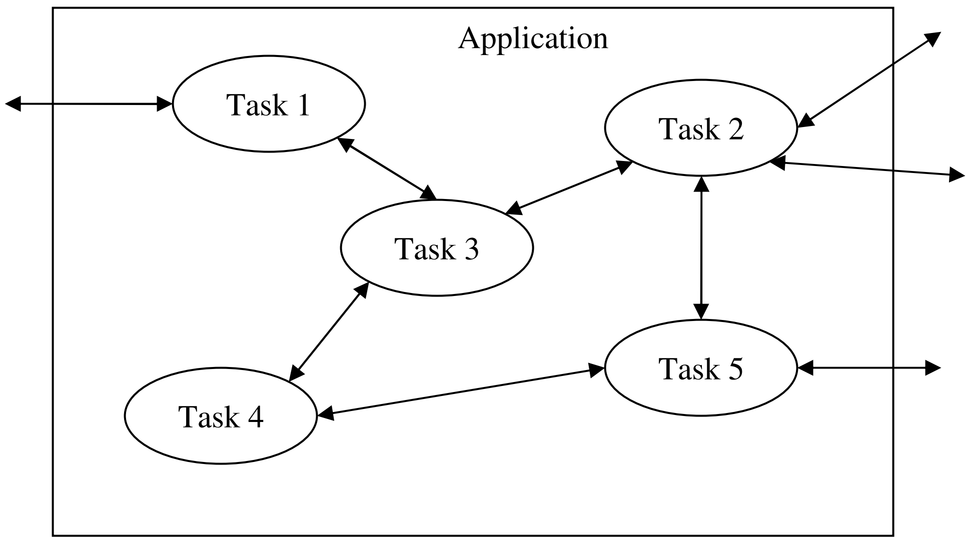


Figure 2.4: Example of dynamic decomposition of an application

If choices of development tool were specified in the general specifications, decomposing in tasks can depend on these choices (for example for a Multilanguage's application). The design phase depends on the development tools if those were specified in the general specifications. If several tools are used, it should be found how to use them to carry out such part of the system and how to interface them. Static and dynamic decomposition depend then also on these tools.

If no development tool were specified in the preceding phase, the choice of the tools can be adapted to the application type in order to simplify static and dynamic decomposition. This phase which accounts for approximately 10% of the total time devoted to the development finishes by the review of general design. This phase can be also called "global design" according to the vocabulary employed. The document produced during this phase is general design document.

2.4.3 Detailed Design

The second stage of software design process allows following the software decomposition, starting from the result of the general design until reaching an external description of each procedures and data structure. In the case of the use of a modular language, this phase consists in defining the interfaces of the modules precisely. In the case of the use of an object-oriented language, this phase consists in defining the objects contents precisely (attributes and methods).

The document of detailed design presents the detailed software architecture. The manual instruction of the software describes the software implementation and its environment. The tests cases which must be implemented during the integration phase are described in the specifications document of the software integration tests. It can also describe the contents of the procedures with a pseudo-language. This phase which accounts for approximately 25% of the total time devoted to the development finishes by the review of detailed design. The two phases of design can be gathered with the term of "internal specifications". The documents produced during this phase are detailed design document, manual instruction and integration tests specification.

2.4.4 Implementation

The procedures identified in the previous phase are coded and tested individually. The result of this phase is the source code and the results of the unit tests. In the case of the use of a modular language, this phase consists in coding the modules bodies depending on their interface. This phase which accounts about 15% of the total time devoted to the development finishes by the review of coding and unit tests. The documents produced during this phase are results of the unit tests.

2.4.5 Modules Test

Each module is tested individually. It is checked that the services specified by the module interface investigated the requirements specification. This phase which accounts for approximately 5% of the total time devoted to the development finishes by the review of modules tests. This phase can be possibly combined with the unit tests of the preceding phase. Results of the modules test is the document produced during this phase.

2.4.6 Modules Integration

The various modules of the software are gradually integrated in successive levels regarding the integration tests specifications. The phase of integration resembles a construction where each building block of the software is associated to form an entity which is used again to build a new block, ect… until leading to very whole construction.

A first level consists in testing those modules functioning correctly together. The following level consists in testing that a task functions correctly by using the modules on which it is pressed. The next level consists in testing that several tasks bound by multi-task mechanisms (signalisation, mutual exclusion, shared memory and more generally communication) function well together. Finally, the final level consists in sharing all the software tasks for an overall test. The test cases, procedures and results of the tests are consigned in the result of the integration tests document. A document gathers the procedures to be followed to pass from the source code to the object code of the software. This phase which accounts for approximately 20% of the total time devoted to the development finishes by the review of overall software integration. The documents produced during this phase are the presentation of the software and results of the integration tests.

2.4.7 Deployment

The software itself is integrated in the external environment (other software, users). It is checked that the developed software meets the expressed requirements in the phase for requirements specifications. This phase which accounts about 10% of the total time devoted to the development finishes by the final review. Receipt result is the document produced during this phase.

2.5 Maintenance

Maintenance is an activity which includes the user training and the technical assistance. It begins with the software delivery and it is completed at the end of the exploitation system.

Maintenance can be:

- Corrective: it is not compliance with the specifications, from where detection and correction of the residual errors.
- Adaptive: modification of the environment (material, software supplies, tool,…)
- Evolutionary: change of the software functional specifications.

The activity gets ready during the development and applies then to the commissioned operational software. When the modifications represent a notable part of the development, it is regarded as a leaving the framework of the maintenance and to be treated like an ordinary software project.

The activities of maintenance cover the following fields:

- Qualifications of the new versions,
- Follow-ups of the modifications,
- Filing,
- Update of documentation,
- Execution of certain modifications

We have to mention that the maintenance of software is difficult, expensive and sometimes source of new errors. However, in practice, it is crucial to adapt the software because their environments of use change. This variation depends on different models of software process which constitute the object of the next chapter.

CHAPTER 3 - SOFTWARE DEVELOPMENT MODELS

After having reviewed the main activities that support the whole development cycle, we will describe the different model and cycle of development. The classical model in software development process is known as waterfall model. Not all the models of development cycle are the same. However, the idea is similar in most cases and the main phases always exist in different models with specific mechanism. Thus, the software is developed in discrete phases. Each one has a definite result and a defined criterion of termination. Moreover, each phase is completed before starting the following one. The classical model in software development process includes the following phases as described in chapter two: requirements, design, implementation, integration, test, and deployment. However, the interaction between these different phases differs from one model to other. Therefore, we try to figure out which model can be used at EMP organisation in order to improve the software development process. In this chapter, we present the different software development models following its evolution instance. In addition, we describe briefly the major advantages and drawbacks for each model.

3.1 Different Models

The primary function of software development process is to find out the order of the implied steps in software development and software evolution. It is necessary also to set up the alteration criteria to progress to the next stage (Boehm 1988) [4]. During the history of software development, different models and approach were recommended to attempt the uncertainty in software development.

3.1.1 Waterfall Model (W. Royce, 1970)

"The waterfall model is a sequential software development model (a process for the creation of software) in which development is seen as flowing steadily downwards (like a waterfall) through the phases" [5]. Each phase must be approved before being able to start the other. The original model did not include the possibility of return behind. The waterfall model with backtracking improves the reactivity of the model. This model is strict and heavy. A later change is expensive. It is adapted if the requirements specifications are clearly identified and stable. Figure 3.1 shows that the progress of the model flows from the top to the bottom, like a waterfall.

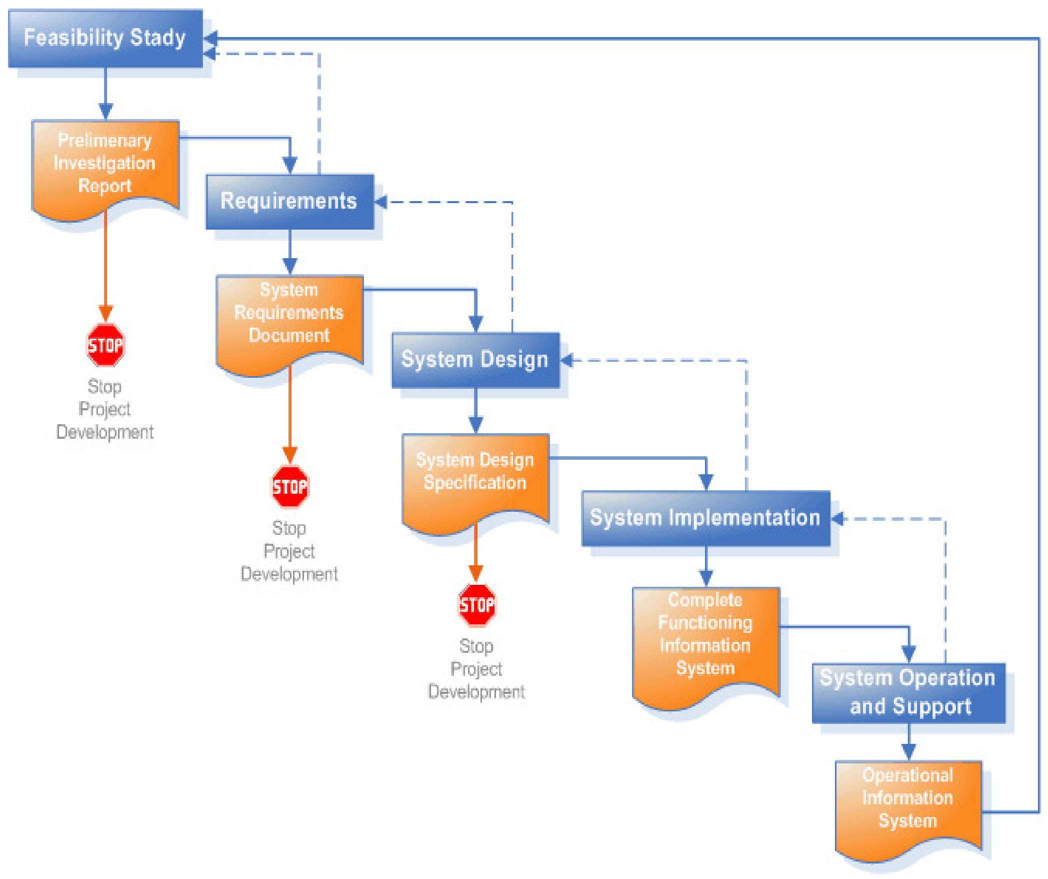

Feasibility Stady
Prelimenary Investigation Report
Stop Project Development
Requirements
System Requirements Document
Stop Project Development
System Design
System Design Specification
Stop Project Development
System Implementation
Complete Functioning Information System
System Operation and Support
Operational Information System

Figure 3.1: Waterfall model.

3.1.2 V Model (Mc Dermid et Ripkin, 1984)

This model is an improvement of the waterfall model, each phase of the project has a testing phase which is associated with next one. The phases of the rising part must return information on the phases in opposite when defects are detected in order to improve the software.

The main disadvantages of V model are the absence of prototyping and the difficulty to follow the requirements evolution… The V model bringing the light the need to anticipate and to prepare, in the downward stages, the "expected" of next rising phases: thus as shown in figure 3.2, the expected from system testing are defined at the time of the software requirements specification , the expected from unit testing are defined within the program specification, ect…

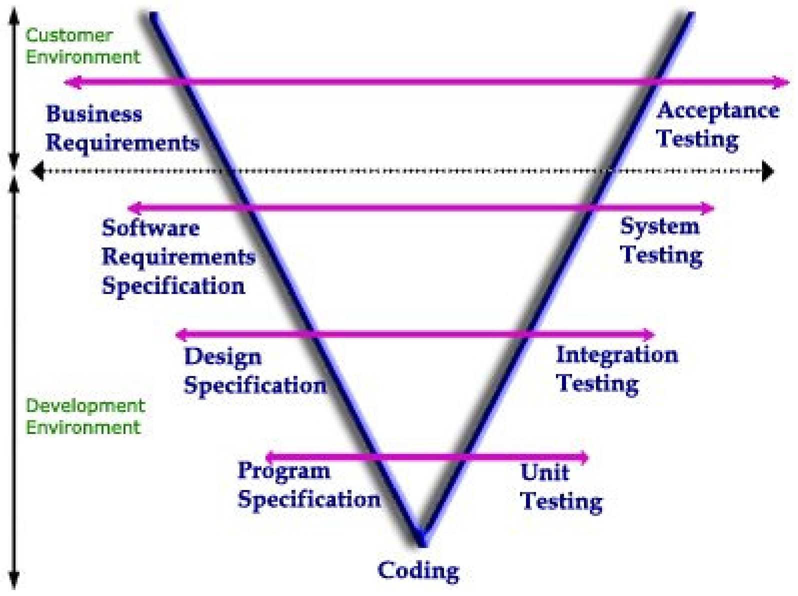


Figure 3.2: V Model

3.1.3 Incremental Model

In the waterfall model or V model, the components are developed independently from/to each other. In the incremental model, only a subset is developed at the same time, initially a software core, then successively, the increments are developed and integrated. It starts by defining the requirements and it breaks up them into subsystems. To each software version, new functionalities coming to fill the requirements are added. It continues until all the required functionalities are filled by the system. An increment is a version of the system: an increase brought during a system construction.

The advantages are that the development is less complex and integrations are progressive. The disadvantages are related to the core of development, increments size and within the difficulty of integration.

3.1.4 Spiral Model

The spiral model is a complex model which integrates an evolutionary sight into the waterfall model based on risks management. This iterative model underlines the activity of risks analysis due to Boehm [4].

Risks Analysis can be summarized with the following symptoms:

- **Human risks**: personnel failure, over-estimate of competences, solitary worker, heroism, lack of motivation…

- **Process risks**: no project management, calendar and budget unrealistic, calendar given up under the pressure of the customers, data insufficiency, validity of the requirements, development of inappropriate functions, development of inappropriate user interfaces…
- **Technological Risks**: change of technology in process, performance problems, incomprehension of the technology bases …

This model of development combines several cycles as seen in figure 3.3. Each cycle of the spiral proceeds in four phases:

- Determine of the objectives, the alternatives, the constraints starting from the results of the previous cycle. For the first cycle is starting from a preliminary requirement analysis.
- Risks analysis and alternatives evaluation.
- Development and checking of the adopted solution.
- Review of the results and checking of the following cycle.

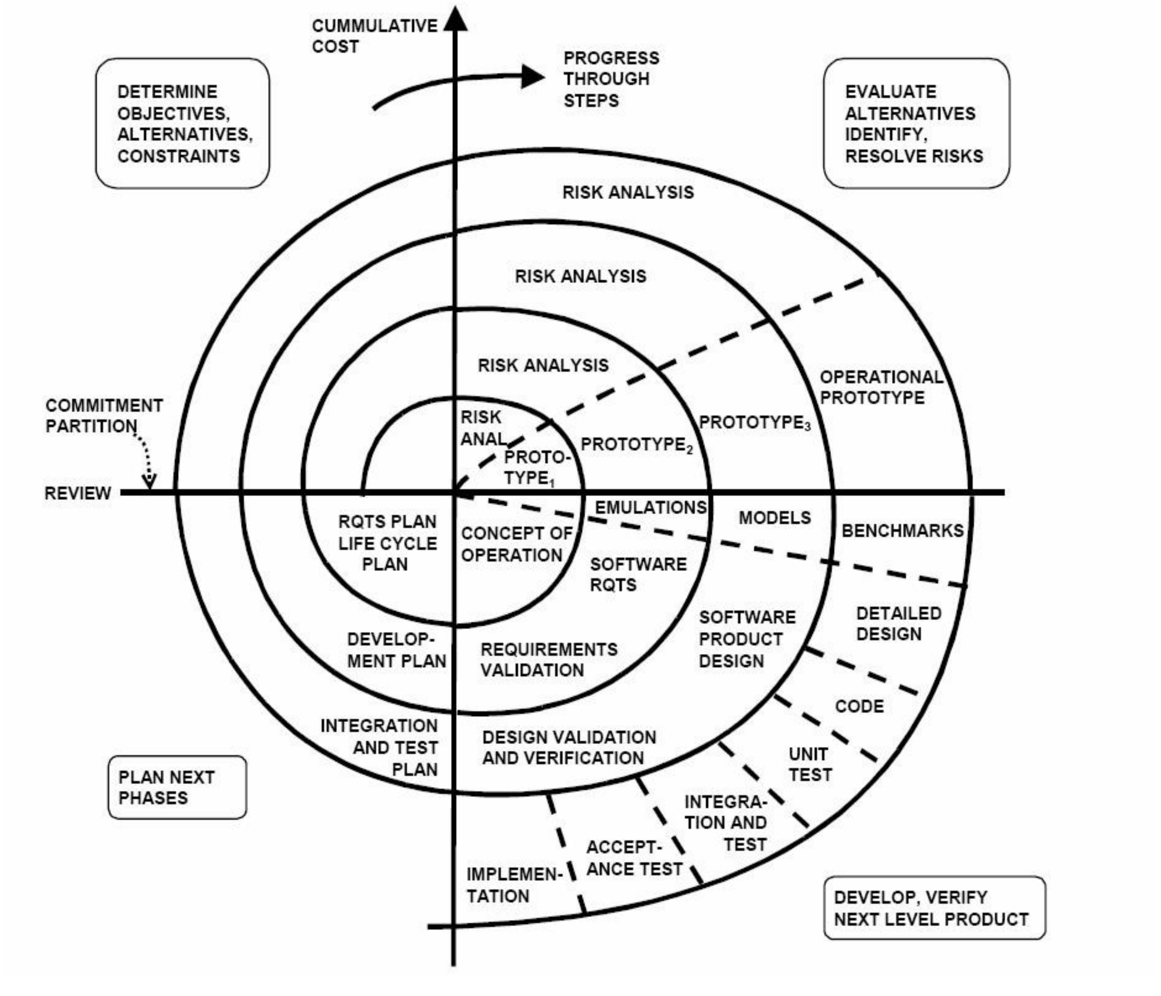


Figure 3.3: Spiral model

3.1.5 Iterative Model

With each new requirement, the iterative model proposes to study feasibility, then to imagine its development, after that passing to its manufacture and finally to the delivery step. Contrary to the incremental model, the versions of the software are not intended in advance.

