Comparison of Software Development Methodologies based on the SWEBOK

by **Elísio Maciel Simão**

Supervisor

PhD. João Miguel Fernandes

October 27, 2011

Contents

List of Figures

List of Tables

Agradecimentos

Gostaria de agradecer ao meu orientador o Prof. Doutor João Miguel Fernandes pela paciência e crítica construtiva dada durante a execução deste documento realizado em simultâneo com uma actividade profissional intensa o que aumentou ainda mais o desafio para mim e para ele.
Gostaria também de agradecer à minha família (Mãe, Pai e Irmão) pelo apoio incondicional dado que reflecte o verdadeiro conceito de família e que promove o que de melhor há em mim.
Finalmente à Liliana, por ser uma companheira de vida, e por estar sempre presente nos bons e maus momentos tanto para criticar quer para motivar, quando é preciso.
Obrigado!

Abstract

We are facing a period where software projects have a huge dimension involving small resources, high risk and a wide range of available approaches. In this scenario the Software Development Methodologies (SDMs) can prove to be a useful ally, but very dangerous and even fatal if misused. The big issue around this matter is how to choose the appropriated SDM that fits a specific project. In the given scope, this dissertation describes a framework for comparing SDMs delivering a set of procedures that should be followed when the choice of an SDM is made. The dissertation approaches the framework by applying it to a group of SDMs that were selected by their popularity and significance. This exercise is done to prove the concept of the framework and to provide a base comparison, with each chosen SDM, that can, and should, be extended by those who choose to use the framework.

The classification is achieved by defining a scale that goes from total satisfaction to no satisfaction, with an intermediate level of partial satisfaction, that is applied to a set of keys. These keys are based in SWEBOK (Software Engineering Body Of Knowledge) that describes and explains the different Knowledge Areas (KA) stating their common issues and best practices. To explain the framework, the dissertation analyzes each KA and evaluates the selected SDMs by assessing how their approach complies with SWEBOK's knowledge areas, using the previous stated scale.

The framework delivered can be enriched by its user who should provide weights to each KA regarding the project in which the SDM will be used and previous experiences.

Resumo

Actualmente atravessamos um período em que os projectos de software têm uma grande dimensão, envolvendo baixos recursos, alto risco e com um variado leque de abordagens a escolher. Nestes casos as Metodologias de Desenvolvimento de Software (MDS) pode ser um bom aliado, contudo se mal escolhido pode ser extremamente perigoso ou até fatal. A questão que se levanta então é, qual a metodologia a escolher.
Neste contexto, este documento descreve um conjunto de procedimentos a seguir para comparar MDS. Os procedimentos são então aplicados a um conjunto de populares MDS provando o conceito aqui apresentado, disponibilizando uma comparação de base com uma explicação para cada metodologia escolhida que pode, e deve, ser estendida por quem utilizar o conjunto de procedimentos aqui descritos.
A classificação é conseguida através de uma escala que vai da satisfação total à não satisfação, com um nível intermédio de satisfação parcial, para cada uma das chaves. As chaves escolhidas são baseadas no SWEBOK (sigla de *Software Engineering Body of Knowledge*, em Português, Corpo de Conhecimento para a Engenharia de Software), que descreve e explica as diferentes áreas de conhecimento da engenharia de software com referência às melhores práticas e problemas comuns para cada uma delas. Para o conjunto de procedimentos apresentado, cada uma das áreas de conhecimento é analisada e as MDS são avaliadas de acordo com a forma como abordam cada uma das àreas de conhecimento do SWEBOK utilizando a escala anteriormente referida.
Estes procedimentos podem ser enriquecidos por quem o escolha utilizar atribuindo pesos a cada uma das áreas de conhecimento com base no projecto a que a MDS será aplicada e a experiências anterirores.

Acronyms

ADM	*Agile Development Methodologies*
BDUF	*Big Design Up Front*
CASE	*Computer Aided Software Engineering*
CASL	*Common Framework for Algebraic Specification*
CMMI	*Capability Maturity Model Integration*
DSDM	*Dynamic systems development method*
FDD	*Feature Driven Development*
ITIL	*Information Technology Infrastructure Library*
KA	*Knowledge Area*
KLOC	*Kilo Line of Code*
PMBOK	*Project Management Body of Knowledge*
RAD	*Rapid Application Development*
RIPP	*Rapid Iterative Production Prototyping*
ROI	*Return of Investment*
SAGE	*Semi-Automated Ground Environment*
SCM	*Software Configuration Management*
SDLC	*Systems Development Life Cycle*
SDM	*Software Development Methodology*
SQM	*Software Quality Management*
SSADM	*Structured Systems Analysis and Design Methodology*

SWAT	*Skilled Workers With Advanced Tools*
SWEBOK	*Software Engineer Body of Knowledge*
TDM	*Traditional Development Methodologies*
UML	*Unified Modeling Language*
VDM	*Vienna Development Method*
XP	*eXtreme Programming*

Chapter 1

Introduction

1.1 Context

The importance of software in the world is constantly increasing, and with the increase of its importance also the size of the software systems is increasing as well as the effects of its development. These effects have consequences on the way software is developed. Software development of huge projects involves huge amounts of money and time, which have to be used in the correct way for delivering the final product with the minimal cost.

With the evolution of software development, different approaches have been presented and used to deliver a software product with aim of reducing time and cost of its development, without impairing the quality of the product. Each approach has its benefits and criticism but they all have similar goals, delivering a quality product in the fastest and cheapest way possible. When we look for a Software Development Methodologies (SDM) we have a set of very different perspectives, and even different approaches for the same perspectives, to choose from. These SDMs define the way we should organize our teams, and the way we should organize our activities, so they provide a good guideline for achieving our goals. So one big question that arises when starting a project is which methodology to choose. Perhaps an even bigger question is how can one answer that question, giving that, each project has its own specific characteristics to take in account. Most companies simplify this issue by adopting one or more methodologies and then all their projects are done using a specific methodology. But, given the quantity of different kind of projects a software company is nowadays involved in, is this the right answer?

In times of economical crisis, where budget and resources are more limited than ever, a bad choice or a misuse of a methodology can lead to its death or to spending more money and resources than expected.

So when it comes to choose a methodology we should rely on something that assures that we are beginning our project using the correct methodology, for the specific characteristics of the project in cause, that the methodology fits those characteristics, and will guide the development the cheapest and fastest way possible. So the big issue is, how can one choose the best SDM for one project.

1.2 Goals

This dissertation provides a set of tools that one can rely upon when choosing an SDM. In this document you will find an explanation of the framework used, and the comparison of some of the most well-known SDMs.
To provide a comparison of SDMs and not to be biased by the project involved, empirical testing and evaluation of the projects was not a desirable solution, so a comparison method should be used without regarding the project it self but a established set of facts and activities involved in the development of a software project. The goal is to provide a framework for evaluating SDMs and output a mean of comparison between them, that can be adapted according to the project and context of the software system to develop.
The principle of comparing SDMs is the main goal of this dissertation, and it must be clear that the intention is not to provide the best SDM. So, if the goal is achieved, the framework output in this dissertation, together with someone who is able to identify which characteristics are most important to a given project, should be able to aid the election of an SDM, that fits the best to the project. That process can be done by, after measuring the approach of the methodologies according to key factors, and giving weights according to needs of the project and its characteristics. Then we can get a comparison of the SDM with an output of accountable metrics according to the characteristics of the project. Thus, the subjectivity is in the side of the one that chooses, by determining the weights, and not by the classification of each SDM. The choice of weights can also be the flaw of the framework, but it helps to serve the one who chooses the weight. The problem of correctly choosing the weights is a problem that won't be addressed during this dissertation, but it is a problem to take in account when making this choice.
The framework delivered here is provided together with the explanation of each key factor for the comparison of SDMs.

1.3 Structure of the Document

The document is structured in 5 main chapters:

- **Introduction**: considerations about the structure of the document and the terms used during the text. A clarification of the objectives of this dissertation and the context involved is also done in this part of the document.
- **State of the art**: in this chapter an overview of the software engineering topics is done composed by a brief history of software engineering, clarifying the different edges of software quality and explaining each knowledge area that is used in the framework.
- **SDMs Description**: in this chapter the concept of SDM is explained in detail and a group of popular SDM is explained and analyzed. This analysis will be used further to support the framework proposed in this dissertation.
- **Results**: in the results, the analysis done previously for each of the SDM, will be used with the framework proposed in this dissertation and the data produced is delivered providing the conclusions and satisfaction data according of the framework. This data consist on the satisfaction classification and an explanation for this classification.
- **Conclusion**: the final chapter is a wrap up of the work done, stating pros and cons of the approach taken in this dissertation and referring to further work that can be added to this dissertation.

1.4 Terms Elucidation

When talking about software development methodologies SDM an elucidation about the terms used is needed, since they vary from author to author. In this dissertation the term **software development methodology** (SDM) is used when referring to a set of practices and roles combined with a definition of software life cycle, also referred as model that supports a certain philosophy of development.
So, when an SDM is referenced it should be read as the combination of the following concepts:

- **Philosophy:** the philosophy of an SDM is the principles behind the SDM, the description of the SDM should support this philosophy. It is frequent to see a lot of authors defending a certain philosophy and criticizing an SDM, that allegedly follows some philosophy, of not doing it, so some times it is difficult to clearly define an SDM philosophy, because when defining an SDM you are putting in practice your own interpretation of some philosophy. When talking about philosophies we also have to take in account the level of abstraction a philosophy have, it can be divided in sub-philosophies (a good analogy is with religion, for instance, there are a lot of philosophies underneath Christianity, and even more religions that practice these philosophies with roles and practices underneath those).

- **Software Life Cycle:** is the backbone of the SDM. It defines the sequence of activities that the software development should follow. It is also responsible for defining its precedences and outputs, according to the cycle of development. The software life cycles can be categorized by the way the activities can be arranged, most of the times associated with the software development philosophy. These categories can be related to the nature of the development cycle, for example being linear or incremental/iterative, or the way they address the sequence of activities, for example, big design up front (BDUF).

- **Practices**: define the activities the SDM should follow, considering its life cycle. These practices come obviously associated with the software life cycle, and with the definition of roles. The practices define the need of performing certain activities to support the philosophy the methodology is build on. Some SDM practices become popular and notable and are sometimes adopted by other methodologies or ad-Hoc development (one of the most notable examples would be the practice of pair-programming and eXtreme programming).

- **Roles**: defining the scope of action and profile of the project's team members. This involves splitting up the activities for each members, but also defining the main focus of each team member by addressing them a role. The roles may also involve responsibility of outputs and procedures.

The separation between some of these concepts it is not always obvious. The software life cycle is associated with the activities and roles and obviously the philosophy can also consider roles and practices if its abstraction level is sufficiently low. The separation of these concepts is important but an SDM is composed by the combination of these factors, and when they come together they define an SDM's weakest and strongest links.
Some SDMs have several versions, that may differ on the life cycle or in establishing different practices and roles. In this dissertation we also extrapolate some software life cycles adopting the practices and roles that are more addressed in the bibliography or most used in the industry.