The goal of an iterative approach is to improve quality permanently. It is composed of 4 stages: (Plan Do Check Act: PDCA). The idea is to deliver quite possible version which can be tested by the customer.

Plan: what should be envisages to do, study feasibility, specifications.
Do: it is imagined how it will carry it out: elaboration, architecture.
Check: checking and measurement of the risks
Act: tests, validation and delivery with the customer,

3.2 Evolution

The description of the various models of software development process which was carried out in the preceding parts follows the evolution of these models. Indeed, at the beginning, it did not have a really software development cycle but the method “codes and debug” was used.

Then, it appreciated that this method was not adapted to the evolution of the data-processing projects. The projects becoming increasingly large, a certain number of phases were highlighted. Afterward the first model of software lifecycle (waterfall model) was given. That was a factual improvement of the data-processing project management. However, this model was limited because it produced a great tunnel effect. In order to minimize this problem, the V model was developed, but this model, just like its predecessor, provided only a tardily version to be validated. Thus, the incremental model was setting up. But great problems of the software development are the constant evolution of the user's requirements. Because change request problems were not considered in the incremental model, the spiral model was thought.

The cycle of spiral model being independent has a problem of big-bang effect that persisted during the integration phase. Hence, the iterative model, which consists with the successive application version and new functionalities are added during the following iterations, was established. At present, taking in consideration the ceaseless evolution of data-processing development requirements succeeds with an iterative development reflecting the state of the art. Figure 3.4 illustrates the evolution of these different models of Software development process.

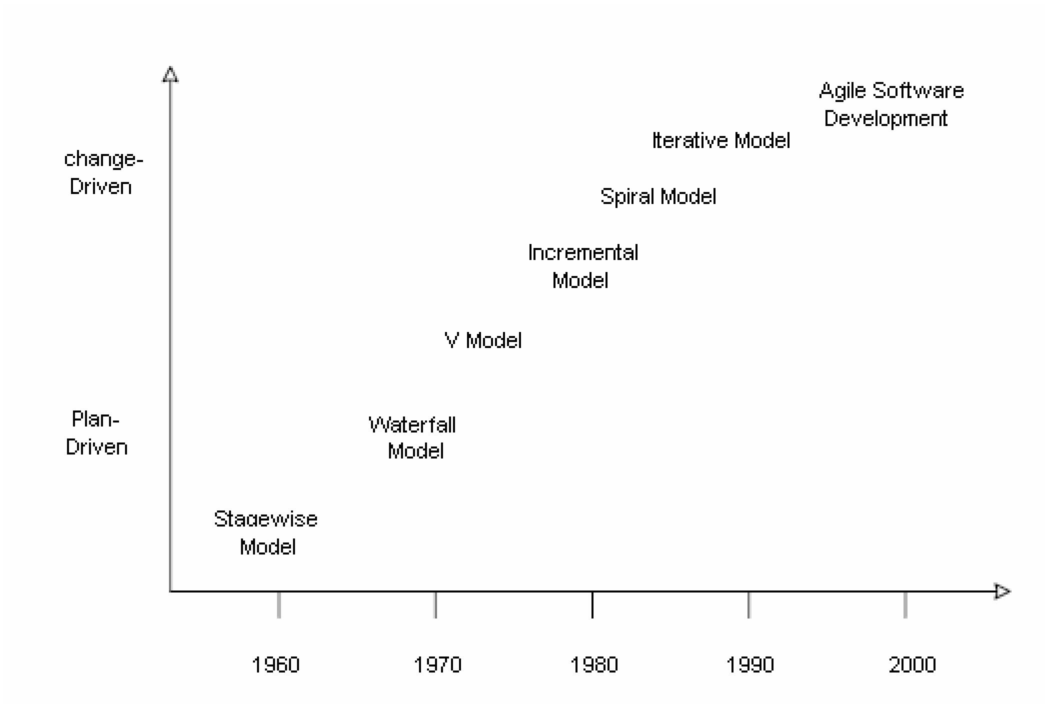


Figure 3.4: Evolution of software development model

The state of art in software engineering is known by agile software development. Agile is an iterative and incremental (evolutionary) approach to software development that promotes iterations development throughout the lifecycle of the project. Agile software development will be discussed in detail the next chapter.

CHAPTER 4 - AGILE SOFTWARE DEVELOPMENT

The concept of agile software development promotes development iterations throughout the lifecycle of the project. The agile methods make it possible to conceive software by implying the customer as much as possible, which allows a large reactivity for customer requests. The agile methods tend to be more pragmatic than the traditional methods. It aims at the real satisfaction of the customer requirement. All these characteristics seem to be interesting to improve software development process as a main goal in this thesis. Agile development encloses a set of values and principles that clarified the main concept. It provides some insight into the how agile is being adopted within organizations as EMP. In this chapter, we illustrate the different values and principles defined by agile software development.

4.1 Introduction

This type of methodology constitutes the current state of art in software development methodologies. That means that it represents what is done best as regards software development, however that does not mean that they are the most used methodologies. These methods result from a mixture between the incremental model and the iterative model aiming to reduce the cycle of software development, thus to accelerate the development. They are very well adapted to the problems of project advance for which the specifications are not stable because of their constant implication of the customer. Moreover, they are very pragmatic and aim at the real satisfaction of the customer and not that of a contract drawn up beforehand.

“Agile software development is a conceptual framework for software engineering that promotes development iterations throughout the lifecycle of the project.” [10]. Each iteration is an entire software project: including planning, requirements analysis, design, coding, testing, and documentation. Thus, agile software development aims to develop and implement software quickly in close cooperation with the customer in an adaptive way, so that it is possible to react to changes set by changing business environment and at the same time maintains effectiveness and efficiency.

“In 2001, 17 prominent figures in the field of agile development (then called "light-weight methodologies") came together at the Snowbird ski resort in Utah to discuss ways of creating software in a lighter and faster way”. [9]. They created the Agile Manifesto, widely regarded as the canonical definition of agile development, and accompanying agile principles. Agile Manifesto consists of four values and twelve principles. We begin by explaining the background of Agile Manifesto and its creator Agile Alliance, and presenting the values and principles of Agile Manifesto in detail. We will reflect the principles of Agile Manifesto to the existing literature, and present favourable and contradicting arguments concerning the principles.

4.2 Values of Agile Manifesto

According to [6], Agile Alliance was formed when seventeen representatives of different agile methods, such as Extreme Programming (XP), Scrum and Crystal Family, met to discuss alternatives to rigorous, documentation driven software development. Thus, Agile Manifesto is a collection of values and principles, which can be found in the background of the most agile methods. Values of Agile Manifesto are the following:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

We found no explicit explanation to which exact methods “agilists” were referring to, they did not clearly point out the shortcomings of the traditional methods' in a detailed level. In addition, the concepts method and process are not clearly defined nor systematically used. Sometimes the “agilists” compare Agile Manifesto and agile methods to process models; sometimes they are compared with development methods.

3.2.1 Individuals and interactions over processes and tools

The project management established the “method/procedure” like the essential vector of success of a project. Whereas in the facts, best guaranteed success are the people. Even a good method will not save a project of the failure if the team is not composed of adequate people that implied in the project. It is preferable to have a welded team of average developers which communicate and collaborate rather than a team made up of even brilliant individualists.

Agile methods suggest that process descriptions and organization charts are required to get the project started; Agile Alliance wants to emphasize the individual people over roles and encourage interaction between individuals. Processes can be beneficial to certain extent, and a total abandonment of processes should be avoided.

4.2.2 Working software over comprehensive documentation

It is imperative to have a functional final application (it is finally the goal of the customer). The rest (as documentation,…) is secondary, although a brief and specific documentation is useful as communication medium. Moreover, the writing and the update of documents are extremely consuming resources. A document which is not put regularly update can become useless or even misleading. It is preferable to well comment the code which is the key element of documentation. It is as simpler to ask a feedback a customer on a functional product as on documentation. In addition, it is always necessary to develop the simplest way because the most complicated will surely not be used.

4.2.3 Customer collaboration over contract negotiation

The changes are predictable within a data-processing project. In fact, it is expected to think that the requirements, planning and the cost can be specified in advance and that they will not change. Based on this principle, the best manner to respond to the customer requirements is to work in close cooperation with him. That makes it possible to answer directly the changes of customer requirements and thus to obtain his satisfaction. That remains nevertheless difficult to implement because that implies a great mutual confidence. Highsmith and Cockburn suggested that "The basic assumption behind this value statement is customer satisfaction in general, which is the main driver in agile software development" [10].

4.2.4 Responding to change over following a plan

The project quality for the customer is often closely related to the adaptation capacity of the development team to the requirements of that. The requirements for a new software system will not be completely known until after the users have used it. Requirements change constantly because of uncertainty and the interactive nature of software, and because of the fluctuating business and technology environments. Those changes should be taken into account in software development. Customer requirements certainly will change in the course of project and it is essential to have a flexible planning and adaptable to these changes. These problems inquire for the customer the bravery to give priorities to its requirements and to accept that certain needs are not finally very clear. It asks for the person in charge to start a project before that everything was not clearly specified in a document.

4.3 Principles of Agile Manifesto

The four fundamental values described above are inflected in twelve principles of Agile Manifesto as demonstrates in [7] and [10]. The principles are the following:

Principle 1: "Our highest priority is to satisfy the customer through early and continuous delivery of valuable software."

The agile methods advise to deliver very early a rudimentary version then to deliver often versions to which is added the functionalities progressively. That allows a fast feedback of the customer and thus an immediate taking into account the change of its requirements. It is as well as a good also for the follow-up of the risks. In our sight, principle 1 strongly supports the analysis, design, and implementation phases as it reflects iterative software development. Early deliverables can be used when further defining requirements, architecture and programming solutions. In addition, project management is assisted, as deliverables can guide tracking the current state of the project and guide in planning the project further.

Principle 2: "Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage."

The changes of requirements must be taken as a good thing because they mean that the team understood and learned how to satisfy the customer in better way. The systems developed with agile methodologies want to be flexible and evolutionary because they must minimize the change impact. This principle can either facilitate different development phases or make them more complicated and difficult. Responding to changes should be done to satisfy the customer. However, radical changes especially in the end of development project can result in problematic situations in analysis, design, implementation and testing.

Principle 3: "Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale."

Contrary to the delivery of document describing the application, the delivery as most frequent as possible of a functional application offers to the customer a better vision of the product and a faster feedback. The feedback gained after each increment can strongly facilitate analysis, design and implementation. In addition, the feedback information can be used in testing and short increments can help in project planning as the progress of the project can be easily tracked according to the feedback.

Principle 4: "Business people and developers must work together daily throughout the project."

The interaction between the customer and the developers must be permanent to guide the project continuously. Collaboration of business people and developers can strongly support analysis phase of software development because the requirements for the software are discussed from both business and technical point of views. In addition, project management is strongly facilitated by this principle because the project can be planned and managed cooperatively to meet the business objectives.

Principle 5: "Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done."

With agile methodologies, the persons are the key factor of the success of a project. All the remainder, processes, environment, technology can be changed if they achieve the correct operation of the team. "Trusting them to get the job done" refers to the trust that managers should have towards their employees. Managers should let their employees make decisions and allow them to make errors. Thus, managers should trust their employees and give them responsibility. Lack of trust may poison the team spirit and hinder the formation of a bonded, close team.

Principle 6: "The most efficient and effective method of conveying information to and within a development team is face-to-face conversation."

Agile methodologies approve the principle that the best manner of communication is face-to-face.

It decreases the needed amount of documentation because the information and knowledge are transferred in personal discussions. Face-to-face conversation does not only refer to "chit chat" but such conversations can also take place when two people are discussing a piece of code while programming it. Face-to-face communication is the best way to transfer tacit knowledge that plays a very important role in agile methods. Face-to-face communication can assist in analysis, design and implementation as problems can be discussed and solved in rich communication mode with constant feedback. Testing, project management and job satisfaction can also be positively affected when following this principle.

Principle 7: "Working software is the primary measure of progress."

The customer has to only to know in which phase of the project the team is, which documents were carried out… The customer wants a product finished which fulfills the requirements that it fixed. The best means to measure this progression is quite simply to see indeed which percentages of the functionalities are implemented.

Principle 8: "Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely."

Agile methodologies want to remove the stress post deployment. It is necessary to avoid the stress all the time for good quality in development. Thus, it is logical to think that work cannot be done in successive sprints but rather in a regular race as in a marathon.

Principle 9: "Continuous attention to technical excellence and good design enhances agility."

The source code of the application must always remain clean to improve the speed and quality of the development. Thus, the members of the team must pay a continuous attention to the quality and the clearness of the code which they produce while carrying out of the refactoring. Making the design and implementation better and better (i.e., refactoring) will strongly facilitate design and implementation, and support analysis and testing. Iterative approach enables utilizing design and implementation as input for the analysis of the next iteration round.

Principle 10: "Simplicity – the art of maximizing the amount of work not done – is essential."

As declared by Einstein "Everything should be made as simple as possible, but not simpler." That applies very well to the software development. Indeed, it is preferable not to anticipate the customer requirements and to make the system simplest as possible, meeting the present requirements so that if necessary those are easily adaptable in the future. The requirements are hardly simple in the beginning but as the project goes on and analysis will become clearer, this principle supports the analysis phase. If the design and implementation are simple, testing is easier and more effective. This principle most strongly supports the design of high-quality architecture and implementation.

Principle 11: “The best architectures, requirements, and designs emerge from self-organizing teams.”

Who better than the team itself can find out which is most able to carry out a task within a team? Highsmith and Cockburn affirmed that: “Self-organizing teams are not leaderless teams; they are teams that can organize again and again, in various configurations, to meet challenges as they arise”. [10]

Principle 12: “At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behaviour accordingly.”

The last principle of suggests that the agility is to be reactive with the changes which are inevitable. Thus, it is clear that an agile development methodology is subjected to modifications. Methodologies of agile development define developments which are agile but they must be to themselves. It is not only the requirements that constantly change; also, the development process is changing over time.

In order to summarize, the concept of agile software development aims to reducing the software lifecycle. Thus, this type of methodology allows accelerating a software development. Starting by developing a minimal version, then it is possible to integrate the functionalities by an iterative process based on customer requirements and tests throughout the cycle of development. The origin of the agile methods is related on the instability of the technological environment. In fact, the customer is often not able to define precisely and completely his requirement at the beginning of the project. Thus, the “agile” term refers to the capacity of adaptation to the context of specific modifications occurring during the development process. For that, several agile methods are defined as Scrum, Crystal Clear, Adaptive Software Development, Feature Driven Development, Dynamic Systems Development Method (DSDM) and finally Extreme Programming (XP) which will constitute our object of the next chapter.

CHAPTER 5 - EXTREME PROGRAMMING

Agile Manifesto with its values and principles gives an "ideological" background for agile software development. Those ideas are materialized in agile methods. Extreme Programming can be considered the most known of agile methods. It is a process of software development which encloses a certain number of practices intended to organize the work of a team development. These practices are focused on the software construction, downstream from the preliminary phases of feasibility and opportunity studies. In this chapter, we introduce briefly the principle of each XP practice in order to figure out which one is suitable to be used at EMP that achieves the main goal of the thesis. Afterwards, we describe the main development cycle used in extreme programming. Finally, we estimate the most important advantages and drawbacks of this agile method.

5.1 Background

Kent Beck has defined XP in [12] "Extreme Programming (XP) is a discipline of software development based on values of simplicity, communication, feedback, and courage. It works by bringing the whole team together in the presence of simple practices, with enough feedback to enable the team to see where they are and to tune the practices to their unique situation".

Extreme Programming is a methodology suggested by Kent Beck, with the assistance of Ward Cunningham and Ron Jeffries. It is the most known of agile methods. It gathers a whole of good practices of development rising from the principles of the agility and leading them to the extreme. Initially, it aims to put together the developer and the customer in the core of the development process. It is relevant for teams of small size about a dozen people. It seems simple of use but it claims much discipline and communication.

5.2 XP Practices

It based on the four principles of agile development "communication, feedback, simplicity and courage". Consequently, it is a question of applying to the extreme the twelve or the thirteen (the tests are often gathered under only one practical) good following practices. According to Wikipedia [11], XP has 12 practices, grouped into three areas derived from the best practices of software engineering:

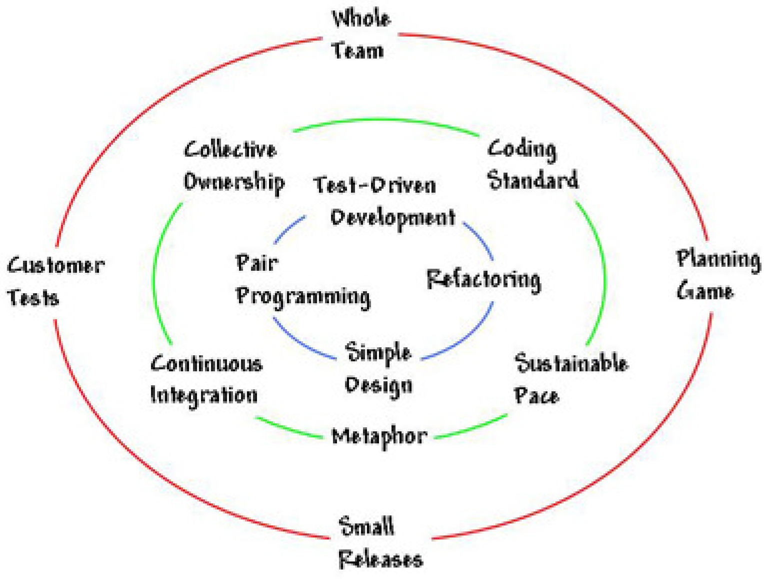


Figure 5.1: Extreme Programming practices

The development is done iteratively and phases sometimes overlap, thus it is difficult to provide a comprehensive step-by-step description of an XP project and the use of the XP practices. Next, we explain briefly the practices further but more information about their dependencies and relations can be found at [13].

1. Whole team: The developers are assisted by a dedicated person who is charged to find out the needs, to fix the priorities, to write the receipt books and to answer the questions of the programmers. This dedicated person must be on the same site as the developers.

2. Planning Game: The purpose of this practice is to plan only the releases. Planning is done in the play form in which both the developers and the customer take part. During a first phase known as exploration, the customer expresses his requirements in functionalities terms. For that, he writes them in the form of "using stories", i.e. short descriptions of the behaviour waiting from the system, expressed with a user point of view. During this same phase, the developers assign to each using story a number of points, time function that they estimate necessary to the functionalities development enclosed in each using story. If it attests that these stories requires a long development time, they are cut out in elementary scenarios.

In second phase known as an engagement, user stories is sorted according to the value which they bring to the customer and the risks incurred during their development. This allows user stories classification by set of priorities. According to the swiftness of the team, the developers and the customer intend on the user stories quantities that will be developed during the iteration. Thus, it will constitute the next release. The swiftness evoked above is an indicator of the number of points which can be developed during an iteration (it is calculated by submitting the report/ratio of the number of points developed during the preceding iteration). This practice permits to combine the customer priorities with the developer's estimation in order to be appropriate in next the release and the date on which it will have to be ready.

3. Test-driven development (TDD): Frequent testing provides feedback that is highly emphasized in XP. XP's testing practices differ considerably from methods that suggest testing in the end of the development or that prohibit programmers from testing their own code. XP's unit tests enable quick changes in rapid software development projects. Some specialists suggest writing unit tests for every feature to ensure that new features work. In addition, all unit tests ought to be run in the entire system before releasing any code to make certain that nothing else is broken. XP's "test-first development" means that programmers write test cases and actual tests before they start programming. The test results offer programmers feedback of the functionality and design of their code. It also enables making the software work straightaway. All the produced code should clear the tests at all times to ensure a functioning build. Tests are gathered together, and a test written for a particular feature should be cleared by the particular piece of code but also the main build should clear the test.

Unit tests enable fixing small problems within a short timeframe instead of fixing large problems later. Furthermore, unit tests should be automated. Automated unit tests are expected to pay off their cost of creation (mostly due to the time that they require) quite quickly in the course of project. In addition, automated tests eliminate bypassing manual tests, which easily occur in the time of stress. When presenting the desired features, the customer also defines one or more acceptance tests beforehand to examine whether the feature is working. The development team builds and runs those tests. The results again guide the team and the customer to further design and implement the software.

Kent Beck suggests in [17] "Programmers write their own tests and they write theses tests before they code. If programming is about learning, and learning is about getting lots of feedback as quickly as possible, then you can learn much from tests written by someone else days or weeks after the code".

4. Continuous integration: The system is not seen as a block but as a simple system to which small modules will be added. These modules are completely assembled and tested at several times per day. That makes more quickly the system stable and operational. The development team should always be working on the latest version of the software. Continuous integration will avoid delays later on in the project cycle, caused by integration problems.

5. Small Releases: The deliveries rhythm must be shortest as possible, that will permit to accelerate the feedback of the customer by providing him practical functionalities quickly. This acceleration of the feedback will reduce the disagreements between the customer and the developers. That will have several positive effects: speed up the development, enhance customer satisfaction because his requirements are finally understood and increase in the developer's motivation.

6. Sustainable pace: The concept is that programmers or software developers should not work more than 40-hour weeks. It is not a question to seek to work little, but the specialists in Programming extreme became aware owing to the fact that the extra work is harmful. Under the terms of this principle, a team should never be overloaded of work more than two consecutive weeks. If there is overtime one week, that the next week should not include more overtime because a tired programmer makes more errors.

7. Customer tests: Fast information feedbacks on the system (in general automated) made up starting from tests criteria defined by the customer. The presence of the customer on site is dictated by requirements as regards reactivity. Indeed, in programming, the developers frequently raise questions about obscure remained points. Thus, the customer can immediately bring answers to these questions, avoiding as well as the programmers start to develop certain functionalities on the basis of what they suppose being the customers desires.

8. Simple design: Simplicity is one of the fundamental values of XP. Simple design refers to developing only as much code as is required, adding only the planned functionality at a time. Keeping design simple is not an easy task. However, XP suggests keeping things as simple as possible as long as possible, and adding only functionality or piece of code when it is relevant. Those which practice XP summarize that under YAGNI's sentence "You ain't gonna need it". The only requirements are to satisfy all the tests, never to duplicate.

9. Metaphor: The metaphor is used to model conceptually the system under development in order to clarify the functionalities to be reached with the assistance of the vocabulary of the customer. XP recommends using metaphors to describe the system architecture. Such images allow everyone to have a global vision of the system and to understand the major elements like their interactions

10. Refactoring: The code being always in modification and not having design upstream, it must be altered constantly to be clean without modifying its behaviour. The developers must be familiar to work each day with the existing code and functioning perfectly to maintain it clean, to make it more readable and more robust. The goal of this practice is to simplify the code, while making so that all the tests are satisfied. From a purely functional point of view, this simplification is not necessary since it intercedes on code which functions perfectly. On the other hand, the refactoring of the code ensures that the addition of new functionalities will be facilitated. The refactoring tends to produce a code thought better, modular, without duplications and thus easier to maintain.

11. Coding standard: It is necessary to have standards of naming and programming. The code must follow this convention in order to be legible by all the team members. That is an aim to allow pair programming and the code partition. This is essential as the code property is collective and the programmers could change binomial regularly. In general, the conventions adopted at the time of projects are intuitive and result from natural practices at the developers.

12. Pair programming: The code is always written by two developers. That makes it possible to have a permanent second evaluation of code and thus more quality and robustness. Nevertheless, it allows also a dynamics of project, a better motivation of the developers as well as a knowledge partition. The binomials change during the project.

13. Collective code ownership: All the team is judicious to know the totality of the code. It means that any pair can improve any code in the system. Collective code ownership also enables changing the code quickly, while the changes do not have to be made by a certain individual assigned to own the code but anybody can make changes.

The basic principles of the XP methodology having been described, the following part will show the principle development cycle using XP.

5.3 XP Development Cycle

According to [14] and [15], the most important phase of the cycle in XP methodology can be represented by the following figure 5.2:

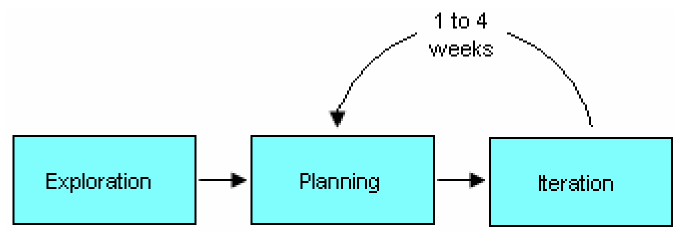


Figure 5.2: XP development cycle

Exploration: This phase allow the customer to express his requirements using the ‘User Stories’. During this phase, the developers consider technical questions intended to explore the various possibilities of the system architecture. It agrees also to make a study for example the limits on the performances level presented by each possible solution. The customer on its side is adapted to express his requirements in the form of using stories that the developers will have to estimate in term of development time.

Planning: In this phase, the developers estimate the development cost of each User Story by using a points system. Then the customer chooses the User Stories which will be developed during the next iteration according to the cost of User Stories, the swiftness (a number of feasible points during an iteration) and the value (priority) which assigns the customer to each User Story. If one User Story cannot be estimated, ‘Spike’ is created. This Spike makes it possible to define one more clearly User Story. It is during this period the customer defines the tests of acceptances.

The iteration: allow to implement User Stories. Initially the binomials are created, then the tests are coded, the code is reorganized to integrate the functionalities resulting from User Stories and finally the functionalities are coded. It is the fundamental iterative and incremental part of XP methodology. With each iteration, the planning process is started again accounting of the information turned over by the preceding iteration.

5.4 Evaluation

In this section, we enumerate the major advantages and drawbacks by using extreme programming. The following factor can be considered the main advantages:

- Agile methodology (all recompenses of this type of methodology).
- Relatively simple to implement.
- Bringing to light the technical aspects: prototypes, development types, tests…
- Innovative: pair programming …
- Very suitable for the team work (motivation).

While we suppose the following ingredient as the drawbacks of using extreme programming:

- Escape the analysis phase.
- Hard to accept by the external customers.
- Implementation slightly hard (change of mentality, adaptation).
- The refactoring is far from being a simple technique.
- Difficult to fix the prices and time measurements in advance for different functionalities because they are not available directly.
- Adapted only to the small teams.

5.5 Summary

This methodology can appear simple to set up, but it requires a change of mentality and an acceptance by the whole of the implied team (customer included). It is not obvious to apply and to control because it claims much discipline and communication. However, it is, as soon as the process is well accepted and controlled, an appreciated methodology by all the team for the improved productivity. It brings also to the teams lot of dynamics and motivation. It remains the most radical (extreme) agile method. It allows high quality of projects according to the systematic tests. It permits to stick to the customer requirements by following a several number of practices such as “test driven development” which will be the purpose of the next chapter.

CHAPTER 6 - XP PRACTICE: TEST DRIVEN DEVELOPMENT (TDD)

In data-processing programming, the unit test is a process that permits to ensure the correct function of a given part of software or a piece of program so called “unit”. Within several applications, writing of the unit tests was regarded a long time as a secondary task. However, the extreme programming (XP) method presented the unit tests, called “programmer’s tests”, in the heart of programming activity. XP method recommends writing the tests at the same time, or even before the function to be tested. This is practice is called Test-Driven Development (TDD). Our priority was to produce something suitable with the existing architecture at EMP. We try to find an improvement solution of software development that places TDD in the center of this thesis. In this chapter, we present the general rules and implications using test-driven development We describe also the TDD cycle for testing via a specific unit framework and how should be integrated in software development process.

6.1 TDD Overview

Putting in place the unit tests, within a development methodology is a strategic choice in continuous improvement objective of software quality. Existing since many years, it is the main interest carried to agile methodologies which contributed in the software development cycles. Kent Beck has suggested test-driven development (TDD) in association with Extreme Programming, one of several agile software development processes [17]. Although TDD is one of the keystones of XP, it can easily be approved to be used in any iterative software development process without adopting any of the other practices of XP. The unit tests aim to detect the greatest number of bugs in the first steps of development. Test-driven development put forward a step to help the developer to conceive clearer and simpler codes. On the other hand, it allows a better risk management and agile piloting.

6.2 TDD Rules

The TDD objective is to produce “clean code that works”. For the most part, there are two simple rules to be used [16]:

1. Developer writes the new code only when an automated test has failed
2 Any duplication of code must be eliminated.

These two rules must be strictly respected, even if they appear difficult in the first time. To understand well the TDD, it should be strictly respect the discipline imposed by the cycle describes below.

To apply these two rules, following sequential steps are used in development as TDD cycle:

- Choose a defect to fix or an area of the design or task requirements to best drive the development.
- Design a concrete test as simple as possible while driving the development as required, and check whether the test fails.
- Alter the system to satisfy the new test without affecting the other tests results.
- Possibly refactor the system to remove redundancy, without breaking any tests
- Go to first step.

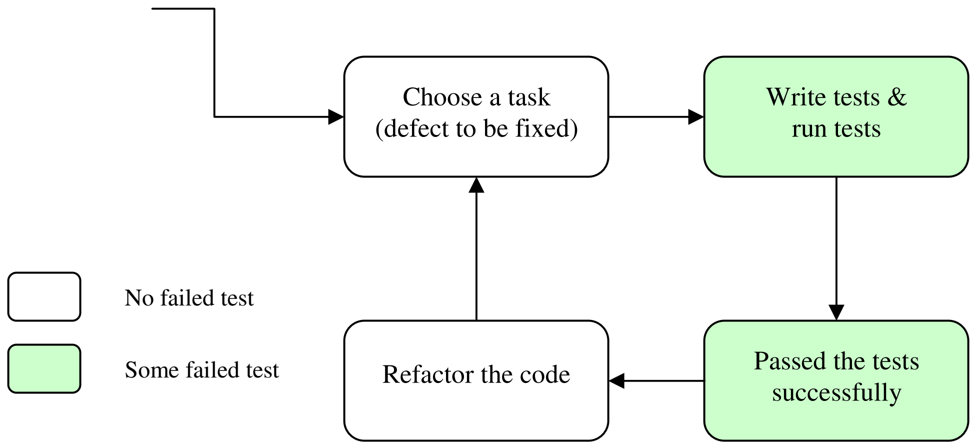


Figure 6.1: Test Driven Development steps

6.3 TDD Implications

As simple is test-driven development as it has various implications:

- We will conceive our code in an incremental method, by having always operating state code, so that this code provides the main information to make many small decisions with the current development.
- We must write our own tests, because we cannot await many times per day that another person does it
- Our development environment must provide a quick response in the small changes events.
- Our code must be composed of very coherent elements in order to make the test as simple as possible.

6.4 Test Driven Development Cycle

Kent Beck recommends in [17]: "the process of implementing each test in the test list is defined by red, green, refactor". The goal of this process is to work in small verifiable steps that provide immediate Feedback. Detailed explanations of the three TDD steps, "red", "green" and "refactor", are given below:

- ***Red*: Select a functionality to add to software or a defect to fix and write a little test that does not work, and perhaps does not even compile at first. This test is used to show whether the functionality is added successfully or not.
- **Green:** Implement the code that is only enough to make tests succeeded.
- **Refactor:** Refactoring means improving the software design without affecting its functionality. The main goal of refactoring in TDD is eliminating any duplication.

An underlying assumption of TDD is that you have a unit-testing framework available to you.[18] Agile software developers often use the xUnit family of open source tools, such as JUnit , CUnit or VBUnit. Testing frameworks based on the xUnit provide a mechanism for creating and running sets of automated test cases. The figure 6.2 below presents a UML state chart diagram for how people typically work with the xUnit tools.

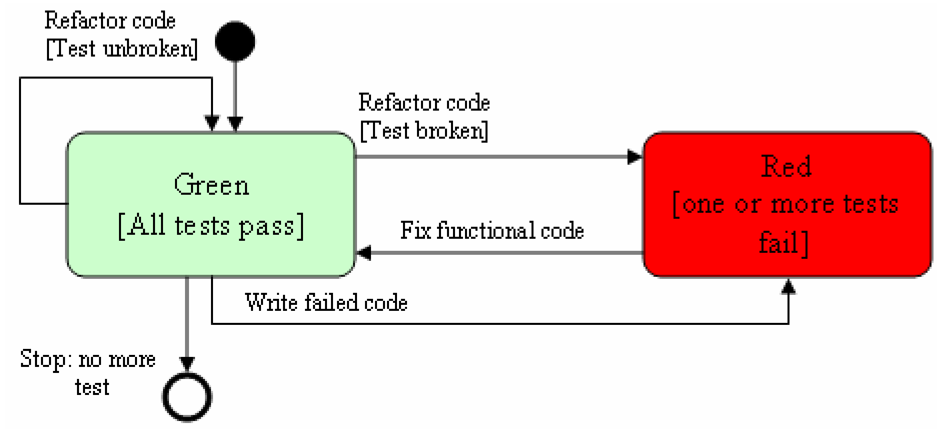


Figure 6.2: Testing via xUnit Framework

According to [17], the main cycle can be developed as follow:

1. Adding quickly a new test: by using a framework from the xUnit framework family. To start the implementation, the developer selects a test from the test list and writes a test case of each new feature.

2. Running all tests and notice the new one fail: The intention of running all tests is to authenticate the new test. The new test should also fail for the expected reason. Testing frameworks usually employ visual signs when reporting success or failure of a test. At this peak, the new test is signed in red flag.

3. Making the necessary modification: In this step, the new feature is implemented by writing code that is 'good enough' to make the test succeeded. The aim is not to write perfect code, but pass the test. The code will be improved later. This is to ensure that all tested code is written.

4. Running all tests and notice them succeed: If all test cases pass, the developer has a positive confirmation that the new feature is implemented correctly and all previously implemented tested features still work. The testing framework will show a green flag for each passed test. If a test fails, the developer must go back to the previous step and add or change some code and then run the tests again.

5. Refactoring the code to remove any duplication: The last step involves cleaning up the code. By frequently running all test cases, the developer can be confident that the code meets the same requirements before and after refactoring. The cycle is repeated until all test cases on the original test list are implemented and passed.

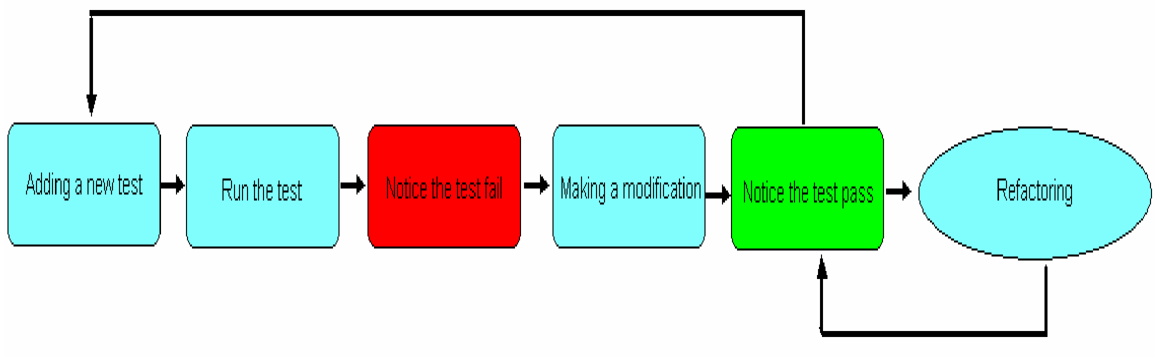


Figure 6.3: Test Driven Development Cycle

It is crucial that this cycle is carried out very quickly, in a few minutes at most. If the cycle realization takes tens of minutes, probably that you are trying to carry out an important step; it is then desirable to try to tackle a less ambitious stage. As first goal, the TDD characteristic is to recognize stages which can appear simple for developers.

CHAPTER 7 - CASE STUDY

In this chapter, we face three major parts. The first and second parts are exciting of research interesting for both software engineering and data communications technologies. Software engineering can be considered as applying theory research, which is described in the previous chapters, to the real world specific environment. This specific environment contains the main data communications technologies existing inside Ericsson Mobile Platforms (EMP). This is the second part of research that encloses the technical part of the thesis. We have to gain knowledge of Wireless Local Area Network (WLAN) as main trends in data communication technology at EMP datacom platform. Thus, in third part, we will apply what we find out in software engineering research to an embedded software development project in Wireless LAN area as a pilot project of testing.