Chapter 2

State of Art

2.1 Software Engineering

In the early days of computer science, programming was viewed essentially as a sequence of instructions and it's main problem was how to place it [11]. All the problems were well understood and normally were used to solve problems, and thereby written by the people that were trying to solve it, so no other person would be involved.
Then computers started to be more common and the amount of people using them increased, and the concept of *programmer* emerged. People were asking others to write their programs implying the use of more evolved, and complex languages. The introduction of the programming business started with the separation of the user, the software and its development, thereby the user had to specify what he intended and then the programmer would have to *translate* it into programming notation [15].
With the increase of the complexity of software to develop and of the amount of people involved in the process, communication took an important role in the process, specially in transversal tasks (tasks that involves different roles within the team). The increase of the importance of communication was also justified by the increase of misinterpretations of what the user intended leading to big differences to what was actually being developed.
The term **Software Engineering** was first formally used in 1968 during a NATO conference [34] in Garmish (Germany) because and during the so called *software crisis* [12] derived from the increase of complexity and computing power of the software projects. Most of the projects were major flops and some of them with irreversible damage including the death of people [25].
Some authors have defined Software Engineering as the development of software by many people and with many versions [33]. This definition brings out

the difference between programming and software engineering, while the first is used to write programs the second is concerned with the process to deliver a program developed by numerous people and with different areas of action. Therefore a software engineer has to be able to translate what is intended for the software to perform and also concern all the different levels of abstraction, from the user point of view to the specific aspects of the programming language, and in different stages of the development process. Software engineering as an obvious relationship with computer science but also with other areas of knowledge, or disciplines, such as management and system engineering, from which much information as been used and adapted to software engineering. Therefore the way the team interacts, and the definition of stages of a project, became an essential part of software projects and different definitions of how to structure, plan and control the software development, became a major study in software engineering. The arising of software engineering also brought up the terms of Software Life Cycle and SDM.

2.2 Brief History of SDM

During the so called, and referenced previously, software crisis, software engineering research was focused on finding a solution for the problems of productivity and quality that were taking over so many projects. This originated a series of developments that were of great importance to software engineering. The concern by the way software was developed started with some authors addressing specific aspects of the process, an example of this kind of output is structured programming [11]. Most authors suggest this represents a naive approach for an SDM, and do not consider it as such, claiming that an SDM is an integrated set of methods and tools that define the life cycle of software development by describing its activities and roles [3]. So they suggest, that Structured programming, doesn't fit this definition and should rather be considered as a technique or practice. Nevertheless this concern with the the way software is developed is strongly related to SDM and can be consider as its ancestor. Another example is object oriented programming, a program paradigm, that also urged during this time, providing tools to deliver better software and stating software development principles to improve the quality and productivity of the development process. During this crisis, the use of formal methods to prove software correctness also had huge developments. In the social component of software engineering and with the establishment of the programmer profession the definition of ethics and principles were also evolving quickly. Sooner these developments were becoming *silver bullets*, believed they solved the problems that were taking over software projects. In 1986, Frederik Brooks wrote an ar-

ticle about how in that decade none of the software techniques would bring an improvement to software production by it self [8], [19], representing the scepticism that was also associated with which those developments represented by them selves.
Parallel to these developments the concept of the software process, and its representation, emerged and sooner the definition of the, now called, traditional development methodologies also appeared. These methodologies were characterized by the use of linear and step-forward software development, and based in models (or software life cycles) that defined the steps that a software project should follow.
Another milestone on the history of SDM was created by the combination of ideas from several scientists, later formalized by James Martin in a book in 1991, which had the same name as the SDM developed in the 80's, RAD (Rapid Application Development). RAD intended to be a response to the called TDM (Traditional Development Methodologies) that allegedly its use implied the projects to be so long that the prerequisites defined in the beginning of the project, by its end were already obsolete, causing the solution to be useless and therefore to be a waste of money and resources.
These critics originated the rising of the called ADM (Agile Development Methodologies) with Jeff Sutherland, John Scumniotales, Jeff McKenna and Ken Schwaber [36] adapting the method based in the work of Takeuchi e Nonaka [43] applied to the Software Development. The work of Takeuchi e Nonaka was based in a rugby technique called Scrum, name which the SDM inherit.
Around 1996 Kent Beck formalized the concept of eXtreme Programming (XP) in his book *"Extreme Programming Explained: Embrace Change"* [5]. This approach was used previously by Kent Beck and other authors, but without the proper notability and only in industrial projects without any formal definition of the SDM, at least externally to the companies in which it was being used.
In 2001 some of the authors responsible for this SDM gather and came up with the agile manifesto, a sequence of principle that they all subscribe and that is available to new subscribers, this subject has a further explanation in 3.5.1 and more on this subject can also be found in the bibliography [42].

2.3 Software Qualities

Software quality is intrinsically related to SDM, as it constitutes the objective of its use. The different kinds and types of qualities are also related with the different phases of the software life cycle. The quality of the software produced is one of the most crucial responsibilities of the act of software engineering. Other fields of engineering have much more tangible ways of perceiving that quality,

civil engineers have ways of demonstrating the robustness of their buildings, and a mechanical engineer has to demonstrate that an engine works properly. The kind of products produced by these disciplines is much more known to the humanity and interacts directly with the common sense of people. The product of software engineering is a software system, even if it is a much more abstract product, it still is, a product (produced by the act of software engineering), and becoming very quickly a product that is used by everyone. Therefore the quality of a product as to be assured, and the problem of defining the quality of the product as been since the beginning of software engineering a challenge which as lead to a amount of discussions and theories around it.
Even being a different task to define what is a software system with quality, there are some formalisms that are, nowadays, common to most software system, without regarding that are specific qualities for different kinds of software systems (a web page, has different quality measures than a core system of a bank).
The term quality is not only focused on the system produced (the software system) but also in the way that the product is developed (the software development process). Additionally, quality can be divided into two major groups, the **external qualities** which are visible to the users and to the people that interact with the system, and the **internal qualities** those which are only visible to people that are involved in the development process (people that develop the software system and the ones that have to maintain it).
In this section we will approach some of the qualities that are common to most software systems, even if for some projects the weight for each of the qualities changes. This section will present a definition for the terms that are further used during this dissertation concerning the quality of software systems. A correct definition of this terms/concepts will be particularly important during the explanation of the SDM. During this, a correlation between quality and the SDM phase, its activities and roles will be established clarifying its goals and purposes. So to fully understand the goals of a certain activity or roles is important to understand what each of this concepts mean and from which point of view they will be addressed.

2.3.1 Correctness

The term correctness is intrinsically connected to software specification. A software specification is where it is stated what a software system is intended to do. The formalism of this specification can vary, the field of formal methods in computer science take the approach of defining the specification with aid of algebras and co-algebras, which is than used to proof the correctness of the software using these formalisms. Is also very common that the term requisites is used along the specification, which is a much more empirical way to specify what is intend for the system to do. Nevertheless, the correctness of a software system is the act of comparing an implementation against its specification. As stated above, the correctness can be verified using mathematical formalisms, the algebras and co-algebras, or by a more empirical way, by performing tests. These techniques are related to the way the specification as been done, and also, although not directly, by the implementation it self. For example, the use of high level languages, specially functional languages, turns the use of formal methods more easy to perform, there are even specification languages (such as VDM, CASL, UML to more information, please refer the bibliography: [45],[9] and [44], respectively). On the other hand, empirical specifications, and the prove of correctness, can also determine how software is developed, one of this approaches is commonly known as Test Driven Development, and the prove of the specification is done up-front in the beginning of the project and the implementation will then be done to surpass the tests previously developed.

2.3.2 Reliability

This concept is highly connected with correctness. Reliability is by definition the ability of a software system to perform which it is required to, and under the conditions it was designed for. It would be of common sense to say that if a software is Correct, then it should be reliable, if the specification fulfill the user requirements and if it does not have coding errors. Although, even if the software is correct it does not imply that is reliable, it could suffer from performance issues, or functional errors that could turn the software into not reliable. Also the concept of reliability is relative, because an error in the program can be harmful and by that you can still rely on the system. Relationship between reliability and correctness can be retain by imagine a mathematical domain $C \subsetneq R$, where C represents correctness and R reliability. Although, in a practical way a program can be incorrect but reliable, if a the specification does not fulfill what is intended but the implementation does (and all the issues of reliability are fulfilled), than the software is reliable but not correct (against that particular specification). Even though, if we abstract the term of the specification to the level, of the program doing **what the user intended**, and not to the **representation of what the user intended**, then correctness also involves a correct specification (which would fail in the above example), and therefore the previous theorem is true.

2.3.3 Robustness

A software system is robust if it is reliable in not only the circumstances that were predicted in requirements and specification. These circumstances are the ones that are not in power of the developer to avoid, such as network crashes or erroneous input from the user. Comparing to classical engineering disciplines, such as, a motor of a car, one can say that a motor is more robust than other because it resists more time to the erosion of the sand in the desert, or because the pieces are more resistant to higher temperatures compared to other engines. Even in these cases, the concept of robustness is not absolute, and software robustness it can be related with different aspects of the software system. Robustness is commonly addressed when talking about operating systems or core systems, and normally comparing the time it can perform without faults or without the need of restarting the system. Robustness also measures the capability a system has to recover from faults and the way it interacts with other systems when a fault occurs. Robustness is normally achieved for preparing the system to have a certain behavior under fault and to address the best use of its environment.
Like stated previously robustness is a relative concept and for that reason normally a software system is proven robust against a standard, that should also be stated in the specification (leading that correctness can imply robustness), and standards are commonly delivered by some institutions and companies which can address which the software system should be ready for and normally the tests it should be exposed before being considered robust.

2.3.4 Performance

Some disciplines in computer science do not consider the performance as efficiency (the act of performing a task), and a program is considered efficient if it ends performing the desirable act (less the infinite tasks which derive from the conditional task e.g. while). In software engineering, as it is more into the empirical experience of software development, efficiency is turned into the use of computing resources in an economical way, strongly connected to the concepts studied in the field of complexity. In software engineering, a software should perform in a reasonable amount of time (bringing back the concept of reliability 2.3.2) . If a user can rely on a software it should perform tasks without the lost of a huge amount of time, which would lead to a lack of productivity 2.3.10, and would affect usability 2.3.5. Some would think that the performance quality with the increase of computational resources would become an obsolete issue, even so, by definition performance and efficiency would demand the use of

these computational resources economically, but also, the argument rests with the introduction of mobility, Internet and with virtualization. Therefore the use of the resources in a proficient way, is required to the quality of the software produced.
Regarding the complexity of algorithms, a very famous approach for this analysis, is the asymptotic approach commonly know as big-O notation which describes the behavior of a function when its limits tend to infinite value. The performance can be calculated using different kind of approaches from the use of calculus of the worst case scenario or more empirically by monitoring the activity and measuring it, with simulation, for example, where the performance of the model is evaluated. It is also common to use stress testing to see how performant a software system is (this kind of testing is also normally used to measure robustness 2.3.3).

2.3.5 User Friendliness/Usability

Usability as been an increasing issue, with the massification of the use of computers. This is one of the most subjective software qualities. A software system could be considered user friendly for a mathematician but it would be very hard to use by a children. User Friendliness is also connected to the disciplines of design in which some standards and best practices exists, but it still relies in a huge amount of subjectivity, regarding someone's taste and accommodation to previous experiences. Some methodologies to address this issue always consider a user representative group that can help to understand the taste and previous experience of a user and accommodate the design . Of course when a software system is intend to mass distribution this kind of scenario is more difficult to predict but benchmarking and statistical input can help minimizing it.
In this particular quality, one can see the *humanizing* factor that software engineering introduced when compared to other areas of computer science, specially when it considers directly the communication issue regarding the concern that software is transmitting to the user what is intended to transmit and in a way that the user can understand the output given.
The user friendliness is also connected to other qualities stated above, such as correctness 2.3.1, a software that does not give the user the correct output is not user friendly, as it induces the user into assuming erroneous facts, also if a system is not *performant* it will not be very friendly to the user, as it has to wait an undesirable amount of time for the software to respond or to act, this is also related to the performance quality 2.3.4.
Previous experience and the background of the user cam also influence usability.Some methodologies suggest a role, that can be address using a representa-

tive group, or event take into account, specially when the software is targeted to mass-us, benchmarking and statistical input to help on this matter.