7.1 PART 1: Software Development within EMP

In this section, we focus on software engineering part in which we will introduce the main architecture of Ericsson Mobile Platforms (EMP). We put emphasis into the factors that are related to the current software development process. First, we explain briefly the platform software inside EMP. Afterwards, we depict the current software development process and finally we show our proposed development process improvement.

7.1.1 Platform Software Structure

Based on [22], Platform Software is a part of the phone software, acting as a general piece of software, which can be configured to work in different types of phones. No dependencies exist from Platform Software to the applications. One common principle for a platform is that dependencies never go upwards. Directions like "upwards" and "downwards" always refer to the arrangements of modules and interfaces.

The Platform Software is divided into certain number of Platform Layers or Platform Groups. Any Platform Layer may be divided as well into several number of Platform Layers or a number of Platform Groups or a number of Platform Modules with associated Software Backplane. Moreover, any Platform Group may be divided into quite a lot of Platform Layers or a certain number of Platform Modules with associated Software Backplane (figure 7.1).

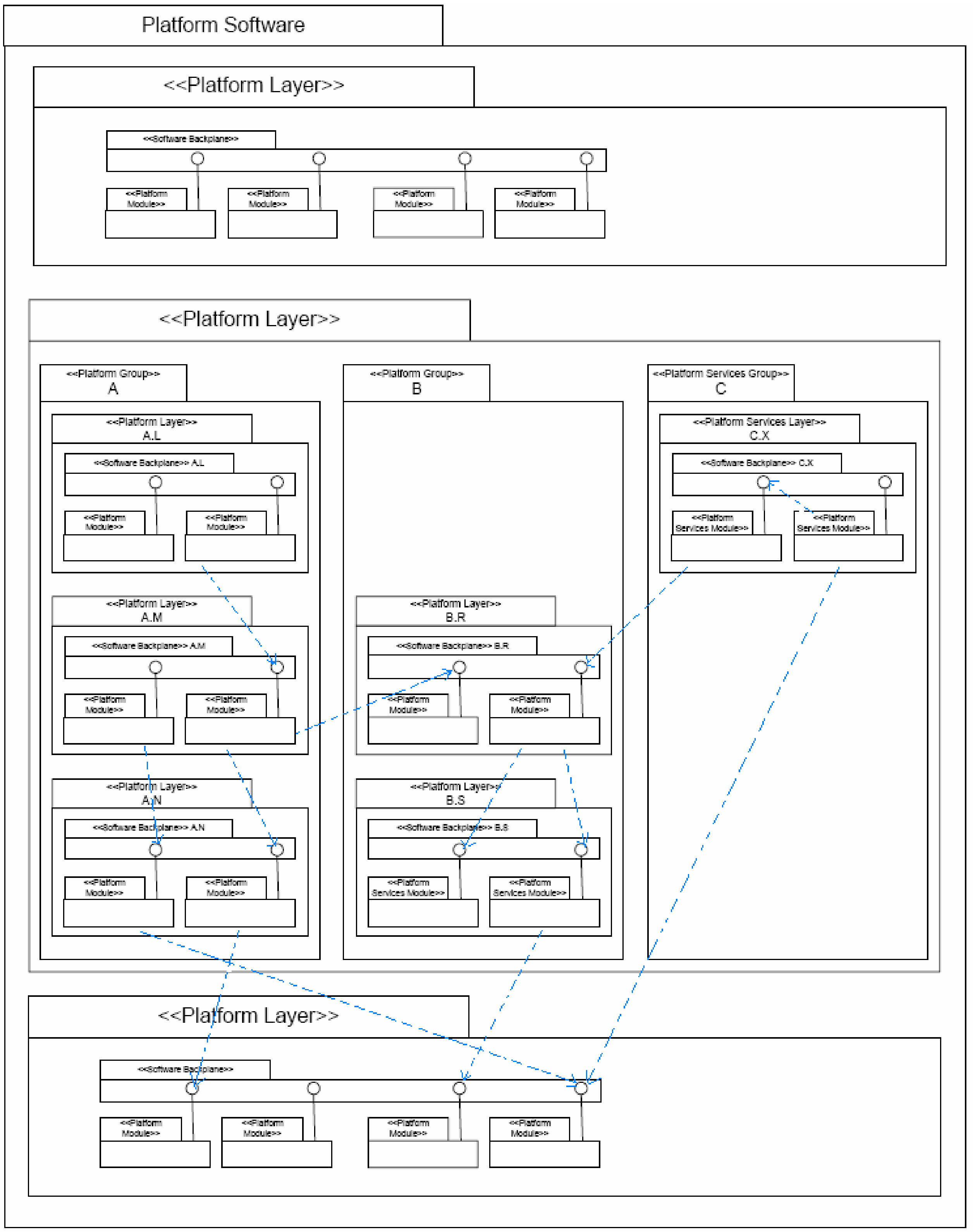

Platform Software
<<Platform Layer>>
<<Software Backplane>>
<<Platform Module>>
<<Platform Group>> A
<<Platform Group>> B
<<Platform Services Group>> C
<<Platform Layer>> A.L
<<Software Backplane>> A.L
<<Platform Layer>> A.M
<<Software Backplane>> A.M
<<Platform Layer>> A.N
<<Software Backplane>> A.N
<<Platform Layer>> B.R
<<Software Backplane>> B.R
<<Platform Layer>> B.S
<<Software Backplane>> B.S
<<Platform Services Module>>
<<Platform Services Layer>> C.X
<<Software Backplane>> C.X

Figure 7.1: General example of Platform Software Structure

7.1.1.1 Platform Group

Platform Groups are a set of functionality within Platform Software. Platform Groups are also called “stacks”. A common type of behaviour and a common type of responsibility relate software belonging to one Platform Group. A Platform Group should have a descriptive name. A Platform Group has the special property that it may not be dependent on (or should have very few dependencies to) any other Platform Group belonging to the same layer. Platform Groups can be a configuration unit, meaning that the difference between two configurations of a Platform often is the presence of Platform Groups. This places high requirements on Platform Groups to be independent of each other and to be able to detect the presence of other modules at runtime. A Platform Group may be organized into Platform Layers or into Platform Modules.

7.1.1.2 Platform Layer

A Platform Layer contains functions on the same level of abstraction. Platform Layers are organized from low to high layers, where a low layer never is dependent on a higher layer. A Platform Layer should have a descriptive name. The partition of software into layers should be guided by the intention to form different abstraction layers (or each layer can alternatively be thought of as a virtual machine providing some meaningful services to the layer above). For example, data communications layers may be physical layer, Link layer and Transport layer. Graphical user interface layers may be pixel graphics, widget graphics and window handling.

7.1.1.3 Platform Module

A Platform Module is a part of a Platform Layer or Platform Group. A Platform Module consists of a number of files, which perform closely related functions. Modules are formed so that many dependencies exist between the files and functions of a module - but few dependencies exist between the files and functions of different modules. A Platform Module may be further divided into sub-modules. A Platform Module should have a descriptive name. The interfaces of Platform Modules are organised into Function Categories. Function Categories are implemented as a number of *.h source files and are part of a Software Backplane.

7.1.1.4 Software Backplane

A Software Backplane is a module, which consists of interfaces (function declarations, type declarations and constants). Each Platform Layer containing Platform Modules has a Software Backplane, which is publishing the Platform Modules Interfaces to other Platform Modules. Platform Modules never interact without interfaces published in a Software Backplane. The software architecture styles for Software Backplanes come in two variants; Classic Software Backplane and for new development: Unified Software Backplane.

7.1.2 Software Development Process

In this section, we describe briefly the current software development process. Afterwards, we specify the main requirements desired to handle the customer requests. As a result, we propose a new process according to what we find in the previous chapters. Thus, we are appropriate to apply "test-driven development" approach in the software development process without jeopardize the existing platform.

7.1.2.1 Current Software Development Process

This is a description of a process used for development of the software which is included in a platform delivered from EMP. The process covers the execution phase of the software platform project. In order to run this process, many software development models are used as waterfall model, incremental and iterative model and evolutionary model. These models were described with detail in chapter 3.

In general, within EMP [22], the current software development process is composed of iterations. Each iteration can be considered as a small waterfall model. First of all, software requirements are specified putting the performance parameters for the new functionality. After that, detailed plans are done to assure all quality activities in different software module. This is consists of key design decisions and software architecture. In the design module, developers are interested to describe the functionality that the design is supporting in the module's perspective. Then, implementation of software code for the module according to the design is tackling. After generating the code, unit tests are created. After that, it is time to investigate what software modules that are involved in the functions to integrate. After making sure that the test environment builds, system is deployed and it is equipped to run it. If an imperfection is detected from any tests or runtime, firstly change requirements are fixed. The cycle is repeated from design module (figure 7.2).

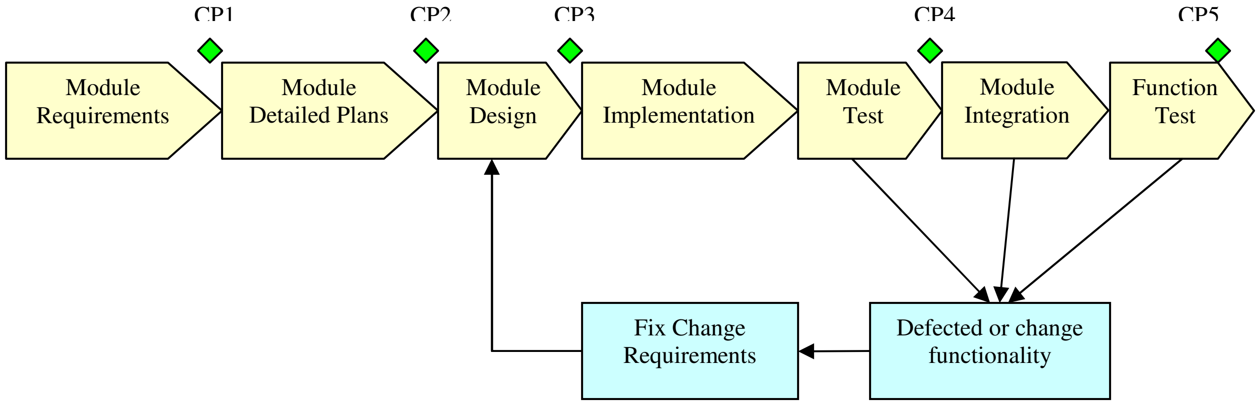


Figure 7.2: Current Software Development Process

At EMP, development process goals are stated in different levels. On a project steering level (EMP) the phases is divided into tollgates (TG) as a business decisions, which is predefined requirement which has to be fulfilled before the process is allowed to continue. Important predefined events during the project are called milestones (MS). Process checkpoint (CP) is a work status defined in the development process.

7.1.2.2 Interpretation and Requirements

According to some suggestions given by the extensive members of software development team, the current development process shown in the above figure 7.2 has a certain drawbacks. The major inconvenience of this process is the enormous time spent between decisions and the feedback obtained on their results. In addition, there is no way to be sure if the source code works successfully before the unit tests. The sources of bugs are hardly detected, because it is required to be searched in the whole code. Some developer may have the experience of spending days to discover a small bug. Moreover, same defects can arise in new software releases because the imperfections those are fixed from integration module or running function tests are not collected on unit tests. After finding a bug and making the required changes, building project over again takes too much time without being convinced that is fixed the problem. Since unit tests are not automated, it is depressed to make tests after small modification. As a result, it averts early finding of bugs and consequently it increases the time to fix a defect.

Our contribution in this thesis is to find new way avoiding these different disadvantages of the current software development process. This is must be done without jeopardizing the existing architecture of the running development projects of Ericsson. A solution that achieves fast process with high quality output and handles a customer requirement shall be proposed in an efficient way. The ambition is to improve the Ericsson platform development process by applying a new specific design that should allow us to handle new customer requests with a good mechanism.

7.1.2.3 Proposed Development Process

In our proposition, we suggest to integrate "test-driven development" in the current software development process without affecting the existing platform. Therefore, we propose locating TDD cycle to cover both of implementation and test modules. It is required that TDD cycle should be a short-term cycle. It should be repeated frequently every day. This proposed development process is shown in figure 7.3.

Prior to explain the proposed process, we have to mention that integration module must be a permanent activity. For that, we start by explaining the detailed plans and design module specifications to consider how it should be used for simple test. As seen in previous sections, module design specification output is the key design decisions for software architecture (figure 7.1). It describes the relationship between the logical entities of the architecture modules. It is recommended that Platform Modules are only dependent on Software Backplanes. One common principle for the platform software architecture is that dependencies never go upwards. Normally further restrictions apply due to logical relations in the design of a particular platform. For example, a datacom module could be expected to be independent of the interface of an Audio module.

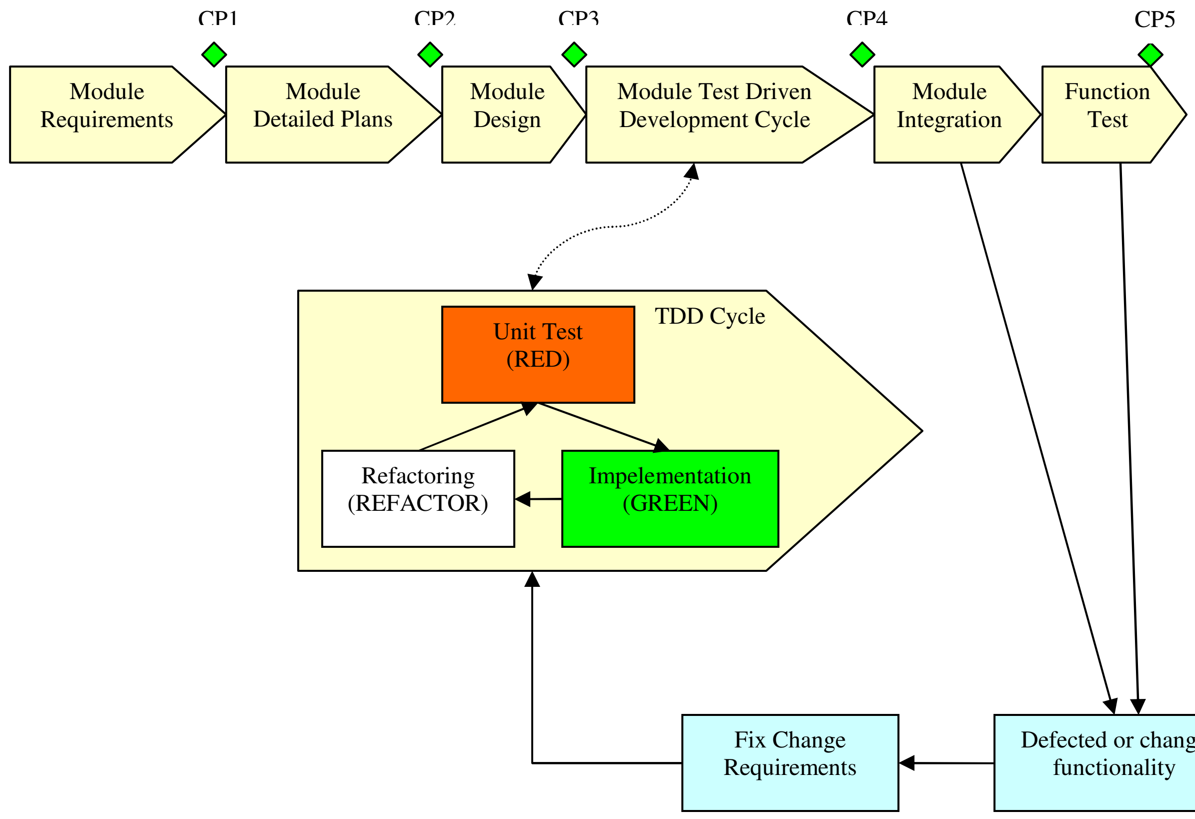


Figure 7.3: Proposed Software development process

Specific configurations of a platform may give variance in the set of modules, the set of interfaces and the behaviour of modules. It must also be possible to plug-in external modules instead of platform modules. This should be possible for a set of prepared platform modules. For isolated testing of module dependencies between Platform Groups, which are part of the same layer as shown in figure 7.1, should be avoided. This is also a design decision that is implied by testability. If dependencies can not be avoided and the Platform Groups contain layers, the layers of the different groups should be synchronized. A synchronized layer is a layer formed by layers from different groups. A proper synchronization has no dependencies upward between the synchronized layers. A Platform Module inside a Platform Layer may not be dependent on a Software Backplane belonging to a higher layer in the same Platform Group. It is recommended that a Platform Module is not dependent on the Software Backplane of the same layer.

The Software Backplane as described in section 7.1.1 is a module, which consists of interfaces (function declarations, type declarations and constants). It should be affirmed that it is not expected that the interface class of modules are completely specified after the design module, it is just predictable that the modules must be defined. The operations of these interfaces are qualified in the TDD cycle, thus we affirm that the development process is driven by tests. After establishing the architecture and the different dependencies between layers and its interfaces, it remains to tackle the TDD cycle described in chapter 6. In fact, test module is an activity that is included in implementation module within TDD cycle. For debugging system, TDD cycle consist of several sub-cycles. The first one is considered the main sub-cycle that is running as much as possible. As illustrated by Kent Beck [17], we start by adding a new test. Afterward, all tests are running and we notice that new one fail. Then, we have to make the required modification. Next, we are running again all tests and we notice them succeeded this time. Finally, we should remove any duplication by refactoring the code. The second sub-cycle is used for target environment to compile the code and to fix compiler error. In the final sub-cycle, we run the unit test in target to fix problems with debugging system. More description will be found in section 7.3.4.

7.2 PART 2: Datacom Platform (802.11 WLAN Specific Area)

In this section, we focus on the communication technology part of the project. WLAN is the specific area of datacom platform that encloses the technical part of the thesis. Thus, we will apply what we find out in part 1 to investigate a software development project in WLAN area.

7.2.1 802.11 WLAN Networks Generalities

As described in [19], 802.11 is a family of standards developed by the Institute of Electrical and Electronics Engineers (IEEE) in the 5 GHz and 2.4 GHz public spectrum bands, which specifies wireless local area network (WLAN) computer communication. Although the terms 802.11 and Wi-Fi are often used interchangeably, the Wi-Fi Alliance uses the term ‘Wi-Fi’ to define a slightly different set of overlapping standards.

The 802.11 standard defines the MAC and PHY layers for a LAN with wireless connectivity. In the original 802.11 specification, two distinct physical layers were specified in addition to the MAC layer; FHSS (Frequency-hopping Spread Spectrum) and DSSS (Direct Sequence Spread Spectrum). Since the first 802.11 specification was released, several revisions have appeared. The 802.11b specification added a high-rate direct sequence layer (HR/DSSS), and 802.11a added a physical layer based on orthogonal frequency division multiplexing (OFDM). The 802.11g specification also uses OFDM, but offers higher speeds and backwards compatibility with 802.11b [19]. The physical layer is divided into two components; a Physical Layer Convergence Procedure (PLCP) and a Physical Medium Dependent (PMD) component. The PLCP is used to transform 802.11 frames into PLCP PDU’s, with a PLCP header always sent at 1 Mbit/s so it can easily be decoded by the physical layer, and a PMD component to transmit the frames.

7.2.2 General Architecture

Wireless LANs can be broadly classified into two categories: ad hoc wireless LANs and wireless LANs with infrastructure. In ad hoc networks, several wireless nodes join together to establish a peer-to-peer communication. Each client communicates directly with the other clients within the network. Ad-hoc mode is designed such that only the clients within transmission range (within the same cell) of each other can communicate. If a client in an ad-hoc network wishes to communicate outside of the cell, a member of the cell must operate as a gateway and perform routing. They typically require no administration. Networked nodes share their resources without a central server.

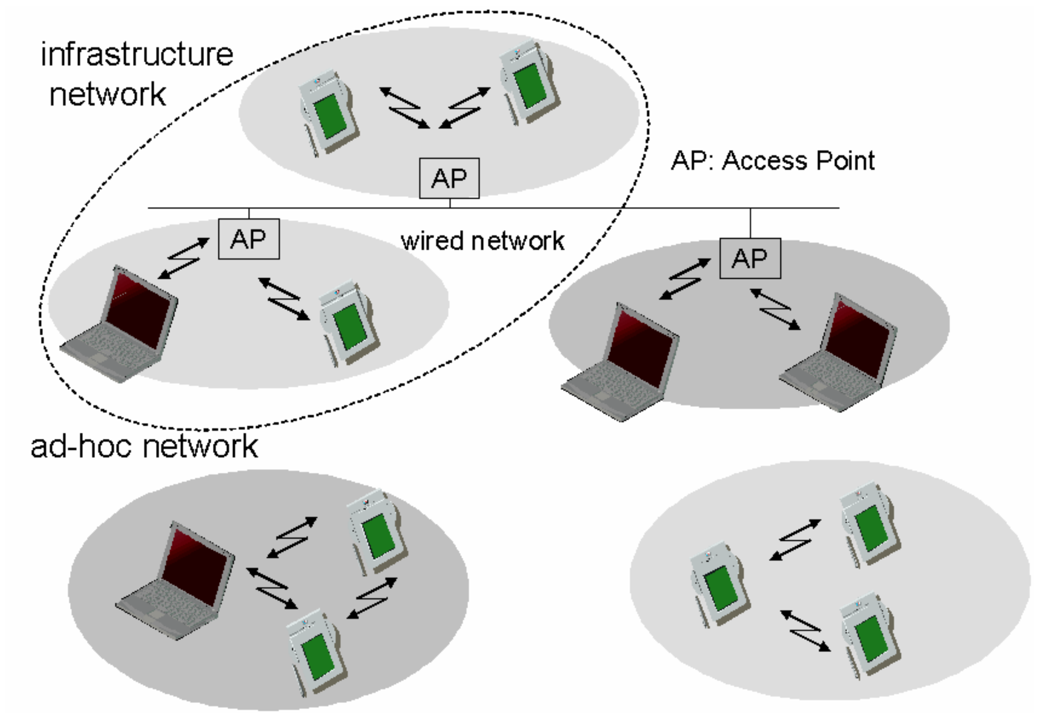


Figure 7.4: Wireless LAN architecture

In wireless LANs with infrastructure, there is a high-speed wired or wireless backbone. Wireless nodes access the wired backbone through access points. These access points allow the wireless nodes to share the available network resources efficiently. Prior to communicating data, wireless clients and access points must establish a relationship, or an association. Only after an association is established can the two wireless stations exchange data [19].

These are the main physical components in an 802.11 network:

- Stations (STA) are the most basic components. The station is terminal access mechanisms to the wireless medium and radio contact to the access point (AP).
- Basic Service Set (BSS): Group of stations using the same radio frequency in infrastructure mode.
- Independent Basic Service Set (IBSS): Group of stations using the same radio frequency in infrastructure mode.
- Access Point (AP) performs a bridging function between the wireless and the wired environment. It is a station integrated into the wireless LAN and the distribution system.
- Distribution System (DS): The wired interface of the AP is connected to the DS. Interconnection network to form one logical network (ESS: Extended Service Set) based on several BSS.

7.2.3 WLAN Functionality: Power Management

The 802.11 standard specifies a common medium access control (MAC) Layer. The MAC Layer manages communications between stations with a certain specific functionalities such as connection management, scanning, power management, roaming, security, ect… In this thesis, we are interesting only in power management functionality as a main technical part of the project. In this section, we describe in detail power management as one of the most important WLAN functionalities and which we have a technical tasks in this specific area.

7.2.3.1 Overview

According to WLAN functionality described in [22], the device is powered on and the firmware and configuration parameters are downloaded when the WLAN service is activated. After activation, the STA can be in one of two power management modes: Active mode and Power Save (PS) mode.

The power management modes apply when STA is in the following states:

- The STA is disconnected
- The STA is associated to an AP (BSS connection)
- The STA has joined an ad-hoc network (IBSS connection).

When the STA is in Active mode, it is always in the state "Awake" [22]. When the STA is in PS mode, it can be in either "Awake" or "Doze" state. The host driver will not monitor if the device is in "Awake" or "Doze" state. It is the responsibility of the device specific driver to handle transitions between the "awake" and "doze" state. There are different power save mechanisms for BSS and IBSS connections. The basic mechanism is that the beacon sender (AP or ad-hoc STA) buffer unicast packets while the receiving STA is in power save mode. When an application enables the WLAN service, the WLAN control module starts the WLAN service by turning on the device. Power off is the lowest power state for the WLAN device. The device should only be turned on when the WLAN service will be used. The device will consume power until the application powers off the WLAN service. A dedicated API is used for turning the device on and off. When in power off state there are no WLAN services available.

7.2.3.2 STA Disconnection

The STA is in disconnected state right after the device has been powered up, or after a BSS or IBSS connection has been terminated. In this state, the power save mechanism is device specific. In disconnected state the transition between awake and doze is done automatically by the device specific driver or the device firmware in order to utilize the device specific power save mechanisms. The transition from doze to awake takes some time. This is device dependent. The device will stay awake for a configurable period in case of succeeding service requests issued by the application.

7.2.3.3 BSS Connections

For BSS connections there are currently two basic power save mechanisms specified in 802.11; 802.11 Power Save (PS) and Automatic Power Save Delivery (APSD) for BSS connections. In specific cases, APSD is a more efficient power management method than 802.11 PS allowing the device to sleep for shorter intervals than the beacon period. APSD is very useful for connections with a fixed data rate, e.g. VoIP and video. The 802.11e specifies two types of APSD, unscheduled APSD (U-APSD) and scheduled APSD (S-APSD) [22].

7.2.3.3.1 802.11 Power Save (PS)

The "Listen Interval" field is used to indicate to the AP how often a STA in power save mode wakes up to listen to beacon. An AP may use the Listen Interval information in determining the lifetime of frames that it buffers for the STA. 802.11 allow two mechanisms for data retrieval when 802.11 PS is enabled; null data packet retrieval and PS-poll.

7.2.3.3.1.1 Traffic Information Map and Delivery Traffic Information Map

STAs in 802.11 PS wake up regularly to monitor the TIM packet (traffic information map), which is sent with every beacon (for unicast traffic). The TIM informs the STA that unicast data is waiting at the access point.

Delivery TIM (DTIM) is sent less frequently (every DTIM Interval) than TIM for sending buffered multicast packets. The AP specifies the DTIM interval. Buffered broadcast and multicast traffic is sent immediately after a DTIM period. Figure 7.5 shows STA1 waking up on each beacon and STA2 waking up on each DTIM.

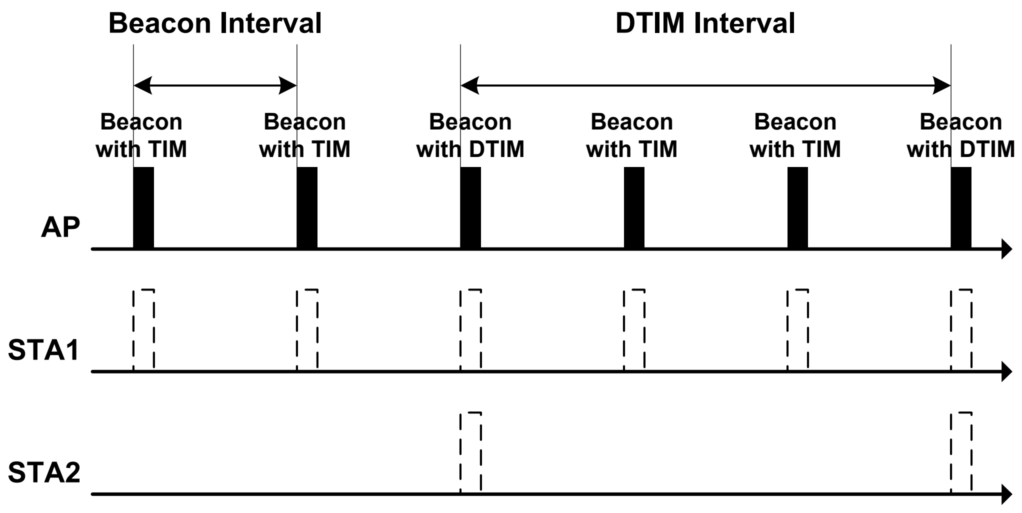


Figure 7.5: TIM and DTIM intervals.

7.2.3.3.1.2 Null packet data retrieval

The device goes from awake to doze if there is no transmit or receive traffic for a certain time set by the host driver. First, it will send a null packet with the PM bit set to inform the AP that it is asleep. Then it will enter the doze state for a certain number of beacons or DTIM intervals (listen interval), set by the host driver.

The STA wakes up to receive a beacon. If the TIM in the beacon indicates that the AP has unicast traffic buffered for the STA the device STA will send a null frame with the PM bit cleared to inform the AP that it is awake. The AP will respond by continuously transmitting the buffered data. Once data is exchanged, the station can enter doze state again. A timer controls how long to stay in "Awake" state before entering doze state, during which no traffic has been transmitted or received. If any traffic is being transmitted or received during the duration of the timer, it will be restarted after expiration.

The host can disable at any time 802.11 PS. The STA will in that case stay in the "Awake" state. Null packet data retrieval is the most efficient method for 802.11 PS when high throughput is expected and the traffic is not periodic or has a fixed speed. This method is also the most efficient power consumption wise.

7.2.3.3.1.3 Power Save Poll

The Power Save Poll mechanism is similar to Null packet data retrieval but with some limitations. Instead of sending a null frame, the STA will send a PS Poll frame to the AP to retrieve buffered data. The AP will respond with sending a single data frame. If more data frames are buffered the STA needs to send multiple PS Poll frames to get all the packets (figure 7.6).

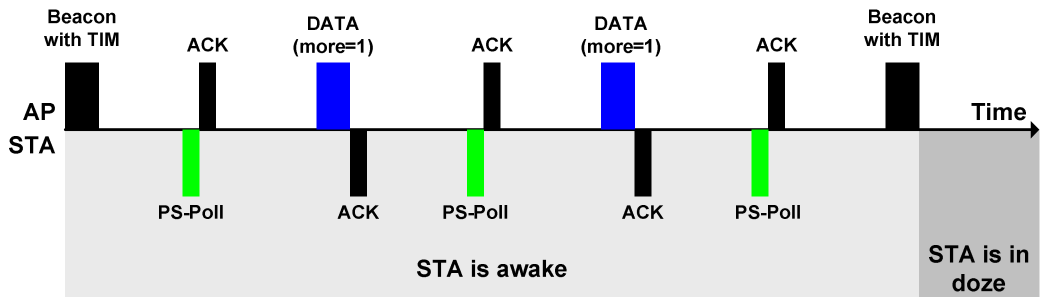


Figure 7.6: Power Save-Poll

7.2.3.3.2 Unscheduled-Automatic Power Save Delivery (U-APSD)

U-APSD is part of 802.11e. Wi-Fi™ has made the Wi-Fi Multimedia (WMM) specification, which covers U-APSD. The WLAN host driver implements U-APSD according to the WMM specification. U-APSD is similar to 802.11 power save. The main difference is that it takes into account WMM. Each WMM AC (Admission Control) can be configured with a specific U-APSD setting.

A STA can specify if an AC shall be trigger-enabled, delivery enabled or both. ACs not configured for U-APSD can use 802.11 power save, that is, frames not buffered for a delivery-enabled AC are delivered using the power save-poll mechanism.

7.2.3.3.2.1 Trigger-enabled and delivery-enabled

The STA wakes up to send a trigger data frame belonging to the trigger-enabled AC to the AP. After receiving the trigger frame, an unscheduled service period (USP or SP) is started and the AP sends frames belonging to delivery-enabled AC's to the STA. The limit of the SP is the "Maximum SP Length" given by the STA at association. The SP ends either after the amount of frames set in "Maximum SP Length" has been sent from the AP. The STA can automatically enter doze state after the SP has expired. The STA can send data frames to the AP without starting a SP at the AP, if the frames sent does not belong to a trigger enabled AC

7.2.3.3.2.2 Delivery-enabled

The STA wakes up to send a data frame belonging to the delivery-enabled AC to the AP. After receiving the data frame, a service period (SP) is started and the AP sends frames belonging to the delivery-enabled AC to the STA. The limit of the SP is the "MaxSPLength" given by the STA at association. The STA can automatically enter doze state after the SP has expired. The STA will always start a SP at the AP when sending a data frame to the AP.

7.2.3.4 IBSS Connections

The STA creating the IBSS decides whether PS is permitted not. This is done by setting the value of the ATIM window. 0 means that PS is off. If the value is non-zero PS is enabled in the IBSS. As with in infrastructure mode, an ad-hoc STA may sleep to save power. All STAs in the IBSS must monitor all frames to know when a STA has enabled power management. The transmitting STAs must then buffer frames to any sleeping STA until the STA wakes up. Ad hoc TIM (ATIM) frames are transmitted in the ATIM window by ad-hoc STAs (figure 7.7). ATIMs are used for sending buffered packets when PS is enabled in the IBSS. At the start of each beacon interval, all STAs stay Awake during the ATIM window. During this period, STAs exchange control packets to determine whether they need to stay Awake for the rest of the beacon interval [22].

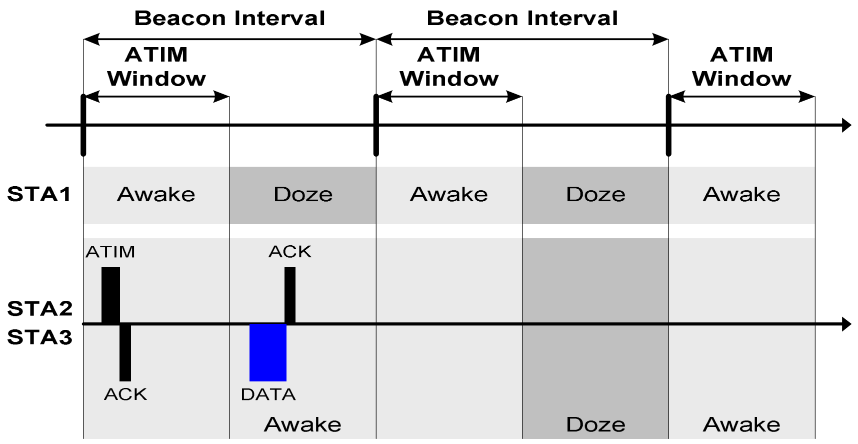


Figure 7.7: Ad-hoc Traffic Information Map (ATIM) window

If a STA does not receive an ATIM frame during an ATIM window, and has no pending packets to transmit, it may sleep during rest of the beacon interval. This is illustrated by STA1 in figure 7.7. If STA2has a packet to transmit to STA3, STA2 must send an ATIM frame to STA3 during the ATIM Window. On receipt of the ATIM frame from STA2, STA3 will reply by sending an ATIM_ACK, and stay "Awake" during the rest of the beacon interval. Data is transmitted from STA2 to STA3 after the ATIM window finishes.

7.3 PART 3: Pilot Project Development

This section represents the hard challenge in this thesis. The pilot project development can be considered as suitable testing for our new proposed way to the current software development process described in part 1. As a result, we have to face one of the original EMP projects in WLAN area as main trends in communications technologies. Our task takes aim at the center of power management functionality described in part 2. Thus, we will describe steps by steps the different modules of the pilot project lifecycle.

7.3.1 Project Management

For internal development of EMP products [22], the two processes product lifecycle management and the software development processes interact. To set up targets for EMP, the strategy development and deployment of the customer requirement (CR) process is used. The CR process is used when a customer requests new functionality compared to what is offered in the platform. The customer projects are handling the customer contacts and filing the CR to the helpdesk team.

The first line of the team ensures that all CRs arriving from customers are assigned each working day to the relevant teams in FIDO, as incident management tool that allows managing incidents into tasks in all phases of product development and maintenance management. Each CR team member should check FIDO for new CRs each morning, and perform initial investigation. The investigation effort should be limited to maximum 1 hour per team member per working day.

Our contribution in this project is to study new customer requirement CR in datacom platform. We have to add two new functionalities in WLAN services according to the requirement specifications outlined below. This could be done in the new way proposed in section 7.1.3 without jeopardizing the existing architecture of the running development projects of Ericsson.