2.3.6 Verifiability

The qualities stated previously were centered on the desired behavior of the system produced, verifiability and the further explained qualities are focused on what the properties of the code produced allows to do. Verifiability is on of the most forgotten qualities of software. The possibility to verify that a software system is correct is indeed a difficult task if the abstraction level of the language on which the software is produced is very low, even the use of good practices on the structure of the code, and it's modularization affect how verifiable a software is.
This is a quality that easily fit in the group of internal qualities, but if you take into account that a software of Internet banking must insure the user that the security is verified we can bring out the quality to fit also as an external quality. To measure this quality is not always an easy task, and is done mainly in an empirical way by trying to prove that the software is correct. The ways of performing a verification, have already been stated above, they can be calculations, with strong formalisms, or monitoring a system and producing an analysis of its behaviour (auditing a software system).
This quality is easily obtain by assuring during the process of development that the software is verifiable with best practices, such as proper commenting the code and providing test cases (or formal methodologies) for every functionality developed.

2.3.7 Maintainability

To maintain software is necessary because a software system is intend to evolve with time, considering adaptation of the context where it is built in (e.g. user,operative system, system core, hardware), or the lack of some of the referred qualities, which would imply a modification in the software system, so it gains a correct behavior. Studies as been carried stating that, in some projects, the time used in maintenance is bigger than the time taken to develop the system. Nowadays with the increase of outsourcing, systems are often developed by people that will not be responsible for maintaining the system, and after the final release of the product, will no longer be with contact with the system. Maintainability is a quality that customers take into account with an increasing weight in the project development and management.
Most SDMs, since their main target is the development of software, don't have a specific maintenance phase (even though, specially the TDMs, do have this

phase), but the main concern of an SDM should be to properly provide the conditions for future maintenance, this can be done by providing tools and documentation that can be used in the future and addressing good practices when developing the system (e.g. commented/understandable code, encapsulation).

2.3.8 Re-usability

The concept of re-usability as two sides, one as an internal quality, that the code produced to some determine module of the project can be used in another system if serves the same purpose and the other, an external quality while the program can be used in different contexts without the need of rewriting a program. The common example when describing re-usability is to address the example of the batch command, when is intended to perform a simple operative system task, one could write an assembly or C program to execute an amount of basic actions, on the other hand it could built a command line instruction or a shell script, that is in fact reusing the amount of work that a batch command requires, and it would be probably a more efficient way of doing it.
From the point of view of the software life cycle, it can also reuse parts of the project with different ways, if a calculation is needed in various modules of the project, maybe is a good start to do a generic calculation module, that can be reused by the other components of the project.

2.3.9 Interoperability

The question of interoperability has two branches :

1. First is **portability**, which means the ability of a software to be installed in different environments. To measure interoperability is extremely hard, and it's normally done when designing the software depending on the environment; and

2. and the other is **integration**, which defines the coexistence with other software elements. This is really crucial in some particular cases, such as decision aid software or core systems, where is common that many programs use the information generated by other software systems.

In some ways portability affects integration, because when trying to make a cross platform software, the integration with the different systems is sometimes tricky due to the differences between platforms. Some software companies, prefer to deliver different releases of the same product to different platforms, taking advantage of the development made for each of the release 2.3.8. Integration can also be assured by creating a layer that addresses this problem specifically by creating components to assure the coexistence with other software elements.

2.3.10 Productivity

This is mainly an internal quality, and not referring for the productivity of the software it self, but from the process of developing the software. The productivity is normally measured by the amount of deliverables, or for the evolution of the project it self. A lot of constraints are introduced when measuring productivity. In a group of individuals, the productivity is not the sum of the particular productivity of each member of the group. A team can maximize or minimize the productivity of each member. The SDM and project management practices crucial for the productivity of a team.
The productivity is by far the most influential quality when determining which is the most appropriated methodology to develop a project, and also where most SDM focus on increasing. Unfortunately is also one of the most unpredictable qualities, causing it to be one issue that as been studied extensively. The SDM is responsible to aid the productivity and to provide tools/methods to measure it, and this is probably where most methodologies diverge, in the balance between increasing productivity and where to *cut the fat* of the project.

2.4 Knowledge Areas in Software Engineering

In this section we analyze the knowledge areas of software engineering according to the SWEBOK (Software Engineer Body of Knowledge) [18]. The SWEBOK provides a division of the knowledge areas (KA) into sub-areas. The KA within the scope of Software Engineering are:

- Software requirements;
- Software design;
- Software construction;
- Software testing;
- Software maintenance;
- Software configuration management;
- Software engineering management;
- Software engineering process;
- Software engineering tools and methods; and
- Software quality.

Now each one of these knowledge areas is going to be explained with further detail.

2.4.1 Software Requirements

- *Software requirements fundamentals*
  A requirement is defined as a property of a real life problem that must be exhibited by the software and the use that is given to it. Requirements define not only the functionality of the system, as it can describe technical details or automation tasks, it as a wide range of definition which must be verifiable.
  Software requirements can describe the properties of the product it self, what it should be able to do, and how it should be done. These requirements derive from the system requirements as they define how hardware, firmware, software and all the support elements of the system. They define the system as a whole, what all its components should or should not do.
  Functional requirements addresses to functions and actions that the software can perform and non-functional requirements are defined as constraints to the product, normally addressed to the qualities of the software to produce (section 2.3).
  Emergent requirements, refer to properties that cannot be seen into functions of the software but as an overall goal of the software as they rely on the system architecture and on the system requirements.
  Quantifiable requirements are desired when it's possible, specially when addressing to qualities, even if the qualities don't refer to software qualities (a good example would be, the system should reduce the amount of paper work in 80%).
  These definitions of requirements are not isolated, as a requirement can fit into more than one definition, these only pretend to define kinds of requirements and how to classify them.

- *Requirement process*
  The requirement process in an activity within a software project and it's normally initiated in the beginning of the project, although they should be continuously refined throughout the project. By not being a discrete process, normally, is very hard to define how the process should be performed.
  A requirement process should identify the requirements as a configuration item and it should be faced with the same practices as any other software element of a development process.
  The requirement process involves different actors or stakeholders which include (not exclusive):

  – *Users*: The users are very important when defining requirements as they are the ones who will be operating the system. It's normally an heterogeneous group, with people with different levels of education and with different roles and requirements needs;

- *Customers*: Customers are the ones who have commissioned the software, who ask for its development, and normally represent, together with the users,the target of the software to be implemented, but who might have different requirements types. A good example to differentiate users from customers is a example of a call center system, which the users, will be the phone assistants, and the customer being the company who ordered the software with a specific requirements. In this case the requirements of users and customers are potentially different, if you imagine that one of the requirements of the software is to monitor the phone assistants activity;
- *Market analysts*: Often, specially when addressing to the called mass-market product, the user can not be easily personified, so a market analyst represents the called common user defining what the user needs acting as a proxy for the users;
- *Regulator*: Some specific domains have constraints in which a software can do and what it should do (such as financial services, or aviation industry). Software has to comply with the requirements defined for the regulators of that specific market or to seek a certification that allows them to operate with certain players; and
- *Software engineers*: Specially on integration issues, software engineers have an important role in defining the requirements for the system to comply. When the development of a system or a component can imply with the reuse of other components, software engineers are called to decide technical requirements, causing them to be important actors on the requirement process.

Most of the times is not possible to comply with all stakeholders requirements as is the job of a software engineer/project manager to negotiate the different trade-offs, without disregarding that to negotiate is necessary that every stakeholder can be identified and their *stake* analyzed.
The requirements process has to be managed and supported, and the cost of each activity analyzed within the different cost and human resources issues.
One of the difficulties of the requirement process is the assessment of the its qualities, cost and time-line within the overall project. As well, customer satisfaction is directly correlated with the requirements definition and the fulfillment of those requirements.

Quality standards and software improvement models are a good aid to orient the requirement process, this subject is closely related to the software qualities and with the definition of the software engineering process knowledge areas. This topic also covers some techniques such as requirement process coverage and benchmarking.

- *Requirements elicitation*
  This topic is related in how the software requirements are retrieved by the software engineers. Human activity is crucial to the understanding of requirements, where communication is crucial within all the project and a specialist should be assigned to mediate the domain of the users and the technical lexicon of the software development, normally designated as business analyst. Requirements sources can come from the process actors or from the different aspects of the organization and operational environment or even from the business and technical knowledge from the market and technology involved.
  There are different techniques of covering these requirements such as interviewing, prototyping, meetings and even observation. The techniques involved should be measured by its cost and resources need. One of the most critical factors in big projects is not involving all the necessary people, and its a difficult task to assure that the correct people have been assigned to the project, since normally too much people reflects in harder negotiations and bringing people without power or knowledge to decide can influence negatively the agility of the process.

- *Requirement analysis*
  The requirements analysis have to deal with the classification it has defined in the requirement fundamentals, to its scope and properties within the project and they should be classified for its volatility or probability to change.
  The use of conceptual models are important when analyzing software requirements, they assist the understanding of the real life problem into the design of the system. The correct kind of the model has to be chosen regarding the skills of the team involved, the nature of the problem it self and should be aware of the tools and methods available. The models are various and can go from data flows to user interaction models and they are an important aspect when analyzing requirements. Another important topic is the negotiation of the requirements with the stakeholders as they may be incompatible, reinforcing the need for communication in all the process.

- *Requirements specification*

The specification deals with the formalization of the requirements and how they should be delivered. The importance of the production of the document and its separation levels, some projects have the need of dividing into several documents or specification according to the classification of the process. The specification must be clear and well standardized, without disregarding the specific needs of the project.
There are several standards to the documentation and production of software specification as they are a good aid to create common terms and rules for the requirements to be well understood by the developing part of the project. In this point, a lot of SDMs diverge into the level of abstraction these documents should have,generally the more traditional methodologies support a more detailed specification with a lower abstraction level, which also implies a bigger risk if a requirement change occurs, and more modern, specially agile, methodologies support more abstract requirements implying more effort on the development but being more adaptable to requirement changes.

- *Requirements validation*
  The validation of the requirements is crucial for the success of a software system, as it proves that the system complies with its objectives. There are several practices advised for the validation of requirements as a software methodology should use them for the success of the project. These techniques cover model validation, the proof of requirements and more empirical techniques such as, requirements review, acceptance tests and the use of prototypes/simulators prior to development in which the system produced (prototype) is used to validate the requirements.

2.4.2 Software Design

- *Software design fundamentals*
  Software design is the process of defining the characteristics of the system, its component and results. These characteristics can be divided in its architecture, components and interface. This process represents, within the software life cycle, the activity of analyzing the requirements and producing the software internal and external structure description.
  Design can be separated into two main activities: **top-level design**, where the architectural design is performed, describing the top-level structure, organization and description of it's components; and the **detailed design** where it's component it's described into more detail to allow its proper construction. The result of this design can be a set of models and/or artefacts that can log the major decisions taken.

There are several principles involved when approaching software design. Abstraction is a principle where the information is viewed from a very high level, removing specificity from the components leading into the production, parametrization and specification of the solution. There are several levels of abstraction, and it's normal that the level of abstractions is reduced with the evolution of the project. Abstraction can also be divided into the elements it represents: data, procedural and control iteration as many others.
Besides the division considering the abstraction levels, design is often divided into functional and technical design, being the first the approach the system will have according to the requirements and the second entering into technical details. This approach has the advantaged of non-technical users to validate and understand the design of the system and being able to argue and comment the delivered design.
One important principle in software design is the relationship between modules, that can be divided into coupling, the strength of the relationships, and cohesion, how the elements of each module are integrated.
Recommended and often used techniques are decomposition and modularization, which consists in dividing the problem into smaller independent problems, that can be treated separately and then integrated into a major goal.
Encapsulation is a technique, very related with the object oriented paradigm, in which the internal details of each module or component are isolated, with each module acting like a black box. It is also common to separate the interface from the implementation of each module, which is related to the previous concept of encapsulating the components, and separating what it is known to the users from what is not.
Software design has to correctly capture all the characteristics of the system, and nothing else. Fulfilling sufficiency, completeness and primitiveness.
Software design is obviously related to software requirements and it is common that these activities are done together reflecting the requirements and design in a single document or procedure, although it varies from SDM to SDM, where before detailing the design, a description of the requirements that support the design are addressed.

- *Key issues in software design*
  Software design addresses many issues in its process, it has to deal with concurrency issues related to the efficiency and execution of processes and how to decompose the system into processes and threads.