7.3.2 Requirement Specifications

The main principle behind the power save mechanisms in connected state is that the device enters low power mode between transfers. To wake up from low power mode, the device can use a timer. The other alternative is that the host triggers the wake-up. If a timer is used to trigger wake-up, the device will notify the host that it is awake and ready for data transfer. The host can start data transfer and continue for a certain time, number of packets or other criteria before the device re-enters low power mode.

If the host triggers the wake-up, the penalty may be increased power consumption or latency because the device has to go through a certain procedure to wake up. The supported WLAN devices will support different methods for handling device and host wake-up. As result, many changes should be made in the different functionalities of power management.

7.3.2.1 Optimizations for Periodic Traffic using Power Save-Poll

A proprietary optimization of 802.11 power save can be used for periodic traffic in the case when U-APSD is not supported by the AP. 802.11 power save will not work for periodic traffic with periodicity less than a beacon period. For optimal power save, a timer is set in the device that will trigger periodical wake-up to transmit and receive packets using power save-poll. This wake-up runs separately from the wake-up configured through the "Listen Interval". Typically, the wake-up interval is set to the interval at which packets are generated and sent from the application. After exchanging all buffered packets the device will immediately enter doze, that is, the More Data bit in the last received and transmitted data packet is cleared.

The wake-up interval is controlled by a timer, "Local Wake-up Interval", implemented in the device. After association the "Local Wake-up Interval" will be set to 0, that is, normal 802.11 power save with null packet data retrieval is enabled. A client can set "Local Wake-up Interval" > 0 typically after having created a TCP or UDP socket to be used for periodic traffic. The value of the "Local Wake-up Interval" should be set according to the periodicity of the data traffic. Unlike WMM/U-APSD connections, the "Local Wake-up Interval" for 802.11 power save will affect all traffic.

The following figure 7.8 shows that the STA waking up when the "Local Wakeup Interval" timer has expired. The host is notified by the device to indicate that it is awake. One packet is buffered by the host for transmission and it is sent immediately to the device. The device will use PS-Poll to receive all packets buffered in the AP.

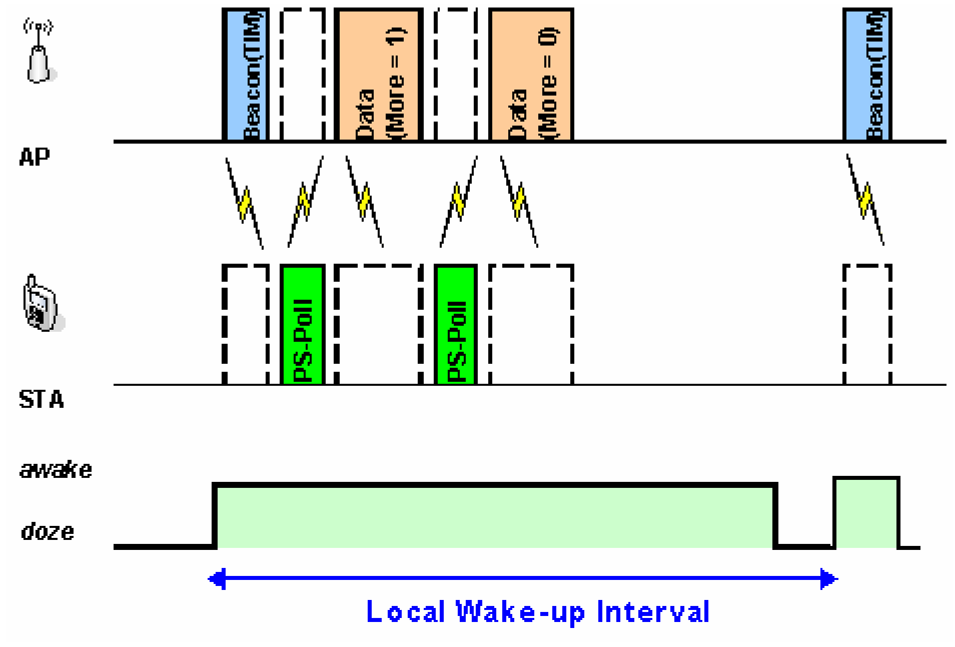


Figure 7.8: Power Save-Poll and Local Wake-up Interval

7.3.2.2 U-APSD with Local Wake-up Interval

Setting the Local Wake-up Interval > 0 will periodically start a SP. Typically the "Local Wake-up Interval" should correspond to the periodicity of the traffic generated by the voice or video application as shown in figure 7.9.

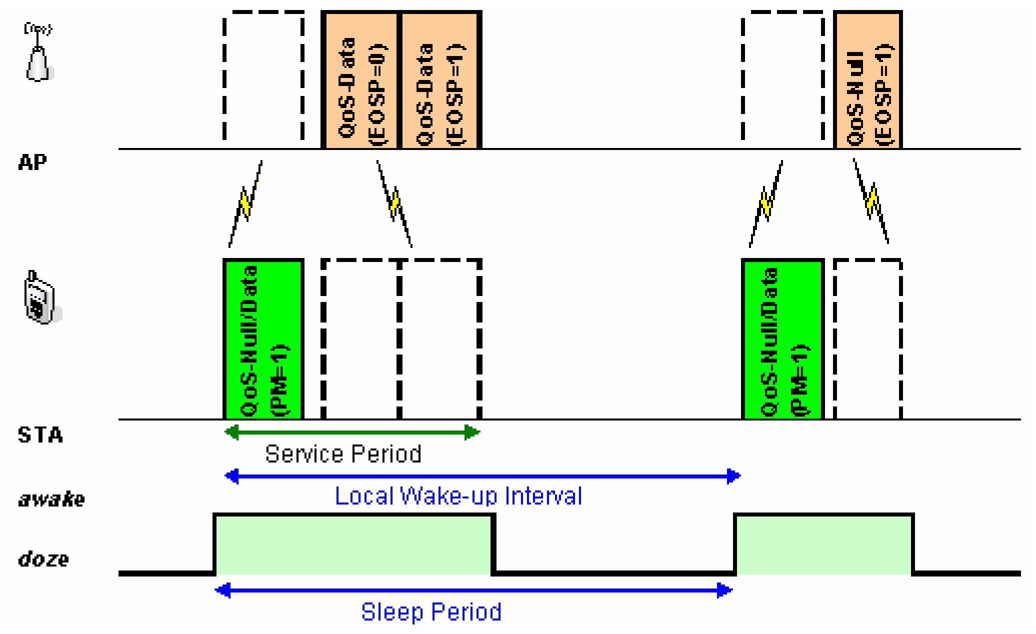


Figure 7.9: U-APSD with Local Wake-up Interval > 0

Note that only one instance of the "Local Wake-up Interval" is used. This means that it affects all ACs.

If U-APSD is enabled for the connection and no periodic traffic in progress, the "Local Wake-up Interval" should be set to zero to enable a 802.11 power save like behaviour (figure 7.10). In this mode of operation, the STA will wake up according to the "Listen Interval". If the TIM indicates buffered traffic a QoS-Null frame will be sent to a trigger enabled AC in order to trigger the delivery of all frames buffered at the AP. The STA will enter doze when "More Data" bit is cleared.

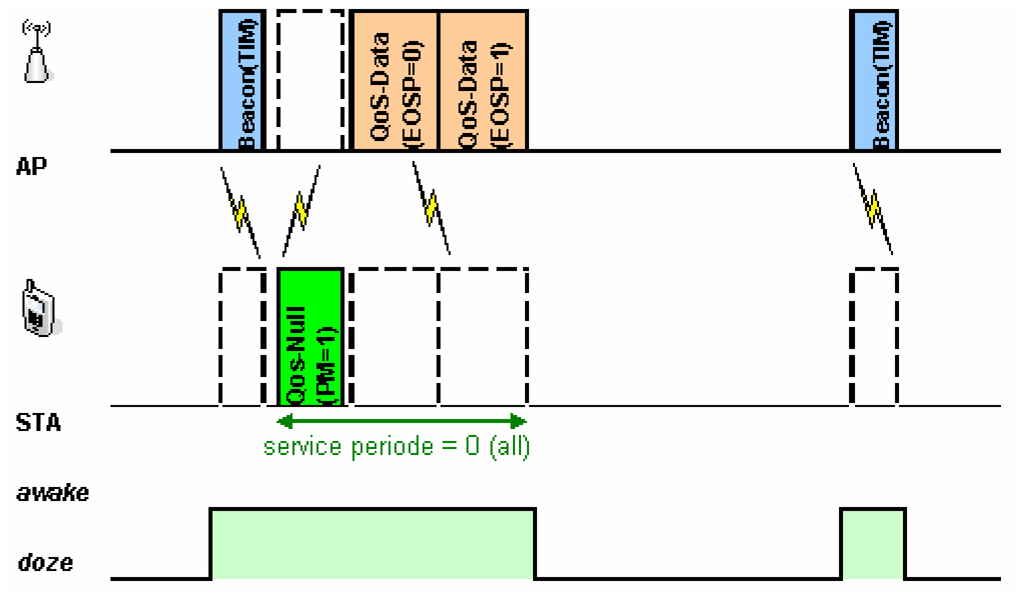


Figure 7.10: U-APSD with Local Wake-up Interval = 0

7.3.3 Module Design

We are tackling to handle new customer requirement in WLAN services that is related to power management and especially to the power save mechanism. We know from 2.2 that both in infrastructure mode and an ad-hoc mode, the STA may sleep to save power. It exist an explicit parameter co called "sleep period" that specifies the interval between consecutive wakeup instances of the device (ms).

The parameter is used to set the interval between sending of a trigger frame for enabling downlink packet transmission in U-APSD mode. This is necessary to keep the downlink traffic going when there is no uplink traffic going for a period > sleep period.

Typically, this is less than the beacon interval. Current valid range for this is [10, 60], but a change request has been filed to increase this to [10, 100]. Thus, we need to set the sleep period >80ms. A value of 0 indicates that the sleep period is disabled. In this case, sleep period will be disabled and the device is not running U-APSD.

The only way to set/alter the parameter "sleep period" is in the "TSpec" parameters. The parameter 'Minimum service interval' in the "TSpec" is used for this. There are two problems with this way of doing it;

- The "Minimum service interval" is to be used when we implement Scheduled APSD, and should be used for this purpose only.
- There is no way to set/alter the value of this parameter in cases where we want to enable UAPSD but not Admission Control.

As a result, these changes should be made as reported by the software platform structure described in the figure 7.1. Thus, two new methods of the WLAN local device interface need to be added in OPA interface. In Software Backplane area, we need to implement two new request functions to get/set the local wakeup interval. Finally, in the WLAN logical driver, we should remove functionality where we use the "Minimum service interval" value to set the Sleep Period. Then, we put in new functionality to receive the new parameter and ship the crucial commands accordingly.

7.3.4 Module Implementation

In this section, we present the way of implementing the new functionalities. The figure 7.3 shows the structure that we proceed to carry out the implementation. Mainly, our approach has the challenge to exploit the test-driven development cycle.

First of all, we start by defining the implementation language and tools. In practice, there are several ways of developing software surrounded by a large organization. Within EMP, software is either developed targeted towards an embedded device (for instance an ARM embedded processor or a specialized ASIC), or targeted towards a PC environment. Originally, we planned to use C language from the beginning as a supported language inside EMP. The natural choice of Integrated Development Environment (IDE) was Eclipse [21]. It is a well-known and widely used open source IDE. Apart from programming languages and IDE, version control is essential for all development as there is a huge amount of versions, branches and variants for the software. This also affects test cases, which need a similar version control system. Version control is implemented using "ClearCase" and an additional configuration layer, called CME.

After defining the tools of development, the implementation phase of the software development process starts. Thus, the module implementation shall be made and exported the interfaces as described in the module interface. It is of great importance that the implementation follows the design exactly as described in sections 7.3.2 and 7.3.3.

Then, we have to formulate a plan for how to administrate the software implementation for the entire software project. The plan shall be detailed enough to define what branches to use when implementing the functions in which the module is involved. The plan contains information on the merges of the various branches and is accordingly the document that determines the order of fully implemented functions in the platform. Figure 7.11 show the software modules within Ericsson Mobile Platform layered architecture.

The architecture illustrate that some parts of the new functionalities have to be implemented in a higher layer (WLAN Control) while the other parts can be implemented in a lower layer (WLAN Logical Driver). Each horizontal line corresponds to one or more APIs. WLAN Control implements the connection handler state machine of different entities. A general strategy in the platform software architecture (section 7.1.1) is to separate interfaces from implementation. This separating strategy supports the idea that interfaces are the "contract" between the interacting modules. The interface should be fixed as early as possible in the development work and can not be changed without negotiations between the developers of the using software and the implementing software. This strategy is implemented in platform domain by the Software Backplane idea (section 7.1.1.4).

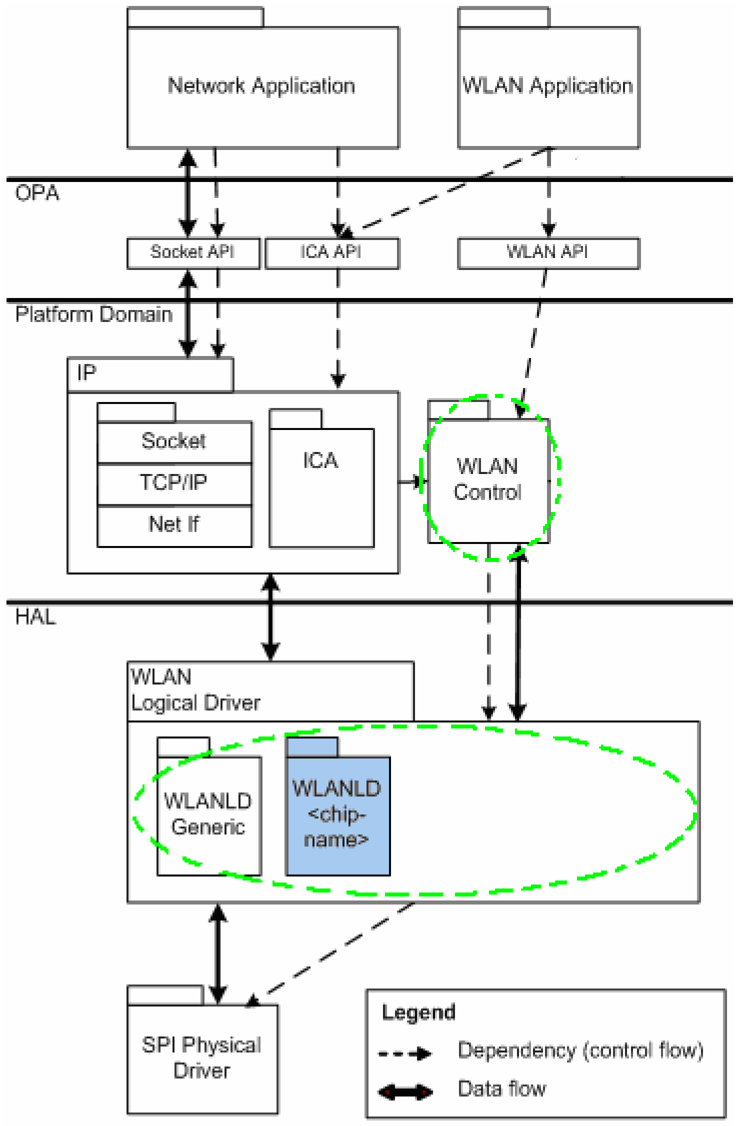


Figure 7.11: Ericsson Host Driver Architecture

In this phase, we are supposed to follow the new proposed software development process (figure 7.3). Thus, test-driven development approach shall be made in the implementation phase. Within an iterative process, we gain knowledge that before tackling programming, we define a certain number of iterations that can exist between different types of interfaces supported by Software Backplane and Hardware Abstraction Layer. Interfaces of Platform Modules are always organised in Function Categories. Function Categories are implemented as a number of *.h-files, which are part of a Software Backplane.

The types of interfaces, which software backplanes support, are Global Services, Event Channels, Utility Functions and Simple Services. Global Services, Utility Functions and Simple Services can be used if the client belongs to a layer above the server. In these cases, the in-parameters form a data flow downwards and the out-parameters form a data flow upwards in the architecture as described in the following figure 7.12.

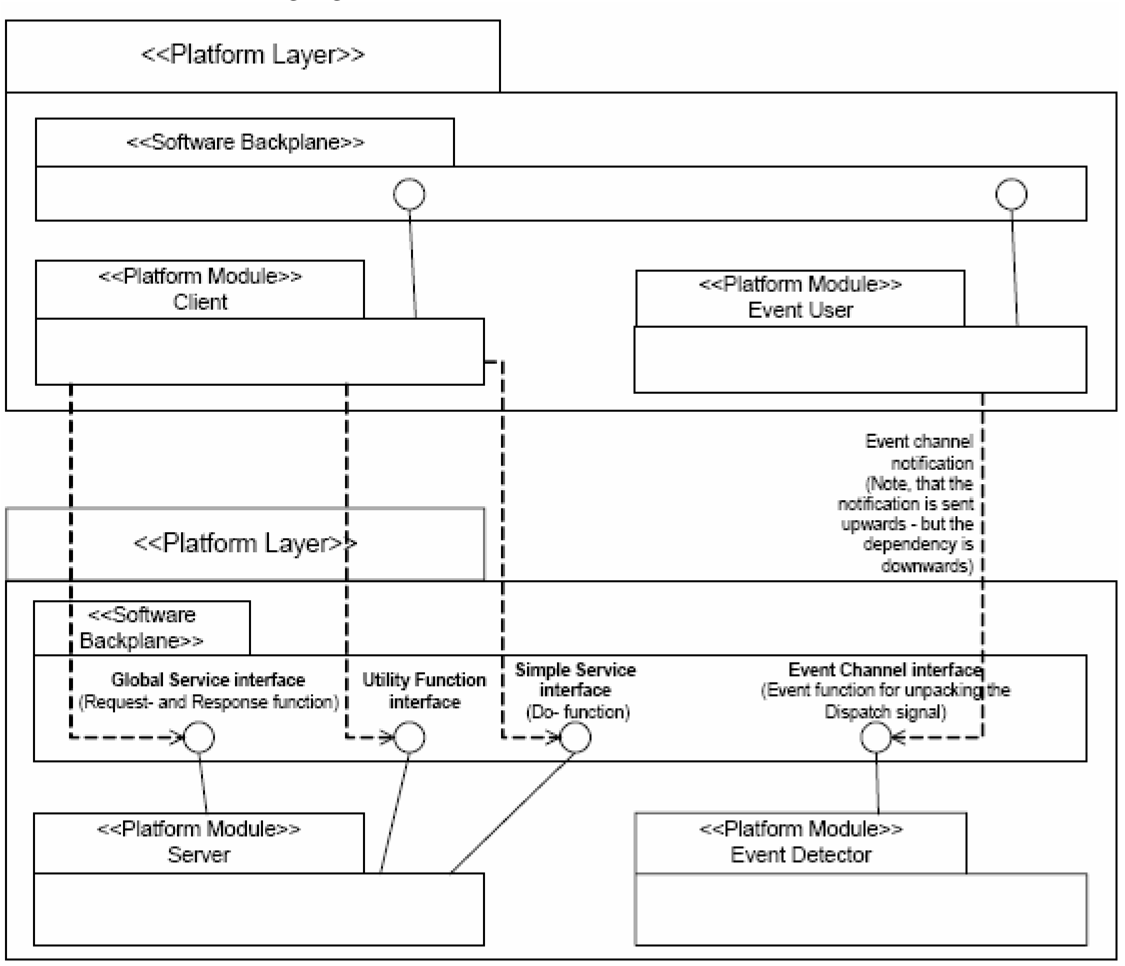

<<Platform Layer>>
<<Software Backplane>>
<<Platform Module>> Client
<<Platform Module>> Event User
Event channel notification (Note, that the notification is sent upwards - but the dependency is downwards)
<<Platform Layer>>
<<Software Backplane>>
Global Service interface (Request- and Response function)
Utility Function interface
Simple Service interface (Do- function)
Event Channel interface (Event function for unpacking the Dispatch signal)
<<Platform Module>> Server
<<Platform Module>> Event Detector

Figure 7.12: Software Backplane and Platform Module Architecture

Here, we have to implement request and response functions of "Local Wakeup Interval" in Global Services. If Global Services with Events are used, the Event data form a data flow upward. Event Channels can be used if the event detector belongs to a layer below the Event User (the subscriber). In these cases, the Notification (the event data) forms a data flow upwards in the architecture. (Note however, that dependency is (as always) downwards for a notification).

According to the description of required functionalities that the design must to be support in the module's perspective, we can reformulate at least three main iterations to be implemented in each category functions as described in the following figure 7.13. In total, we have about twenty fours main iterations that should be developed in the whole platform architecture for the "Local Wakeup interval" functionality.

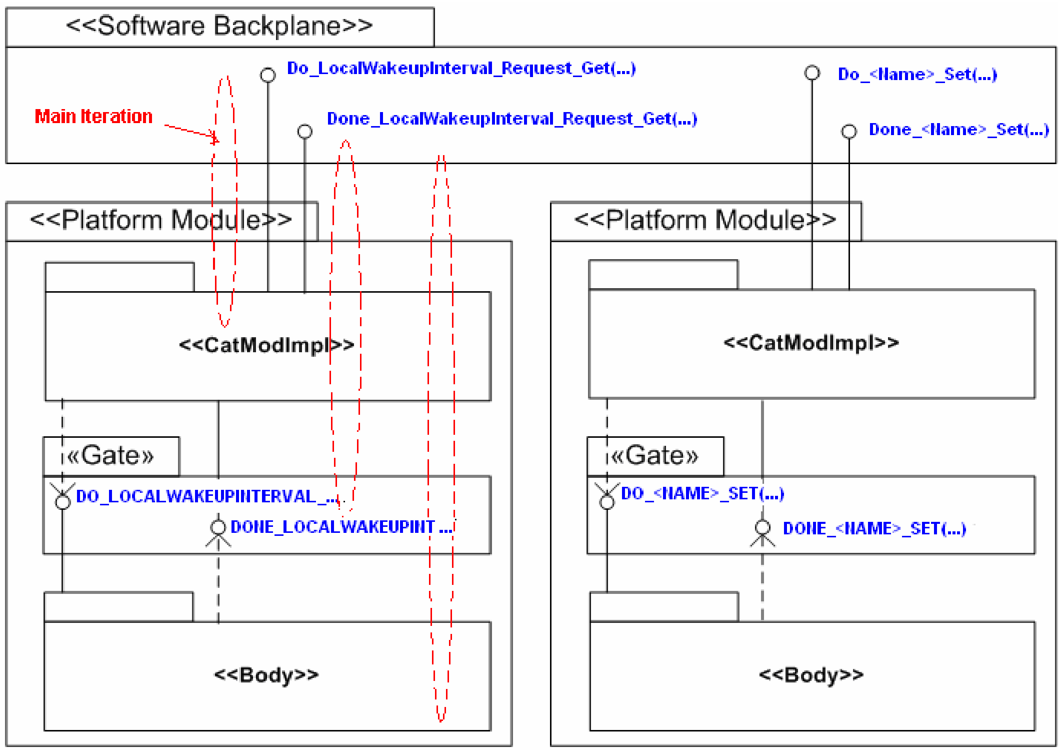


Figure 7.13: Simple Service Interface Implementation Overview with Iterative Process

After identifying the main iterations, it is coming now to tackle the test-driven development (TDD) cycle. This is the core of the module implementation phase. We will implement the software code for device drivers needed for the test program according to the module design specification. We have to mention here that our implementation will be done in two different ways. Those iterations, which are related to WLAN Control, are done with the new proposed software development process by applying TDD approach, while the others are implemented with the current process of development in order to evaluate and to find the major difference between these two mechanisms. We will describe that in detail in the next sections. We are interesting now to describe the new manner of implementation using test-driven development.

The most important of the TDD features is a unit test framework. This unit test framework must have some properties to respond to the challenges of TDD and the major conditions of TDD described in chapter 6. Putting TDD into practice does not have to be discouraging. Many resources exist to help the developer. The tests used in TDD are automated unit tests, designed to validate that a unit or module of code is working properly. To support the creation of these automated tests, a unit test-writing tool exists for almost every programming language. As we planned to use C language as a main developing language within EMP, we used CUnit framework in our implementation module. CUnit like framework is enough to write test methods fast enough. As expressed in [20], "CUnit is a lightweight system for writing, administering, and running unit tests in C. It provides C programmers a basic testing functionality with a flexible variety of user interfaces." More detail about CUnit framework about interfaces, functions, methods can be found in [20].

Then, this unit test framework provides reusable test of different Local Wakeup Interval functionalities which is easier to use. It enables an automatic execution for regression tests. Each method is tested while developed. For that, we create a list of tests that should be compiled according the mains iterations mentioned before. We start with simplest that works and incrementally we add code while testing. According to Brian Nielsen, the generic basic use of the unit test framework can be summarized in the next figure 7.14:

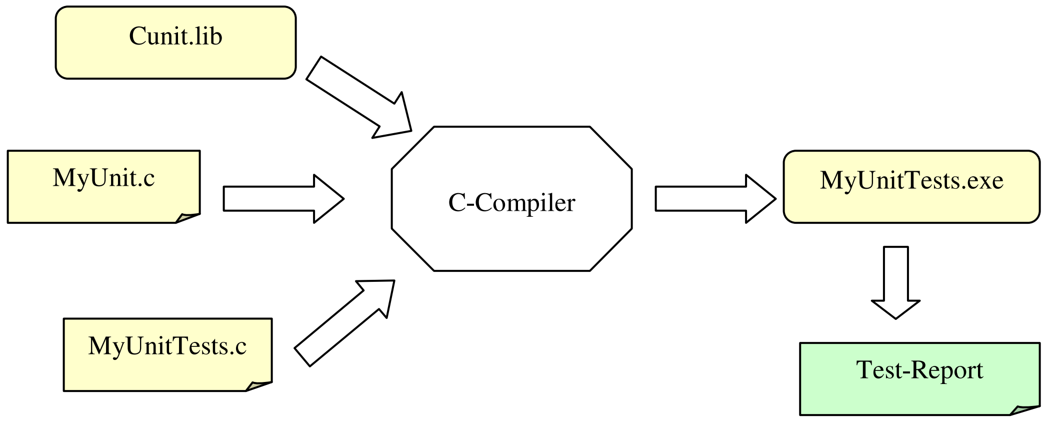


Figure 7.14: Basic Use of Unit Test Framework

After the list has been compiled, we implement the test functions and running the test using a 'TestRunner". For each program modification, all tests must be passed before the modification is regarded as complete. To do that, by following the TDD cycle, we write a test case and we check it fails, after that, we write a new code to check finally that the test passes. We can group multiple Test Cases by using Test Suite. The core parts of the unit test framework can be recapitulated in this figure 7.15:

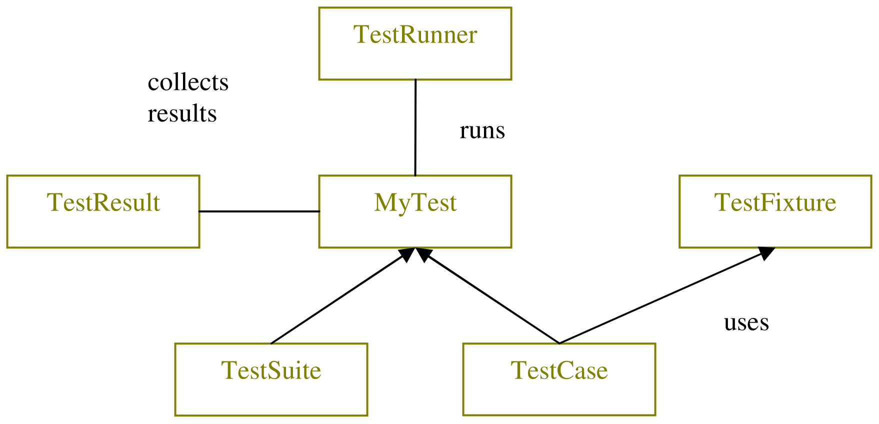
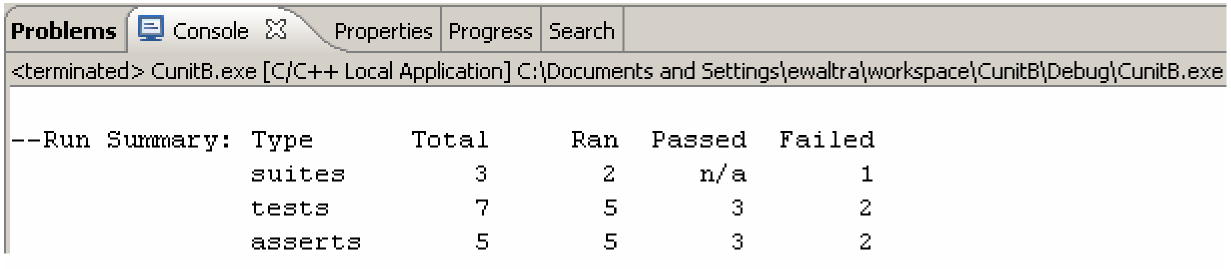


Figure 7.15: Core parts of Unit Test Framework

First, we write a test case to exercise a single software component. A Test Case is a composition of concrete test procedures which may contain several assertions and test for several test objectives. For example, all test of a particular function like Request_WLAN_LocalWakeup Interval_Get. Thus, we define one or more public test_Request_WLAN_Local….() methods that keep fit the object over test and assert expected results. Next, we add a Test Suite that contains a collection of Test Cases which is registered with the test registry. The test registry is the repository for suites and associated tests. Finally, either we have to run all tests in all registered suites or individuals test or suites can also be run. During each run, the framework keeps track of the numbers of suites, tests, and assertions run, passed and failed. Figure 7.16 shows an example of how many tests were run, any errors or failures, and a simple completion status.

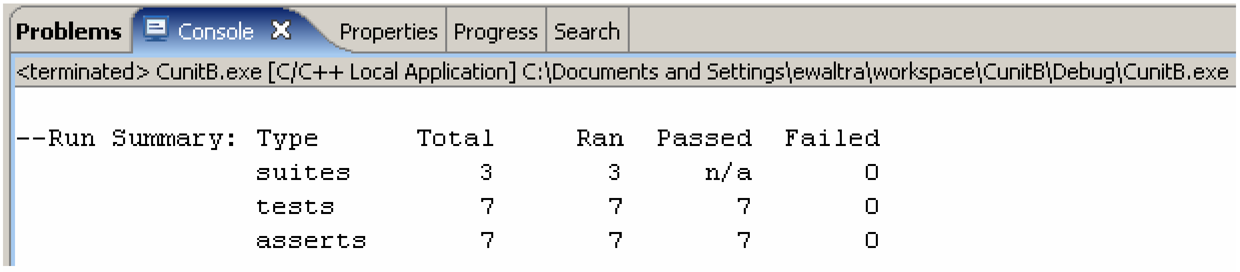


```
<terminated> CunitB.exe [C/C++ Local Application] C:\Documents and Settings\ewaltra\workspace\CunitB\Debug\CunitB.exe

--Run Summary: Type      Total     Ran  Passed  Failed
               suites        3       2     n/a       1
               tests         7       5       3       2
               asserts       5       5       3       2
```

Figure 7.16: Example of console interface with some tests and suites failures

After specifying the main problems of failure, in the next step we write some code that will pass the test. This code is not necessary to be perfect, the most important that is only designed to pass the test. As soon as we pass all test cases, we can be certain that the code gathers all the tested requirements. It is an indication to start a new cycle. Figure 7.17 shows that all Test Cases passed successfully after having fixed the errors and failures in the previous example shown in figure 7.16.



```
<terminated> CunitB.exe [C/C++ Local Application] C:\Documents and Settings\ewaltra\workspace\CunitB\Debug\CunitB.exe

--Run Summary: Type      Total     Ran  Passed  Failed
               suites        3       3     n/a       0
               tests         7       7       7       0
               asserts       7       7       7       0
```

Figure 7.17: Example of console interface with successful tests and suites

We can resume that the implementation restarts with implementation of all interface functions as stubs, then it must be possible to build the complete system before the implementation continues. The implementation and the module tests are executed in parallel i.e. tests are executed on the parts that are implemented at the same time as the implementation continues.

7.3.5 Integration and Testing

After implementing and developing the software code, we introduce the integration and testing phase in this section. The integration is made on a number of integration branches as described in the project plan. Each of these integration branches is tested against the system test specification, only good test cases need to be tested on these branches. When an error is encountered in the Integration stage, the corresponding module test description shall be updated with a test that traps the bug. Thus, we investigate what software modules that are involved in the functions to integrate. We create an environment for the software module test to make sure that the test environment builds. The application can be built for different targets, either for the target hardware or for the simulation environment. The platform includes functionality for managing the connection establishment and applications in the Win32 or target hardware environment. This functionality promotes a uniform approach when developing and debugging applications in the Win32 and target hardware environments. Figure 7.18 shows the different targets of an application build.

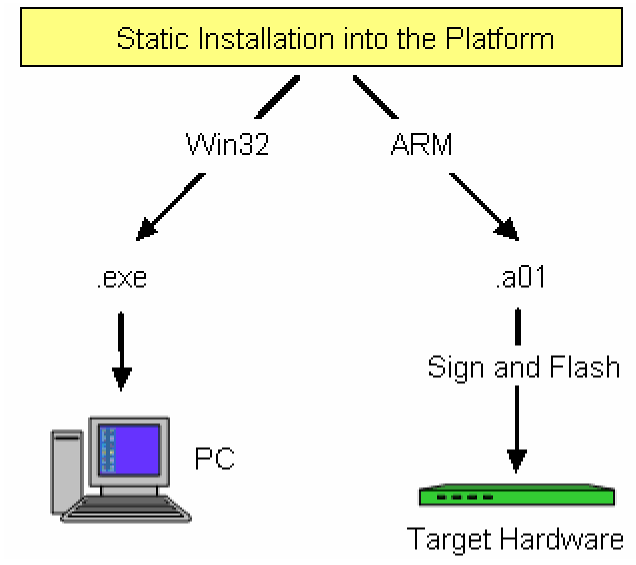
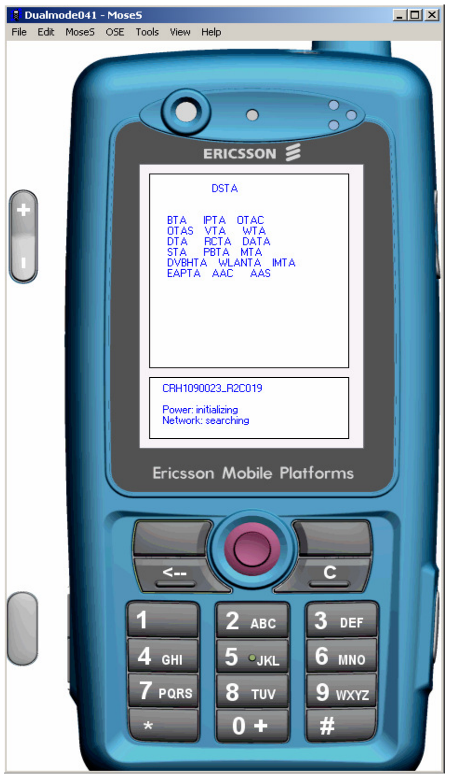


Figure 7.18: Different targets of the application build.

In the Win32 environment, the target-behaviour is simulated through a PC executable called MoseS. MoseS contains the entire platform functionality and runs the actual platform code to simulate asynchronous target-behaviour. This simulation environment includes, for example, a graphical simulation window, logging (through the 'msLog' tool) and control/simulation of events (through scripting).

In order to test the Local Wakeup Interval functionalities that we have implemented, we start our testing with the simulation environment. When our build process has been run successfully, executable binary files are produced. The target was first software simulator, thus, the MoseS program can be run directly in windows (figure 7.19).



```
        DSTA

BTA   IPTA  OTAC
OTAS  VTA   WTA
DTA   RCTA  DATA
STA   PBTA  MTA
DVBHTA  WLANTA  IMTA
EAPTA  AAC   AAS
```

```
CRH1090023_R2C019

Power: initializing
Network: searching
```

Figure 7.19: Generic Moses Window

When we compile the platform for simulation on PC, the operating system and hardware will be simulated as well. Thus, the result of the compilation is a Win32 executable that simulates asynchronous target-behaviour by running the actual platform code on the Microsoft Windows NT4/2000/XP Professional operating system. This simulation environment is called MoseS. However, MoseS is used not only for simulation, but also for debugging software and for testing.