All these processes, if they run concurrently, have to be synchronized and scheduled which strongly relies on the atomicy of the design.
The data-flow control and the data organization is also crucial to define the design, how to react to different events, according to its characteristics and how to handle them properly.
The distribution of the software across the hardware is also an important task when addressing software design, specially when handling with distributed software and client-server architectures. This activity deals also with how the components will communicate with the concepts of the use or non-use of middle-ware components.
Interaction and the presentation of the information to the user is also an important task of the process of design software, which information is pertinent for the user to see and for whom is important to be seen. The handling of the information, for how long will it remain accurate how long and when it will be available and other information flow issues are also important when dealing with the process of Software Design.
The design of the user interface and the importance of properly addressing the issues of usability/user-friendliness qualities are crucial elements during this activity.

- *Software structure and architecture*
  The software architecture is a description of its components, how they are organized and the relationship between them. With the evolution of software engineering the architectures started to be decomposed, not only by abstraction levels but by the style they reflect on the system, which also introduces the notion of architecture families, with a common set of constraints that each style follow. There are several styles and structures of architectures which can go from distributed systems, interactive systems, adaptable systems among others. The use of architectural styles is recommended for its common use, creating an easy understanding of technical constraints for common related platforms. Although it should be carefully use so it will not disregard the specific aspects of the system being developed.
  Other standardization commonly used within software engineering are design patterns, which differentiate mainly from the architectural styles by it's level of abstraction. Design patterns are also called micro-architectural patterns, and they describe common solutions for common problems.
  The reuse of software has also to be addressed in the design process and it is an optional approach to the development of a system. There are several families of programs or frameworks, often addressed as packages, that can

be used to resolve common problems. The reuse of components is also an option to be addressed in the design process.

- *Software design quality analysis and evaluation*
  The quality of a software design is strongly related to the quality of the product delivered, therefore the measure and review of the quality of the software design is a important activity of the design process to ensure that the final product will have the desired software qualities (section 2.3). These can be done by assuring that the software design meets the required software qualities, obviously some of the software qualities cannot be measured during the design process, because they have to deal with the execution of the software. All of the qualities should be revisited after the implementation (the satisfaction of the software qualities in the design process does not imply the satisfaction of its implementation).
  There are several techniques that can provide an aid to ensure the quality of the software design. Review of the designs artefact's with formal or informal tracing, static analysis or non-executable analysis or more dynamic approaches using simulation and prototyping.
  Qualities are often hard to quantify, and therefore to measure quality is not an easy task, but it can help to estimate various aspects of software design. The use of measures to asset the size, quality and structure of the design have been proposed according to the approach chosen to perform the design. When referring to measures there are two main categories, the functional oriented design measurement and the object oriented design measurements. The fist category approaches design structure by decomposing the design into its functions and computing measurements for each function. The second category uses measurements at the level of class, a subcomponent of the design representation internal data (normally a class diagram).

- *Software design notations*
  Software design notation is used to built common terms, that can be an aid when understanding the design. There are several software design notations that are based into different levels of design, abstraction and design styles. These notations will not be extent in this document. For further informations of software design notations please address to the bibliography [18], [14], [32] and [29]. The important idea to take from this sub-area is that it is advisable for a software development methodology to have a software design notation or to promote the use of one. Is also usual that a project have a specific notation, according to the team involved and the environment where project is enrolled. Some projects also use

glossaries to aid people with the notation used and to help newcomers to easily read and understand the notations used and therefore the software design.

- *Software design strategies and methods*
  A software design strategy guides the design process and the method. Being more specific defines the use of a notation and a process that have to be carried to produce a design artifact. There are several strategies, that can be used when performing the design, and as the previous sub-area it will not be further detailed in this document, due to the extent of strategies existent. The use of design strategies and methods is a good aid to the design process and an SDM should address the use of one, or to promote it. To choice of the strategy is related to the kind of product that is being designed and the conditions in which this is being designed. This strategies can also be fitted into groups according to the initial approach, top-bottom or bottum-up are typical groups where strategies can be grouped into. Some SDM also directly influences the design strategy by defining it or by guiding it. It's important that an SDM addresses a strategy or encourages the use of one.

2.4.3 Software Construction

- *Software construction fundamentals*
  Software construction is defined as the detailed creation of working in meaningful software, going through, and combining, the activities of coding, verification, unit testing, integration testing and debugging. This KA is related to all others, as is the core of software development and strongly depends on Software Design and Software Testing.
  When constructing software, one must minimize the complexity of almost every aspect of software construction. The human brain, memorizes very easily simple structures, putting much more effort when leading with complex structures. So the simplicity of the aspects of it's construction are advisable to be as simple as possible, the minimization of the complexity is accomplished by empathizing code writing to be clean and readable instead of very smart and complex. This can be accomplished with the use of coding standards, correct naming of variables and classes as well as organizational code, meaning that within a company people should reuse source packages and styles of programming to increase the readability of the code.
  Another important concept when addressing to software construction is the anticipation of change. It is predictable that software will change

during times, as well as it's requirements, and it's construction should be ready and oriented to change. The verification of software is also an important task when construction software, as it's process should reflect this concern. Therefore, software should be constructed in a way that allows to detect errors and the functionalities to be verified. The previous concept about minimizing complexity aids these two aspects of software construction, when code is simple and easy to understand by software engineers, anticipating changes and verifying the code becomes, also, more easy to perform.

- *Managing construction*
  The construction management is supported by the use of models and methodologies, the main concern of this dissertation, The main goal of an SDM is to manage the construction, of course all the other KA are important but the main focus of an SDM
  The choice of the methodology should empathize the concepts previously addressed: **minimizing the complexity**, **anticipation of change**, **construction for verification** and **the use of standards**.
  Measuring the construction process can be performed, by measuring its artifacts and activities. The code inspection statistics, complexity, effort, faults and scheduling commitment are examples of elements that can be measured.
  Techniques to increase quality of software are various and will not be extended in this document, but they should be addressed by the SDMs.
  The choice of programming languages is also very important for the sake of software construction but the complexity of this choice and the believe of the methodology being adaptable to various programming languages, the theme will not be further addressed in this document.

2.4.4 Software Testing

- *Software testing fundamentals*
  Testing is the activity of evaluating the product's quality, identifying the problems and defects, commonly referred as bugs, and improving the product by correcting them.
  The key issues when addressing to software testing are: the fact that software is *dynamic*, and will not react in the same way to input, and even the same input can produce different output, according to the environment and current state of the product. Software test is *finite*, the possibilities of tests and inputs to perform are enormous and an exhaustive testing,

even in a small program, is not temporal feasible to perform. When performing tests, a software engineer have to select the test set, which will produce a variety of outputs and different levels of effectiveness. The selection of criteria can be aided with risk analysis and other test engineering techniques. The possibilities of acceptance of the outcome of the tests is related to what is expected:

- **Testing for validation**: producing tests for the user to confirm the correct behaviour, normally addresses as User Acceptance Tests;
- **Testing for verification**: producing tests to prove that the implementation fulfills the specification, by trying to perform the functional requirements of the product; and
- **Testing for reasonability**: producing test that are known to produce errors in previous versions or in classical implementations of the specific product its being implemented, and according to the experience of the tester and software developer. This kind of tests are also used to verify that the product has a reasonable response to the input it receives, a classical example is the elbow test where the software is tested to an random input from the keyboard. This kind of tests is commonly used in an ad-hoc way by developers during the development phase. Some authors defend that reasonable testing is providing that all the specifications, and requirements have been fulfilled by the implementation, then considering all the above testing outcome acceptance tests.

Testing has to be seen as an activity to perform not only after the development to detected failure, but as an activity to prevent problems from occur during the development, it is better to avoid problems than to correct them.

- *Test levels*
  Software test can be performed in different stages and with different purposes. The target of the test can be a function a module or the whole system. Testing is also commonly divided by its target, defining:
  - **Unit testing**, where the modules and components are tested isolated;
  - **Integration testing**, where the relationship between the modules are tested concerning possible incompatibilities and the common behaviour of all modules; and

  - **System testing** where the functionality of the product as a whole is tested, after all the above tests and in production environment, or in very similar conditions.

  Testing can also be divided according to the objectives of the test, a test can be used to asset different qualities of the software and a division can be done by testing it's qualities (see section 2.3), as well as other characteristics of the software development process, for example, stress testing, installation testing or recovery disaster testing.
  In products that are target to mass-use, its also common to release some alpha and beta versions of the product to a group of user, normally power-users or even developers, to do the final testings before completely delivering the product.

- *Test techniques*
  The techniques to perform tests are used to increase as much failure potential as possible in the product that is tested. The techniques can go from more empirical, as *ad-hoc* testing and exploratory testing, to more formal, as decision tables or to proof that the product (implementation) implements the specification using calculation. Other approaches are based on the code developed (code-based techniques) or in errors (fault-based techniques). Examples of code-based techniques are control-flow or data-flow techniques and of fault-bases are error guessing or mutation testing.
  Techniques can also empathize on the usage that it will be given to the software (usage-based techniques) or in it's nature, examples are, respectively, operational profiling and object oriented testing. Combinations of different kinds of techniques are also possible and advisable.

- *Test related measures*
  Measures in software testing are a way of determining it's quality. These measures can be related to the coverage of the test, the amount of functionalities that are tested, as for the characteristics of the software according to the test behaviour. The first kind of measures are related more to the quality and extension of the test and to the validation that the test can perform. While the second kind refers more to the quality of the software it self. These measures can be determined by a fault analysis and classification, or by the density of faults in the program. There are several techniques that can be further analyzed in the bibliography [18].

- *Test process*
  When addressing to test processes the attitude of its development rep-

resents a key principle to it's success. The problems of the human ego, related to the non-success of tests, can lead to the misuse of its process, as also to solving its problems. The use of test guides is advisable to avoid mistakes and to provide a mitigation of the risk by prioritizing tests and by serving as a play-book to be performed.
The use of test patterns and reuse of previous test batteries is desirable by its cost and time reductions but it has to be used carefully for it can be inadequate for specific situations.
To decide when to stop testing is also very important issue, because early termination of the testing process can lead to uncovered errors and problems and the delayed termination can lead to not fulfilling schedules and to a waste of effort and resources.
Considering its activities the software testing process can be divided into:

- *Planning*: deals with the allocation and coordination of the personnel as well as the resources needed to perform the tests and to all variables of effort, techniques and reactions to problems.
- *Test-case generation*: Implementing the tests according to the techniques chosen and the expected outcome of the test.
- *Test environment development*: Setting up the resources, hardware and software needed for the execution of the test and also for the logging and tracking purposes.
- *Execution*: The act of performing the tests developed on the previously configured environment. The execution should be performed in accordance to the established procedures and techniques with the results correctly documented.
- *Results evaluation*: The result evaluation is used to asset the quality of the test, that could imply to re-execute the tests and to re-asset the quality of the product that is being tested.
- *Test logging*: The logging of testing activities is a good practice to aid the correction of problems and to have an idea of the current state and evolution of the product.
- *Defect tracking* : To be able to understand where the product has failed, and what caused it to fail, is important to reorientate the project, according to the software engineering practices. It can also provide an aid when solving similar problems in the future.

The testing process is very important in the evolution of a software project, and a lot of techniques and processes exists on this matter, there are even

test oriented development strategies (commonly known as Test-Driven development) where automated tests are developed up front, following a short development cycle which its result is then submitted to the previously prepared automated tests. Also, modern approaches try to perform user acceptance tests, earlier in the project, normally by doing unit testing and providing the integrated behaviour relying on simulation and prototypes, which can help to re-orient the development according to the users input.
Regarding SDM, all address this issue, and they should provide techniques and tools to mitigate risks and to help improve the testing process. With the aim of an SDM being the increase the efficiency of the development process, testing should be one of the processes it should aid by addressing it properly.

2.4.5 Software Maintenance

- *Software maintenance fundamentals*
  As was stated previously, software product is in constant change and evolution. Even the operative system and environment are in constant change. Also when the product is delivered, defects not retracted in previous phases are uncovered and new user requirements appear. The actual maintenance usually starts with the end of a warranty period after the product delivered, where a post-implementation support refines the product until is considered in production, although maintenance activities should start earlier in the process.
  The maintenance is defined as the activities required to provide cost-effective support to a software product. These activities concern not only the modification of software after the delivery, but also the correction of faults, improvement of software qualities and pre-delivery activities such as planning.
  Maintenance sustains the software product throughout its life cycle and the modifications required are registered, tracked and the impact of the changes is measured. The outcome of the process is code, with software artifacts being modified and after properly testing delivered in production environment. Maintenance can also be responsible to provide support and training to users which means, that besides involving tasks of the software development it can have a boarder scope of action.
  Maintenance starting in early phases reduces the effort of the people involved in this process, maintainer may learn from the development knowledge and has to be ready when the developing team is no longer avail-

able to provide aid to this process. Therefore maintenance should be involved throughout the software development life cycle and to take the documentation and code developed and to be evolved progressively with the product. Maintenance should provide fault correction, improvement of the design, enhancements, support the interaction with other systems, adapt the software to its environment (hardware, operative system, network, etc.), migrate legacy software and to sustainably retire software.
The people who perform maintenance activities have to develop different task in order to provide the concepts described above:

- Sustain the control of the software day-to-day functionalities;
- Control and sustain the software modifications;
- Automating and perfecting existing functions; and
- Preventing that qualities of software, such as, performance, don't degrade to unacceptable levels.

Maintenance can be divided into different kinds:

- **Corrective maintenance**: modifications (reactive modifications) performed to correct a problem discovered after the product is delivered;
- **Adaptive maintenance**: modifications that will prevent the software from becoming unusable or incorrect due to environment changes;
- **Perfective maintenance**: modifications that will provide an increase of a certain quality (normally performance); and
- **Preventive maintenance**: modification to detect and correct an potential fault of the software product before it becomes an actual fault.

• *Key issues in software maintenance*
The key issues in software maintenance can be grouped into wider groups:

  - **Technical issues**: referring to the issues based on the technology involved, these kind of issues are related with dealing of the *limited understanding* of a person who was not involved in the development of the software and has to perform modifications in it, *testing* features all over again after the modifications are provided can be significantly expensive in times of effort and resources, specially when concerning integration testing, a modification in a small piece of software can compromise the entire application. *Impact analysis* is also an issue to be addressed when performing maintenance activities, the people who perform the activities must have a strong knowledge of the system's structure and its content. With changes, a risk assessment must be done, determining its scope, estimating the resources needed, analyzing the trade-off cost/benefit and to informing others of its complexity and possible side effects. *Maintainability* (see section 2.3.7) is also a technical issue that affects the process of maintenance and have to be properly control to avoid extra costs. This is specially important during pre-delivery support.

– **Management issues**: these issues are related to act of planning and allocating resources to perform the maintenance as well as organizational aspects. The management should be concerned with the *alignment of organizational objectives*, how the organization will benefit with the maintenance of the product and the trade-off maintenance cost/extending software life. *Staffing* issues are also important, because people who are in charge of maintenance tend to be considered, with prejudice, less-ranked staff. This issue is very relevant, as maintenance staff is very important to keep. They are essential for the survival of the software, and current changes of staff carry cost and effort increase, to provide the required knowledge. *Process* management during the pre-delivery phase is also an issue, as maintenance staff as to be provided with the proper knowledge, but it should not create obstacles to its development speed and efficiency. Also, as it was referred previously, the maintenance process carries tasks outside the scope of development and these activities are a known challenge to the management of the process. *Outsourcing* and *organizational aspects* are issues related with the choice of staffing, the duality between choosing a member of the development team to perform maintenance or someone outside the scope of development. Recurring to people from a different organization but reporting to people within the organization, is also an option to retain. This is normally designated as outsourcing and concerns problems that should be addressed to the management of maintenance.

– **Cost estimation**: this issue is essential to the success of maintenance. The management should be aware of the different categories of maintenance and to be able to estimate its cost. This estimation provides an important input when determining the impact of changes as well the planning and management of its process. There are several models, to estimate cost of maintenance, some more formal, like parametric models and more empirical ways based on the experience of previous maintenance (use the bibliography for further detail [18]).

- *Maintenance process*
  The maintenance process is described as going through:
  - *Process implementation*;
  - *Problem and modification analysis*;
  - *Modification implementation*;
  - *Review and acceptance*;
  - *Migration*;
  - *Software retirement*

  This process addresses numerous activities, divided into main groups: **unique activities** and **supporting activities**. The first is related to the ones that are unique to this KA, (transition, modification request acceptance/rejection, problem report/help-desk, impact analysis, software support and service level agreements [18]), while the second to the activities that considered support to other KA: maintenance planning, configuration management, quality assurance, audits/reviews and user training are examples of such activities.

- *Techniques for maintenance*
  Techniques to aid software maintenance are program comprehension, re-engineering and reverse engineering among others. These techniques refer to understanding the software developed and to then develop the changes. Program comprehension in understanding the software by using documentation and code browsers to aid the software modifications. Re-engineering is the examination and radical reconstitution of the software developed. Reverse engineering inverts the development of the software produced to understand its components and their relationships and to rebuild them from the analysis done.

2.4.6 Software Configuration Management (SCM)

Software configuration management is a support activity within software development where the configuration, defined as the sum of functional components and physical characteristics of the system, is identified and controlled throughout the system's life.
SCM process is characterized by the following activities:

- *Management of SCM process*
  The SCM is responsible for controlling the evolution and integrity of a

software product and the identification of all its elements. To control and manage change, provide verification, recording and reporting of the configuration a comprehension of the organization context, evolving software, hardware and firmware issues strongly related with quality assurance issues and also with maintenance.
There are some constraints when addressing to SCM, such as, internal policies and procedures within the organization where the system is developed.

- *Software configuration identification*
  The identification of configuration items is the base of its management. It should identify the items to be controlled, their relationships and the baseline of the software development, which configuration items will be available at a specific time of the project.

- *Software configuration control*
  Configuration control addresses the changes needed to be done during the product's life and their impact on configuration and all the process taken to evolve the configuration to fulfill the changes as well as tracking and reporting these changes.

- *Software configuration status accounting*
  This activity leads with the information needed to manage the software configuration and with its recording and reporting. The report of configuration status, aids the configuration management. This may imply measuring, for example, the effects of a certain configuration at a given time, and the information produced will be important for assuring the quality assurance of the system.

- *Software configuration status auditing*
  While the previous activity deals with the production of information about status, this deals with the evaluation of the conformance of the system with the guidelines, standards and can be based in different aspects of the configuration (functional, physical or related with the baseline of the software project).

- *Software release management and delivery*
  This activity relates with the delivery of products and trying to manage these deliveries. Building of software libraries and versioning are issues addressed in this activity as well as the management of the delivery, executable product, documentation and all the environment variables needed for the software product to be released.

2.4.7 Software Engineering Management

Software engineering management is an organizational process composed by concepts of process and project management. When dealing with software engineering management KA we can decompose them into the following topics:

- *Initiation and scope definition*
  The beginning of a software project deals directly with the software requirements KA and it is during the initiation that the methods of requirements elicitation are defined and applied retaining the different stakeholders perspectives and therefore to the determination of project scope, objectives and constraints.
  The feasibility analysis is also important to determine that opportunity and conditions are proper for the project success. This often requires estimation of effort and cost as well as that all technical skills and resources are cost-effective available.
  Change is an inevitability of software, specially their requirements, so it's vital for the project sucess to obtain the agreement of the stakeholders and that the scope and requirements are not rigid, and that reviewing and revision should be performed according to a risk analysis .

- *Software project planning*
  Software planning process should receive the input provided for the scope and requirements definition and by the feasibility study, a process is then chosen according to the nature of the project. Resource allocation and equipment are then define and the determination of which deliverables to produce. The reuse of software can also be considered during these stage. Scheduling of the activities is also important to define, determining task dependencies and cost/effort estimation. To correctly perform the planning, an analysis of risks and possible problems should also be part of planning.
  Quality should be defined using pertinent attributes of the specific project and in quantitative and qualitative terms (see section 2.3) and a plan for quality assurance should be done, determining deliverables verification and validation throughout the development process.
  A plan for how the project will be managed should also be delineated and it should fit the process chosen. As the change is inevitable this plan for management should be of extreme importance for the project success.

- *Software project enactment*
  The plan is then putted in practice, and the process should enact the planning. The adherence to the plan is essential for the satisfaction of

the stakeholders and to the validation of the requirements. Implementing the plan is then acquired in reference with the schedule defined and with deliverables to produce. Along with the implementation of the plan, measurement and monitor process should be performed throughout the process, measuring qualities of the product and to monitor the adherence of the resources to the plan. The outcome of the monitoring and measuring should provide input to control the process that will lead to reorienting the plan or introducing practices to increase the quality or adherence to the plan. All these activities should be reported at specified and agreed periods to the organization and stakeholders and should report mainly in the adherence to the plan.

- *Review and evaluation*
  Reviewing and evaluation should be performed at critical points of the projects, normally within releases, and that will review factors as satisfaction of the requirements and evaluating efficiency of the project, the adherence to plan. These milestones review and evaluation should provide the stakeholders the achievements of the project and problems faced during its development, that can aid to the satisfaction of the requirements defined by the stakeholders as well for the stakeholders can unblock some issues and provide guiding to plan modifications and improvements.

- *Closure*
  The closure of the project is when all the milestones addressed in the plan are fulfilled and all the satisfaction and validation from the stakeholders is given. One of the issues of closure is to determine when it's the correct time to define that the project is done.
  And the revision of the work done is then performed to gain from the problems and success of the project leading to an organizational learning and improvement.

- *Software engineering measurement*
  The use of measures should empathize the managing role and develop a commitment for measurement. The staff and managing of the team should accept a guide and the requirements for each measurement. This means that a definition for the scope, which item should be measured, and a plan, defining the criteria and data sets to perform the measurement. These procedures should be used according to the plan defined and properly evaluated. This measurement should be integrated into the process and their evaluation should produce enhancements and improvements.
  Of course this is not an easy task, and SDM is the correct tool to address this measures. The objective of an SDM is in fact to support the

software engineering management in providing the correct approach to impediments by defining a cycle, the evaluation of the activities and to improve the management effect of the project evolution. To evaluate an SDM by verifying that it is providing the correct measurement of software engineering management is not possible without the empirical approach that was putted beside in the goals and objectives of this dissertation. This sub-knowledge area will provide input to asset if an SDM is addressing the software engineering measurements, but it cannot prove that is done in an efficient way, thus that would require an empirical approach, like stated previously.

2.4.8 Software Engineering Process

Software engineering process can be defined as the definition of the software life cycle, its implementation, assessment, improvement, measurement and change. Following we will present these concepts:

- *Process implementation and change*
  This sub-area leads with the organizational change the infrastructure, activities, models and practices of the software have to be well described when considering the process implementation. The infrastructure is essential to the success of process implementation. The resources have to be available and responsibilities properly assigned.
  To manage the software engineering process a set of activities have to be performed in order to obtain continuous feedback of the process and its improvement. These activities are:
  - *Establish Process Infrastructure*: to establish commitment into process implementation and to allocate the appropriate infrastructure.
  - *Planning*: understand business objectives and the needs of the project, to identify the weak points and strong points of the process and to make plan for the process implementation.
  - *Process implementation and change*: the plan defined in the previous activity is executed and to deploy the process or change the existing defined processes.
  - *Process evaluation*: the materialization of the process implementation and change benefits to the project and how the implementation and change has influenced the development. The results are used for the subsequent cycles.
- *Process definition*
  Process definition can be of different kinds, it can be a procedure, a policy

or a standard and they are defined to increase the qualities of the product, enabling communication and understanding among other factors. The types of process definitions required depend of the project context and the reasons to produce it.
Software life cycle models serve as the definition of the phases that occur during the software development, defining its activities and combined with a software process, where these activities are detailed and ordered. A software methodology arises when these are associated with a philosophy, its practices and roles.