After successful testing on software simulator, we have to test our implementation on target hardware as well. Here, the binaries have to be downloaded to the hardware target, and we used the Platform Assistant program.

For the target hardware environment, the build of a static installation into the platform results a binary executable program (often called a monolith). The monolith can be built to execute from RAM and/or Flash memory based on the configuration options specified at build time.

Just like in the simulator case above, the application software entities can be executed and controlled, and their behaviour can be observed through the User Interface mechanisms defined by the application software and/or through the platform manager (providing application management operations).

Several test cases are tested for WLAN power save functionalities and defined with Andreas after implementing "Local wake-up Interval" functionality [22]. The next chapter will deal with discussion and evaluation the performance within theoretical and technical implications.

CHAPTER 8 - DISCUSSION AND IMPLICATIONS

In order to evaluate the performance of our new proposition for software development process improvement, we discuss in this chapter the theoretical and technical implications of using test-driven development. We suggest a performance analysis based on [23]. By the way, we will describe what benefits and improvement can be approved within software development process. We express as well the limits of the thesis.

8.1 Theoretical Implications

As described before, test-driven development as powerful synergy emerging between coding and testing is one of the test-first programming concepts of extreme programming. XP is a form of agile software development which is based on four main values and twelve principles. Hopefully, almost of these values and principles described in chapter 4 are used inside EMP organization. Thus, it remains to evaluate the theoretical implications of XP values. In this section and according to the twenty weeks of experience, we illustrate the theoretical rapports by using TDD as one of the XP practices in our project.

During the project, we have been worked within software system design group. I occupied the position of software developer. We have making a strategy to be used during our work. We will have to fix regularly meetings once a week, and sometimes more than once depending on our requirement. During the meetings, we discussed with each other about how we should solve problem, how we should divide our work into parts and what parts that should be finished until the next meetings. This is approved the communication as a main XP value. Thus, building software requires communicating system requirements to the developers. The absence of communication is certainly one of the most serious defects which paralyze a project. Using TDD tends to make our communication ubiquitous between all members: between developers and managers (tests, estimates) and between developers and customers (tests, specifications). The intention of this practice, which forces to communicate, enables each one to share the useful information.

Moreover, TDD gives evidence of simplicity. This value reposes that is less expensive to develop a simple system leaves to have new expenses by adding additional functionalities later. Thus, it encourages being directed towards the simplest solution which can satisfy the customer requirements. In fact, within an iterative software development, not all requirements need to be definitely specified and stopped very early. However, even with the traditional iterative development, at a certain point, the specifications must be blocked to provide something. At this stage, the project management of the contents is concerned. In fact, with agile development, our specifications can change at any time. The idea is that the customer can continue to carry out modifications as a long time as it gives the priority to these changes during the suitable iteration.

For example, if the customer asked for three reports and that he wants one more lately, this last report can be added to the list of the requests without problem. In this stage, the customer gives the priority to the new report. If the budget of the customer is open, then there is no formal process of contents change, extra functionality can then be added later. Test-driven development approach affirms that when modifications occur, it must be prepared to respond. On the other hand, the specific modifications have consequences, in terms of budget and delivery dates, which must be approved by the partner.

In addition, feedback is immediate for the developers thanks to the unit tests. For the customers, the feedback is done on a scale few days by dint of the functional tests which enable them to have a permanent vision of the system state. A permanent feedback is positive. Any variation compared to the planning and its expectations can be detected, and consequently it is easier to correct them quickly. For the developers, a permanent feedback makes it possible to locate and correct the errors much more easily. This concept of feedback is essential so that the project can accommodate the change of customer request.

8.2 Technical Implications

In this section, we evaluate the technical implications rising from our practical implementation of software development process. We need a basis to appreciate product quality and effort measurement. We can considered EMP, as a commercial organization, has a four main objectives: increasing functionalities, customer satisfaction by reducing time to market, improving product quality and finally reducing the cost. We suggest a performance analysis to determine if the project meeting aims and goals.

The most important objective of EMP is to produce a high quality product within schedule and financial plan. We can achieve this ambition by improving software development process. As described in figure 7.3 of our proposed process improvement, several checkpoints (CP) are defined after each final module phase which characterizes a work status defined in the development process. On a project, the phases is divided into tollgates (TG) as a business decisions, which is predefined requirement which has to be fulfilled before the process is allowed to continue. The important predefined events during the project are called milestones (MS).

The MS provide a basic schedule list and progress information for main activities and events. Currently, before starting any project, it is possible to estimate the amount of effort to complete scheduled activities. We were interested for our implemented and tested “Local Wakeup Interval” functionality. Thus, we estimated the amount of effort needed between CP3 and CP5 according to the current mechanism of development (see figure 7.2). Afterwards, we compare the estimation effort with real time spending to complete the general tasks using test-driven development as a new concept at EMP. The major activities have been established within effort estimation which includes hours used for development as shown in table 1.

General task and activity	Estimated effort (hours)	Real spend time (hours)
Software Backplane implementation	15	13
Updated module test specification in SWBP	10	8
Updated module design specification	10	2
Hardware Abstraction Layer implementation	25	16
Updated module test specification in HAL	20	26
Code complete	20	18
Build for all relevant functionalities for debugging	10	10
Function Test complete and freeze the modules	5	5
Total working hours	115	98

Table 1: Mapping between effort estimation and real spend time on developing activities.

As result, we spend fewer hours for developing than the estimated effort in general. Unit testing help us to write code faster while increasing quality. Thus, by writing tests using CUnit as a framework, we spend less time debugging. Module test is an activity that is included in module implementation. The implementation and module tests are executed in parallel. Tests are running on the parts that are implemented at the same time as the implementation continues.

According to the results of table 1, it is not the best measure for evaluation because the spend time on developing activities depends definitely on person performance as well. The perfect way to appraise the process improvement comparing to the traditional approach can be done by two similar projects. One is developing with the current developing process while we apply our suggested solution for the other. Unfortunately, this is not viable in practice and it can be considered as a limitation for this project. That is why we based our measure on effort estimation as shown in table 1 in first step. Moreover, we suggest implementing the interfaces of a platform module with two different ways. In fact, within an iterative process and before tackling programming, a certain number of iterations must be defined for specific interfaces. These interfaces with different types are supported by Software Backplane and Hardware Abstraction Layer (figure 7.11). Those iterations, which are related to WLAN Control on the Software Backplane, are done with the new proposed development process by applying test-driven development, while the others are implemented with the current process of development. Here, we discover the major distinction between these two mechanisms. The results give us an idea about the fact that the total number of faults and errors for debugging was much lower within TDD compared to the traditional approach. Consequently, the amount time to spend on debugging activity varies significantly. In fact, according to several interviews with some technical persons, we notice that the main problem consist in the huge time wasted for system building. Working with both mechanisms, we perceive that after each simple modification, we need in average quite a lot of hours to build the complete system again with the current process. That leads much elapsed time of development for each test case. We avoid this kind of problem with test-driven development. Thus, we notice that time of build system decreases significantly within fewer faults during development phase.

Now, we can affirm that unit testing helps us to write code and to build system faster while increasing quality. The basic idea results from the training psychology: the more feedback follows closely an action, the more important training. The fast feedback for the developer allows a quick correction and a rapid change of functionality. Increasing product quality is related to the errors and defects occurred during the development stage. Within traditional approach, we have the experience of spending much time to find a small bug. When a defect take place, it is hard to find the source of bug because we must search the fault in the whole code packages. However, we spend less time to correct an error by using unit testing since tests can be run offering an immediate feedback. In fact, each time we encountered a problem, we tried to isolate this difficulty with an elementary test case in order to be reproduced as simple as possible. This enabled us to make much localised corrections to fix the errors quickly. We spent very little time to seek bugs in our code with TDD opposite to the actual mechanism. Indeed, we need some help from an experience person to fix a bug where we developed the interfaces of Hardware Abstraction Layer. While with TDD approach, we succeeded to figure out developing interfaces of Software Backplane and we spent less time to seek bugs in our code. In fact, instead of spending time to examine the execution of debugging system, we spent time to think of tests that can have another advantage. However, the time of debug is money thrown out of the window because this time is wasted without any benefit. On the other hand, we, as developer, take profit from this time spending to write tests in the future because these tests could be reused when the code is modified. Thus, time spent to make TDD is an investment, instead of being a loss.

In conclusion, the fewer defects occurring with TDD, the more product quality delivering and the less time we spend on system debugging (figure 8.1). So when development time can be decreased, customer satisfaction can be achieved as well by reducing time to market and reducing the project cost as well.

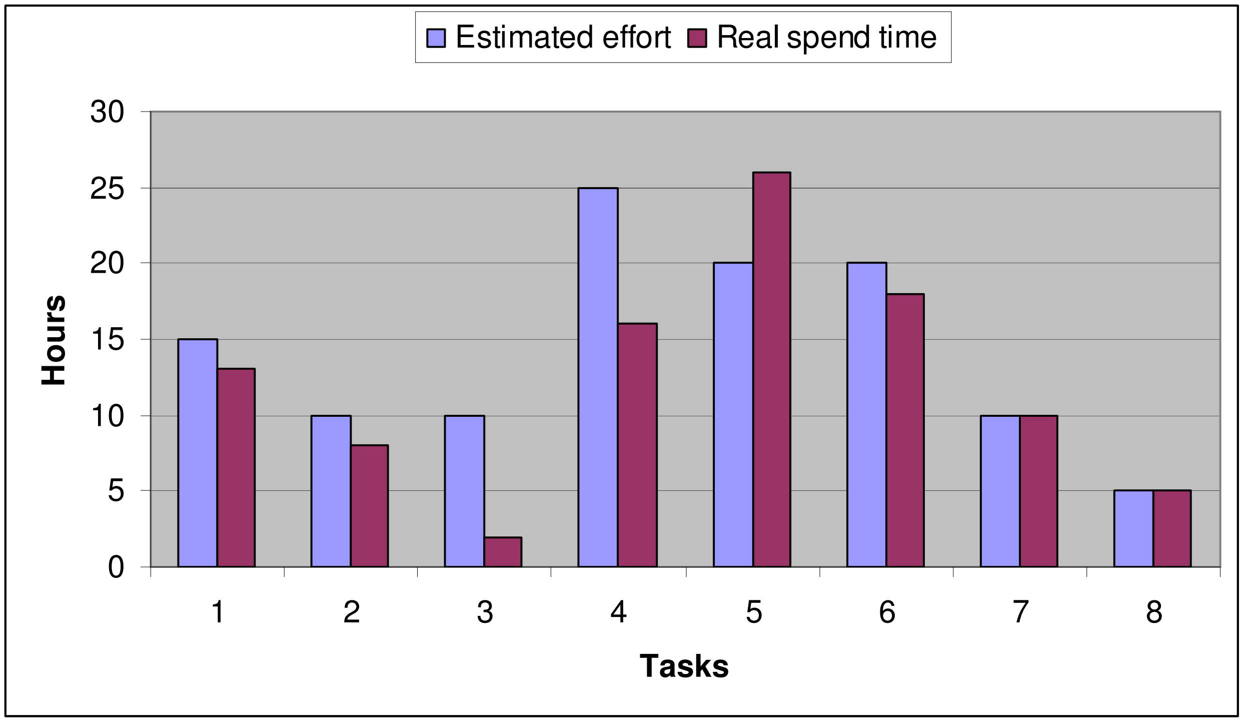


Figure 8.1: Illustration of effort estimation and real spend time on developing activities

CHAPTER 9 - CONCLUSION

The aim of this thesis was to improve the Ericsson Mobile Platforms (EMP) development process. Therefore, we investigated a solution that achieves a fast process with high quality output and handles a customer requirement. Consequently, a software development process improvement has been proposed.

We started by looking into different processes and working methods within EMP. We studied how new functionalities could be added without affecting the existing architecture of development process. Then, we tried to identify the main concepts of software development that lead us to achieve our solution. With the study of the current software development process, we learned and noticed its drawbacks. The main shortcoming is that the process takes too much time to add new functionalities. Furthermore, the sources of bugs are hardly detected.

In the first part of our work, we had the challenge to figure out an efficient approach for this limitation. It was very important for us to define a new mechanism that should be as simple as possible and suitable for the existing architecture of EMP. Thus, Test-Driven Development (TDD) as one of the most important Extreme Programming (XP) practice has been proposed. It was a solution to be applied in software development process improvement, especially; this concept of development was not used before at EMP.

The second part of this thesis put emphasis on the data communication technologies. We had the challenge to apply our proposed solution within one of the running development projects of EMP. Our project was related Wireless LAN as a specific area into EMP datacom platform. First, we described briefly the power management functionality which is associated to our pilot project. Then, new customer requirements for power save has been derived to be implemented. Afterwards, the different stages of development have been presented by giving an overview of each phase.

The performance of using TDD in software development process has been assessed by discussing both the theoretical and the technical implications comparing to the traditional mechanism. It has been shown that the proposed software development process proved the behaviour of TDD. It is useful in both implementation module and test module which are executed in parallel. The performance can be extended by increasing functionalities. We spent the major time developing new functionality rather than debugging the code. With decreasing the time of development, customer satisfaction can be achieved with reducing time to market. The fewer defects occurring with TDD improves consequently product quality and reduces the cost of project.

ACRONYMS AND ABBREVIATIONS

AC	Admission Control
ACK	Acknowledgement
AP	Access Point
API	Application Programming Interface
APSD	Automatic Power Save Delivery
ARM	Advanced RISC Machine
ASIC	Application Specific Integrated Circuit
ATIM	Ad hoc Traffic Information Map
BSS	Basic Service Set
CP	Checkpoint
CR	Customer Requirement
DS	Distribution System
DSDM	Dynamic Systems Development Method
DSSS	Direct Sequence Spread Spectrum
DTIM	Delivery Traffic Information Map
EMP	Ericsson Mobile Platforms
ESS	Extended Service Set
FHSS	Frequency-Hopping Spread Spectrum
HAL	Hardware Abstraction Layer
HR	High Rate
IBSS	Independent Basic Service Set
IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronics Engineers
MAC	Media Access Control
MS	Milestones
OFDM	Orthogonal Frequency Division Multiplexing
PC	Personal Computer
PDA	Personal Digital Assistant

PDCA	Plan Do Check Act
PDU	Protocol Data Unit
PHY	Physical Layer
PLCP	Physical Layer Convergence Procedure
PM	Performance Monitoring
PMD	Physical Medium Dependent
PS	Power Save
QoS	Quality of Service
RAM	Random Access Memory
S-APSD	Scheduled Automatic Power Save Delivery
SP	Service Period
STA	Station
SWBP	Software Backplane
TCP	Transmission Control Protocol
TDD	Test Driven Development
TG	Tollgates
TIM	Traffic Information Map
U-APSD	Unscheduled Automatic Power Save Delivery
UDP	User Datagram Protocol
USP	Unscheduled Service Period
WLAN	Wireless Local Area Network
WMM	Wi-Fi Multimedia
XP	Extreme Programming

REFERENCES

[1] Barry W. Boehm: “Software Engineering Economics, 1st edition.” Prentice-Hall PTR, 1981, Saddle River, NJ, USA.

[2] Wikipedia: “Software Development Process”, [referred 2008] http://en.wikipedia.org/wiki/Software_development_process

[3] Raghu Singh: “International Standard, ISO/IEC 12207, Software Life Cycle Processes”. 1995 Federal Aviation Administration, Washington, DC, USA. http://www.abelia.com/docs/12207cpt.pdf

[4] Barry W. Boehm: “A spiral model for software development and enhancement”. Computer, 21, 5, 1988. pp. 61-72. http://ieeexplore.ieee.org/iel1/2/6/00000059.pdf [referred 2008]

[5] Wikipedia: “Waterfall Model”, [referred 2008] http://en.wikipedia.org/wiki/Waterfall_model

[6] Agile Alliance. http://www.agilealliance.org/ [referred 2008]

[7] Manifesto for Agile Software Development, site design and network 2001. http://www.agilemanifesto.org/ [referred 2008]

[8] Barry W. Boehm: “Get Ready for Agile Methods, with care”. Computer, 35, 1, 2002. pp. 64-69.

[9] Wikipedia: “Agile Software Development”, [referred 2008] http://en.wikipedia.org/wiki/Agile_software_development [referred 2008]

[10] Highsmith, J., and Cockburn: “Agile Software Development”. The People Factor. Computer, 34, 11, 2001. pp. 131-133.

[11] Wikipedia: “Extreme Programming”, [referred 2008] http://en.wikipedia.org/wiki/Extreme_Programming

[12] Kent Beck: “Emergent Control in Extreme Programming”. Cutter IT Journal, 13, 11. 2000.

[13] Kent Beck: “Embracing Change with Extreme Programming”. October 1999 (Vol. 32, No. 10) pp. 70-77.

[14] Extreme Programming: http://www.extremeprogramming.org [referred 2008]

[15] Extreme Programming: http://www.xprogramming.com [referred 2008]

[16] Wikipedia: “Test-driven Development”, [referred 2008] http://en.wikipedia.org/wiki/Test-driven_development

[17] Kent Beck: “Test Driven Development: By Example”. Addison-Wesley Longman, 2002.

[18] Michael Ellims, James Bridges and Darrel C. Ince: “Unit Testing in Practice”. Proceedings of the 15th International Symposium on Software Reliability Engineering, ISSRE 2004.

[19] Pejman Roshan and Jonathan Leary: “802.11 Wireless LAN Fundamentals”, 1st Edition, Cisco Press, Dec 23, 2003.

[20] CUnit official website: “A unit testing framework for C.” http://cunit.sourceforge.net/doc/index.html [referred 2008]

[21] Eclipse: http://www.eclipse.org [referred 2008]

[22] Internal confidential sites at Ericsson Mobile Platforms (EMP), Grimstad, Norway. [referred 2008]

[23] Mary Shaw: “What Makes Good Research in Software Engineering?” International Journal of Software Tools for Technology Transfer, 2002, vol. 4, no. 1, pp. 1-7. http://www.cs.cmu.edu/~Compose/ftp/shaw-fin-etaps.pdf [referred 2008]