- *Process assessment*
  Process assessment is performed using assessment models and methods. Assessment models retain what is considered to be a good practice in terms of different variables, such as technical software activities or management, examples of these models are ITIL or CMMI. To perform the assessment a method is also required, these methods will provide quantifiable scores which will characterize the capability of the process.

- *Process and product measurement*
  To measure activities and software products is not an easy task and it carries a lot of complexity and subjectivity. Process can be measured by its strengths and weaknesses or by the impact it produces on the outcome. This project outcome can be quantified, as an example for faults per KLOC (Kilo Line of Code) or for the amounts of lines coded for each functionality.
  While this dissertation provides a method for choosing an SDM based on the weight given to KA in software engineering and the correct addressing of such, to measure the process and the product obtained after using a certain SDM can help to guide future choices. The methods combined provide decision based on experience and on facts, and a new weight can be used, that is the past experience with a certain SDM that can help to come up with a choice. In fact even previous evaluations using these dissertation strategy can be improved by measuring the process and the final product.

2.4.9 Software Engineering Tools and Methods

Tools and methods are used to assist the software engineering activities during the software life cycle. The tools can be computed-based tools that can simplify some of the activities performed. These tools are widely scoped and they can be divided according to the KA and sub-divided into the activities they intend

to act. Methods are more commonly divided into the kind of approach taken, heuristic approach, formal approach and prototyping approach.
A software methodology can advise the use of tools when referring to its practices. Even though they are not essential to the success of software engineering approaches they provide a good aid to its activities.
The methods are part of an SDM, so they all address methods that support the SDM provided.

2.4.10 Software Quality

- *Software quality fundamentals*
  The objective of a software engineering project is to assure that the product it produces will have the desired quality, not only the software it self but also the activities performed.
  Quality does not come for free and it have associated costs. These costs can be divided into: prevention cost, appraisal cost, internal failure cost and external failure cost. Each product have a predictable cost and associated perspectives for that cost. So the trade-off between value and cost as to be established, these decisions should be done during software requirements definitions, but they are likely to arise throughout the software life cycle.
  Software quality as to deal not only with the quality of the product, but also with the software engineering process quality, both aspects have been discussed previously on this dissertation (see section 2.3 and section 2.4.8).

- *Software quality management processes*
  Software quality management (SQM) is applied to all the aspects of the software process, products and resources. This activity is responsible for the definition of process owners, the requirements for that process and the definition of the process it self. SQM is also responsible for measuring the process and its artifacts and to provide that the the outcome of this measurement is properly channeled throughout the members of the project and the way this feedback is provided.
  Planning for quality involves defining products in terms of the desired quality characteristics and to define a plan to achieve these products. SQM processes can be:

  – *Quality assurance*: assures that the software products and processes are in conformance with the specified requirements. These involves planning and enacting according to what was planned in order to provide and sustain that quality is being built into software;

- *Verification and validation*: strongly related to testing, these activities identify faults and verify if products satisfy the requirements. While verification attempts to ensure products are being built correctly, validation attempts to ensure that the intended product is built according to what was specified and fulfilling the designated requirements;
- *Review*: the review process monitor management artifacts and activities. During the review, technical aspects and the suitability of the software built has to be analyzed and an inspection for software anomalies, or less formally walk-troughs the product, to assure the correctness of its development; and
- *Audit*: independent evaluation of the software according to guidelines and regulations to asset its conformance, these activities are normally done by someone who is not involved deeply in the development of the product in order to the job as with the required independence.

Chapter 3

SDM Descriptions

3.1 Software Development Methodologies

Previously in the introductory chapter a terms elucidation is given (1.3) explaining that durint this dissertation an SDM is a combination between a software development life cycle, also called software production process or model, a set of practices, roles and a philosophy that sustains these concepts. The main objective of an SDM is to enhance a team productivity in order to obtain a product (software) as fast and cheapest as possible. Most production and manufacturing processes are very extensively studied when the objective is to deliver tangible products. To build a product its positive to take advantage of common production processes highly standardized and automated and it's all about making a product as reliable, predictable and efficient as possible.
Although the software process takes a huge amount of intellectual and creative activities, not so adaptable to automation. And last, but not least the product of software is submitted to high instability and requirements changes, therefore obliging the software to be adaptable and to evolve in time.
So the real question is how to make a software product as reliable, effective and predictable as possible without disregarding the unique characteristics of a software product. The differences between SDM are precisely in how to obtain this for a software project. As it was explained in the state of the art section, there have been the need to seek an answer and also plenty of attempts to enhance software production, and it has proven to be a difficult task. The beauty of software relies in its complexity and adaptation, software is almost everywhere nowadays, and to define a methodology that supports such different targets is obviously difficult.
So the first step is to put software into a production line, separate its development into phases, create a software life cycle (although they are not exactly

linear to all life cycles), then to provide roles to do work in these phases and then to provide the proper control for this activities by defining practices and tools that should be used to avoid failure and to increase efficiency. Supporting all this with a philosophy that can convince people to commit to this practices and roles and you have yourself an SDM.
In the following sections we will describe a group of SDM, chosen by its popularity in the academic and industrial worlds and grouped by its similarities in the philosophy and life cycle. The order in which they appear is not random and they should be order chronologically. The given prove that to choose an SDM is not an easy task since that all these SDM are all used plenty used criticized and adored with a lot of successful and disastrous stories to support them. And here is where this dissertation enters and try to help to choose an SDM by giving weights and classifying facts and making a sustained choice.

3.2 Traditional Development Methodologies

The first SDMs are today commonly known as Traditional Development Methodologies TDM. These models were mainly based in defining a Software Life Cycle and how all these stages should be approached. Before the TDM the process of developing software, even if it was in a big project the tasks were performed *ad hoc* without any formalisms or definition of what was being done, this is stated as *code and fix* or more jocosely *cowboy coding* methodology, which is by definition the lack of a methodology. With the inclusion of formalisms the process gain much more effectiveness and correctness but also become more rigid and with less flexibility. These methodologies are supported by a strong preparation before coding and by relying each phase in very tangible artifacts, providing documentation and careful verifications and validations.

3.2.1 Waterfall Methodologies

The waterfall model was first used in late 1950's in a military project called SAGE (Semi-Automated Ground Environment) which was a air-defence software system, but only became popular in the 1970's when it was used as a standard industrial practice and most *state of the art* software engineering textbooks.
The waterfall model is divided into phases, each one structured in a set of activities that can be performed concurrently, and the input of one phase is the result of the previous phase, giving the idea of its name of a waterfall of inputs and outputs being enriched by the activities of each phase.
There are many variations of the waterfall model but they are very similar and

they use the same underlying philosophy. The following phase division will be considered in this dissertation:

- Feasibility Study;
- Requirements Analysis and Specification;
- Design and Specification;
- Coding and Module Testing;
- Integration and System Testing;
- Delivery and Maintenance.

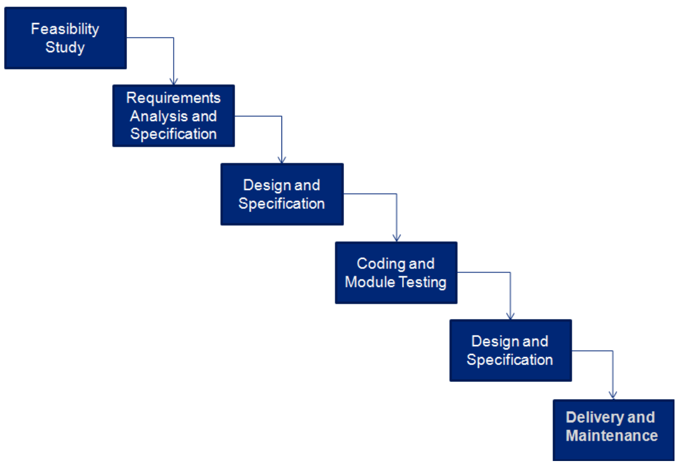


Figure 3.1: The waterfall model

Feasibility Study

This phase is not stated in all variations of the waterfall model, due to not being applied in practice very often.
The feasibility study starts when a problem emerges, the first step into the feasibility study is to understand the problem correctly by analyzing it and finding alternative solutions. Therefore this phase should take into a deep knowledge of the problem which consumes a lot of time (and money) to the people who are undertaking the activities. It's normal that the feasibility study, outputs instead of a feasibility study document, delivers a commercial proposal, where normally only the advantages of undertaking the project are shown. Obviously if a feasibility study determines that the project would not be much of an advantage to the client, the time and effort given to the task are wasted (if a proper

alternative is not achieved). Although, the feasibility study is normally done into short time bounds and in a lot of pressure, it's crucial that a simulation of the project can be studied and that the resources needed and time-lines are estimated.
The result of this phase should be a document stating the definition of the problem suggesting alternative solutions with the expected benefits and challenges and all the costs, dates and required for each of the solutions presented or suggested.

Requirements Analysis and Specification

After understanding the problem, and realizing which seems to be the best solution, the goals of the solution should be very well defined. By this, functional and technical requirements are defined. These requirements should state the qualities stated in the previous section 2.3 with refinements for the solution in cause.
The center of a requirement definition is not how it should be done it's all about what must, should or has to be done. The output of this phase is a requirement specification document where all the requirements are stated, this document should be revised with the customer for it to be clear what the product is suppose to do.
The requirements can be divided into three big types:

- *Functional Requirements*: These kind of requirements specify what the solution should do. this can be done using formalisms or a specific notation, provided that what the solution does is clear.
- *Non-Functional Requirements*: These are the requirements that infer about the quality of the software produced and takes into account operating constraints, performance, etc.
- *Development and Maintenance Requirements*: These requirements take a special account in big customers with quality assurance teams and methods, these requirements are about the procedures the solution must enrol before delivery and in which conditions maintenance will be performed.

Design and Specification

The design and specification phase results in a document describing how the software product should be built. It is based in the requirements defined in the previous phase. It is suggested to divide the software into modules, and is in this phase that the division occurs and the relationships between modules are

defined. The design and specification can be done iteratively going through different levels of abstraction, starting from the point of a formal specification that will be decomposed into an implementation in the next phase. There are many different approaches into writing specifications from the formal point of view to the other more empirical methods. To see more about specification languages and methods see [45], [44], [9] and take a look into correctness quality in section 2.3.1.
It is also a suggested that a company should use a standard on specification and design documents, regarding how appropriate the standard is for the specific solution in which it should be used.
In short words the design and specification is like the project of a house or a roadmap of a trip, its the guideline of the implementation, and should be as clear as it is possible. The process of design and specification can be called *disambiguation* where vague terms will be translated to terms that a programmer would not have doubts into converting to a programming language.

Coding and Module Testing

Coding is the part where the software is actually built. This phase translates the design and specification document of the previous phase into a programming language, as in the previous phase the notation and standards are advised. If it was suggested in the previous phase that the product should be split into modules and if the implementation follows the specification the software should be coded into different modules. These modules should be tested individually before being integrated in the next phase and all the tests should be documented. The module testing should also occur in this phase, which should be the first line of testing, standard in module testing is also advised and a line plan of tests where the kind of tests to perform should be explicit (e.g. white-box, black-box, unit testing etc.). The inspection of a well written code and the qualities of software are also desirable some of them taking more relevance in later stages of the coding process. Some authors suggest that correctness should not be verified in this phase, although correctness of the product cannot be proofed from one module, if the specification and the functional requirements are also designed in a modular way, module correctness can be measured.

Integration and System Testing

In the integration and System Testing, the modules built in the previous phase are assembled and is sometimes included into the coding phase. Although integration requires different kind of coding and testing. The testing in this phase

should assure the relationship between modules, as defined in the specification, and integrated tests should be performed, the module testing does not assure that the system testing. During this phase the software system should be tested in the environment that it is supposed to work and also the integration with the system where it will work (hardware, operative system, users, etc.).
The output of this phase should be a produced that can be used, and can be shown to the users and the so called *alpha testing*, tests under realistic conditions should be performed. It is also suggested that the integration tests follow a standard within an organization. The integration can also be done according to standards (such as top down or bottom up) and documentation of the tests performed should be done.

Delivery and Maintenance

In the delivery phase, the application is ready to use before the beginning of the phase, and the delivery is normally carried out in phases, to minor changes and the impact of errors and misunderstanding of requirements.
It is normal that a controlled experiment is carried with a group of prior users, that will provide output of the system, often called *beta testing*. During these activities is nowadays common to put professionals from social areas with technical professionals to assure that the software fits all the required qualities, some complex systems also require people that can explain better to users than the people who have developed it.
After the product has been submitted to the *beta testing* if it is not reverted to an early phase it will be released to the customers. After the release the maintenance of the system is essential, specially when requirements change fast and the system have to evolve fast as well. The maintenance is by that responsible for correction of missed errors and to adapt the system for the current needs. Maintenance within the Waterfall methodology goes through repeating the life cycle for the modifications to be done.

Critic to the Waterfall Methodologies

The Waterfall main principle can be "*measure twice cut once*" which is its most loved and hated principle. The benefits gained from the use of the Waterfall Method are unquestionable because it brought discipline and method that was most needed when the method was created. It brought discipline, planning and management to a world where coding and fixing was the main strategy. It is calculated that a project costs much more when late requirements are discovered. The Waterfall method is categorized as Big Design Up Front BDUF, where first is well decided what/how and if it is worth to be done ("*measure*") and then

start coding the product ("*cut*"). The Waterfall method is an ideal model, and obviously can only be approximated and the discipline and perfectionism defended by the method is also its worst critic. Some critics suggests that Waterfall is too rigid and that most clients/users don't realize some of the requirements until late in the project, so a project should not *waste* a lot of time in the pre-coding phases, on the other hand *climbing up* the Waterfall is much more expensive than going through the phases. It is also suggested by some authors [15] that the Waterfall method to be effective should be used with the feedback loop between phases, so that a mistake in the beginning is not postponed to later phases and that the linearity of the process is the main key of success when using the model.
The principle that the balance between resources and requirements are difficult to define and that to use documents/specifications are not mutable according to the software development is a static document that can be changed, but according to the model, to change it, it would imply going back to a previous phase is also a common critic to the model. Also that the *real life* projects are in constant change and requirements change more often than predicted and that the monolithic design of the method is not sufficiently agile to adapt to those changes.
Even though its critics Waterfall is still considered as one of the most important methods which as brought a lot to the software industry, and it is fact a reference still today when approaching Software Methodologies. It can also be considered as the father of TDM.

3.2.2 Transformation Methodologies

The transformation model is rooted in mathematical and theoretical work on turning specifications into implementations. The model is strongly connected to formal methods of specification. With this method a specification should be as abstract as possible and then refined into an implementation. This model is not used often in industrial software, besides in some cases in the called critical software, and it can be viewed as a step by step refinement of the specification, and on each step the level of abstraction is lowered until it becomes executable. The process is lead by proving its correction from the previous step, normally using formal mathematical proof. Optimization of the implementation should be performed and the reuse of software components can be part of the project ensuring that the reused material is formal verified against the specification. The refinement process is done with the use of algebras and co-algebras and through calculation. These activities can be helped with the use of proof or verification language (or even specification languages [9] and[45]). For more information on formal methods and program calculation please see the bibliography [30] and [31] (in Portuguese). During the development process there is also a place for redevelopment or the adaptation of new or future requests, but the development would began in the point where is appropriated and will accommodate the changes to the verification and correction of the software.

Critic to Transformation Methodologies

The transformation model is still very research oriented and it requires a lot of familiarity to mathematical and formal methods which may not be suited for large projects were the requirements analysis is very difficult even using more informal processes. One of most pointed out issues is concerned with change of requirements which must be changed in the initial model and if a change is done it has to be drilled down until the final the implementation because all implementation has been calculated/proven from the refinement of the specification. So a small change in the requirements would affect all the work, and all the verification and proof of correction had to be redone. Even though the transformation can bring the correction to the equation of software development and all the process of verification, if done correctly can proof that a software *implements* the specification.

3.3 Evolutionary Methodologies

3.3.1 The Win-Win Approach

The evolutionary methodologies are based in the evolutionary life cycle, this model is an incremental model where the new increments are developed in a reactive way. This methodology was raised by the awareness that first versions lead to re-work or to high rates of adaptive or perfective maintenance. To prevent the need of extra development and analysis in later phases of a project, this methodology bring the stakeholders to evaluate and verify the software in early phases. So the first versions of software, often just prototypes are released to stakeholders (users, customers, etc.) to test the product and with their input an increment is specified, implemented and then integrated with the previous solution. First versions might be even *thrown away* if the increment does not fit at all with the previous version, causing that it cannot be integrated. This philosophy of development advocates the principle of evolution by *operational experience*. This approach disrupts with the *measure twice, cut once* advocated by the waterfall methodology and one of its mentors Tom Gilb [17] mention in his book *"do it twice"*. This method could be confused with *cowboy coding* or ad-hoc development, but instead it relies on specific activities that should be

performed and it provides practices of management and documentation of every step taken. The solution is properly specified, according to the philosophy of the methodology, but solutions are proposed and negotiated instead of decided and proven. The requirements analysis are then mixed with the development which is refined taking advantage of modeling and prototyping. Tom Gilb defined this strategy has *Win-Win*, because the customer wins, because it gets what it wants, with the input, and the development team doesn't need to perform rework on the final implementation, they are changing the development as it goes.
The software life cycle of this methodology may vary from author to author, in this dissertation the following is presented:

Requirement Analysis

In this phase the requirements are analyzed. The methodology defends that the common approach of *"How"* and *"What"* should not be used and advises a *"How well it must do what"* approach. This means that normally during this phase people are worried in finding all the functionalities and how to implement them and they should be also focused on defining the proper quality for each functionality. Then for the requirement analysis a specification is delivered explaining the main functionalities to support the first version of the product (or the first prototype).

Deliver

The deliver phase is where artifacts are released to the client, this means that after the requirements analysis the specification is released with a prototype of what the product will be. The deliver phase occurs for all the artifacts in the project, including all the versions of the software. This phase is characterized by small periods of development so the stakeholders can be almost constantly providing input and feedback to the team. Each deliver is designated as an increment and after validation it should be integrated with the previous releases.

Measure

In this phase an input by the users/customers is given by measuring the operational value of the software (in all critical aspects and qualities). This phase can be done with user acceptance tests, or in more informal ways by showing the artifact and request input. This phase also should be done to all artifacts, and occurs immediately after the deliver phase. The objective of this phase is to identify requirements that weren't correctly identified or interpreted by the team, and it should be intercalated with small periods of development so the changes identified don't imply massive changes in the products. This measure also defines that the product is in conditions to terminate the development of the product.

Adjust

This phase occurs just after the measure and it should be followed by a deliver. It is characterized by the development of the changes identified during the measure phase, with the incorporation of the feedback provided. This feedback should be systematic to produce the work that it has to be done by the development team. This phase shouldn't be very long and it should be delivered to the client as soon as is in conditions to be measured, otherwise, the measurement phase can result on a waste of time as resources for development. This means that a modularization of the product is also advisable so they can be measured individually. This approach can be adapted to measure integrated artifacts or only modules and then later the integrated is again submitted to the different phases.

Terminate

This is a final phase and is achieved when the solution delivered fits into the needs of the client and no more development, besides maintenance, is needed. This means that the product delivered fulfills the objectives, always considering that it does as well as it should (according to the previously stated *"How well it must do whatâ* requirements), and no more adjustments are needed. Obviously, and according to what was stated previously, this phase can occur only to a module, not implying that the project has terminated but that the specific module is done and should only be submitted to the process when the integrated solution is delivered. This is always a critical part of the project and it's always hard to terminate a project and an approach regarding the quote *"perfect solutions are enemies of good solutions"* should be applied. In a software product there is always space for improvements and the difficult task is to realize when is it good enough (again the same issue stated in the requirements), and that's why this methodology states that during the requirement analysis

To see more on evolutionary methodologies please see the bibliography [17] and [16].

Critic to Win-Win Approach

The Win-Win approach, and evolutionary methodologies, advocates the use of prototyping and reflects the need for the user and client to be involved in the development process and can decrease the time of revising the requirement analysis before having anything tangible to work with. The cost of throwing away prototypes is not very well understood by the majority of software vendors and resistance to change is also common within clients, so these problems should not be neglected. If the requirements are not well understood from the first prototype and to actually throw everything of the first version is very hard, and from the point of view of the developer is not easy to understand and accept that the software produced is going to be disposed and can cause some issues within the team and a resistance to adjustments. Another issue is related with the common practice of using prototypes to sell the idea to the client that the software will fulfill the requirements, which can cause a deviation from the philosophy of the methodology. One of the most positive criticism to evolutionary methodologies is the use of prototyping as a good practice to detect early modifications and needs of change, and using this with disregarding documentation and to empathize the requirement analysis. This methodology is still used (or its derivatives, like the spiral methodology) and is defined by removing some of the rigid development of waterfall (section 3.2.1) without going deeply into more flexible processes like agile methodologies (section 3.5).

3.3.2 Spiral Methodologies

The Spiral Model was first defined in an article by Barry Boehm [6] and where it introduced an effort to combine the advantages of top-down and bottom-up strategies (see table 3.1). To achieve this, he combined the use of prototyping and design. A prototype is built in a top-down approach, each step is inserting more detail into the specification, until a operational prototype is released, then the the implementation is performed by picking up this operational prototype and defining a detailed design of the product that will lead implementation, implying a bottom-up approach of this process.
The spiral methodology gets its name from the Cartesian diagram representation with a spiral with center in the point (0,0) of the diagram. The radius of the spiral is the representation of the accumulated process cost, that should be followed by the review effort, and the angular dimension its progress (see Figure 3.2).

The spiral methodology can be seen as a derivative of the evolutionary methodology, where an iteration of the cycle is based on the result of the previous one. The change from the original methodology is in its risk analysis in the measure phase (see table 3.2) and the combination with a systematic control in a linear sequence (like waterfall 3.2.1). This leads to the fact that Spiral Model evolution it's cyclic. This model emphasizes the design process for an identification and

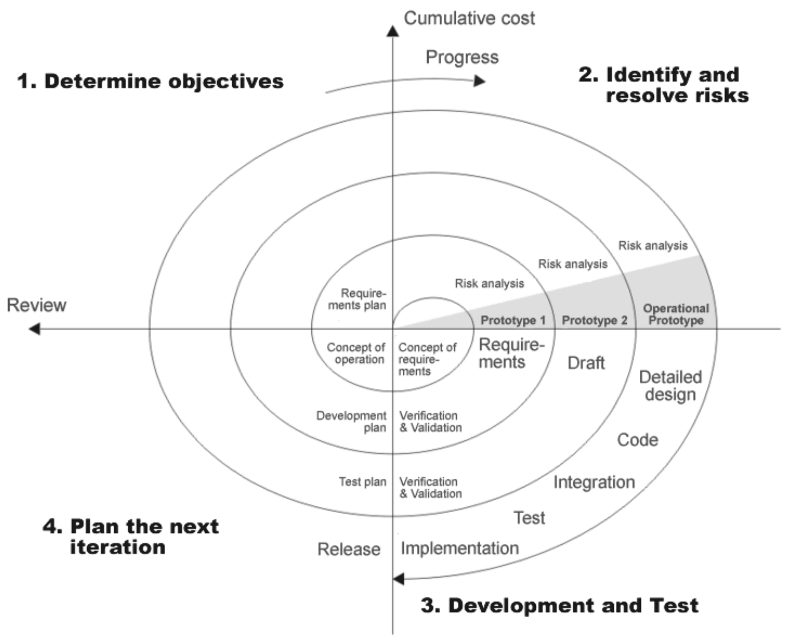


Figure 3.2: Spiral Model, Boehm,1988 (CC)

Top-Down	Bottom-Up
This strategy is based in beginning into a very not detailed analysis of the system, and then refining it into different sub-systems into greater detail going through it until the entire definition is reduced to a set of base definitions. The theme words for this strategy is analysis or decomposition.	This strategy is almost the reverse of the previous one. The definition of the system is began by its base elements, and then trying to linked them into a greater system (inferring the system, by a set of base elements). The theme words for this strategy is synthesis or induction.

Table 3.1: Top-Down and Bottom-Up Strategies

“(...)discipline whose objectives are to identify, address, and eliminate software risk items before they become either threats to successful software operation or a major source of expensive software rework“ - Boehm 1989 [7]
”Risks are potentially adverse circumstances that may impair the development process and the quality of the products.“ - Fundamentals of Software Engineering 1991 [15]

Table 3.2: Risk in software projects

mitigation of the high risk problems and a differential reaction for the severity of the risk in cause.

This methodology phases are better understood when placed in the diagram represented in figure 3.2, but a description of each phase is presented next:

Determine Objectives

This phase is characterized by identifying the objectives of each iteration and to define a requirements plan to build the project. Its main concern is the review of work done, and as stated previously it should increase in each iteration. In this phase a requirements plan is elaborated to guide the iterations of the project.

Identify and resolve risks

Mainly characterized by the implementation and analysis of prototypes. Each iteration should provide a prototype lead by the previous risk analysis performed. This risk analysis should determine what should be changed from the previous prototype and what should be done, to minimize the risk of project failure, in the next prototype. When a prototype is ready this phase terminates. This is similar to the measure phase combined with the adjustments that derive from the measurement. The implementation of a release begins with the operational prototype, delivered in this phase.

Development and Test

The development and test phase is responsible for delivering artifacts and for its verification and validation. The concepts of requirements are defined in this phase and after the first prototype the requirements are defined also in this phase. Then for each prototype presented a draft product is verified and validated until, with base on the operational prototype, the detailed design is presented. With the detailed the design the development of code, followed by integration and test is done until the implementation is achieved.

Plan the next iteration

As all the previous phases, this phase starts with the results from the previous one, so with the validation and verification of the requirements this phase will produce the concept of operation. With the verification and validation of requirements the development plan is also done in this phase and for each draft, and its validation/verification, the test plan is elaborated until the release is delivered. This phase is also responsible for feeding the first phases, determining the cycle iteration, for each plan elaborated the determination of objectives and risk analysis is performed supporting the new prototype and so on until the release that terminates the spiral.

Critic to the Spiral Methodology

The Spiral Model has introduced the risk analysis into the planning a software product which the earlier process model did not have. This measure occurs in a natural way in a real-life situations but it was not considered as a part of the management and planning of the project, was just performed taking intuition and empirical knowledge. This and a strong versatility of the model are the great advantages of using the spiral model.
Although the risk analysis can be hard to perform and if a high risk is measured as a low or not important (or the opposite) the development process can be easily become an disadvantage. Also the misuse of the spiral iterations can lead to a waste of time if the risk, tracking and control are not performed carefully. Also from the point of view of the contractors you can have an image issue, where the work performed it's not very tangible from his point of view in early stages of the spiral, but that can be mitigated by using the clients into the process of risk analysis by, for example, performing user acceptance tests with the prototypes delivered (like in evolutionary methodology 3.3).

3.4 Rapid Application Development

3.4.1 Changing Plans

This methodology represents the shift on setting the importance in the planning phases for the implementation this methodology made the introduction for the agile methodologies to arise. Before this methodology a great effort of the software development process was in planning.RAD (Rapid Application Development) opened the door for less planning and more action philosophy, defended by the agile development methodologies. Although other methodologies could be used to represent this step RAD has been choosen for its relevance and use in industrial and academic environment.

3.4.2 RAD

The Rapid Application Development was first published in a book by James Martin in 1991 [28], although it was based in previous works of Scott Schultz who created a methodology called RIPP (Rapid Iterative Production Prototyping). The base for RIPP was the work Barry Boehm ([6]) and Tom Gilb ([17]) the creators of evolutionary methodologies, Spiral (section 3.3.2) and Win-Win (section 3.3.1), respectively. The believe that the later methodologies (such as waterfall model 3.2.1) were taking too long to develop, that by the time they were finished the requirements had already changed, which resulted that the system built, especially in large projects would not correspond to the current needs. Scott Schultz was the first to introduce the term *"timebox"* (it was in fact a concept used also by Tom Gilb but without actually using the term time-box [17]. James Martin took the RIPP and formalized it and extended with values from other development models and created RAD (he called time-boxing a variant of RAD in his book [28]). These methodologies started the rupture with the so called TDM, they were the first to evolve from the monolithic and linear way of developing software (RAD is also often called the father or grandfather of the Agile Development Methodologies ADM).The work of James Martin was also formalized by other authors, also with the interaction of James Martin. This methodology was a reference by he introduced and used some concepts that are the core elements:

1. **Prototyping**: Prototyping was the base of the evolutionary model (section 3.3), but in RAD the construction of prototype is used with the purpose of kick-off for design. The objective is to built, in just a few days, a light version and serves as proof of concept for the client and as a point for the discussion with users and client for refining the requirements. This prototype is advised to be accomplished with the use of CASE (Computer Aided Software Engineering) tools, that focus in capturing the requirements, converting them into a data model and finally converting into code and auxiliary components (such as databases).

2. **Iterative Development**: This resembles to the spiral model (section 3.3.2), but the length of each iteration has been revised to a short period (one day to three weeks top), which represents the closeness to agile methodologies (section 3.5). The feedback for each iteration is crucial to the success of the method. The feedback is used (as in most development methods) to provide refinements of the requirements and to control and manage the progress of the project.

3. **Time Boxing**: Is a time management technique use to plan deliverables in short time periods. The objective is instead of delivering an complete future of version, it's to split the development in goals within a time box, with a period no longer than six weeks long. For each time-box a budget and a deadline is defined. The use of strict time boxes is one of the main

principles of RAD and assures the length of a development iteration and it can manage the expectation of the client according to the deliverables. The misused of time boxing can derive the iterations of the project to be futile and to roll back to a linear development.

4. **Team Members**: The RAD advises the teams to be small and experienced and where the client performs an important role in the development process specially during the initial and final period of an iteration. Team members should be able to perform multiple roles not to disregard important aspects of the system and are organized into called SWAT teams (Skilled Workers with Advanced Tools).

5. **Management Approach**: Management should be as active and involved as possible, it's one of the most vital activities in this methodology to asset risks and to mitigate them, specially when considering long development cycles. Also the management is important one balancing clients misunderstandings and on motivating the team. It should emphasize a strict control of time-lines and on removing the political and bureaucratic obstacles from the developing team, nevertheless keeping the team aware of the critical points and major risks.

6. **Tools**: Obviously when the method was formalized, the tools used are nowadays almost obsolete although others evolved, but the major point is that the RAD methodology relies on the use of state of the art tools (such as IDE's and 4th generation languages) to speed up the development process.

There are many divisions in phases of the RAD process, here the approach of James Kerr and Richard Hunter [21], will be considered for its popularity among industry and academic players in software engineering.
Kerr divides RAD into 5 phases as shown in figure 3.3:

1. **Business Modelling**: This phase is characterized by understanding the functional operations demanded by the business which the system will serve. All business process are cleared during this phase and modeled during this phase. This task is necessary for understanding not only the functional requirements but also all the business where the system is involved to mitigate probable change of requirements and the need of new functionalities.

2. **Data Modelling**: With the output from the previous phase, then the need of technical resources is needed, the business is understood and there

is the need of understanding where the system is needed and which information is needed for the system to work. During this stage the data flow of information is modeled according to the business model previous defined.

3. **Process Modelling**: After understanding where is the information and how can the system get it, the need of understanding what should the system do and perform with the information provided. The flow of the information within the system is then defined and concrete data objects that will be processed by the system.

4. **Application Generation**: The application generation should take advantage of tools available to the project to reduce the development to the minimum, and the reuse of software is encouraged for its reduction on testing. Automation tools are also strongly advised to enhance the development and generation.

5. **Testing and Turnover**: All new components should be tested and integration testing is essentially for the success of one deliverable. As the reuse of software is encouraged, the testing can be minimized to integration tests (understanding that the component being reused has been tested). The called turnover is used for modifications and changes detected in the final testing and to deliver a release for the focus group sessions to evaluate the state of the project and the deliverable it self.

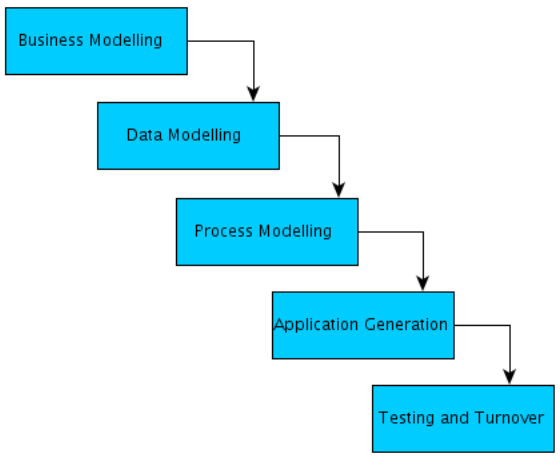


Figure 3.3: Rapid Application Development

Critic to Rapid Application Development

This methodology represented a cut with the previous methodologies without disregarding the important features in older development models. It introduced some new practices and reused others, such as:

- The use of time-boxes;
- Combined risk analysis (as address in the spiral model 3.3.2);
- Prototyping (also present in evolutionary methodologies 3.3)

One of the innovation brought by RAD was the effort to reduce the time used for each iteration as most models only focused on reducing the overall time taken in the development. This can be viewed as a divide and conquer strategy, reducing each iteration for reducing the overall activities and therefore the project. For this purpose, RAD encourages the use of modern technology and management techniques.
The main criticism to RAD is its feasibility in large projects. The encouragement of reuse, is also cited as a high risk for the success of innovation projects. Another common criticism to this methodology is the difficulty of its implementation, and the fact that if is not properly used it may lead to a chaotic development.
The methodology is also criticized by not addressing properly the performance software quality, because of its goal to reduce time of development specially when developing critical software that can require an extra formal analysis and cutting edge science, although this criticism can be rejected if the testing and turnover phase addresses all software qualities.
The RAD methodology represents a mark in the history of SDMs and it is the connection between the Traditional Development Methodologies and the Agile Development Methodologies, together with the evolutionary methodologies, it opened the door for new approaches to arise, and most of the new methodologies that did, were inspired by RAD. This methodology is still used by some major software companies and is one of the most quoted methodologies both in academic and industry papers about software development methodologies.

3.5 Agile Development Methodologies

The ADM (Agile Development Methodologies considered the modern wave of software development), were based strongly in iterative development as the later TDMs started to put in practice. The term agile development was first used in agile manifesto (3.5.1), by the year of 2001. The main difference of these new methodologies was the self-organization of the team (one of its ruptures with RAD) and the use of management skill within a cross-functional teams (one of the resemblances with RAD). During this section some of the most notable methodologies will be addressed, regarding the popularity and relevance.

3.5.1 Agile Manifesto

In the year of 2001, a group of representatives of the new wave methodologies gather in Utah (United States of America) in a ski resort to discuss the need of *lighter* software methodologies opposed to the *heavy monolithic* traditional methodologies. By the end of the gathering they deliver a manifesto addressing a statement of principles that they (17 authors) subscribed. The full transcript of the manifesto can be read in table 3.3. With this gathering the Agile Alliance was created and they also added to the manifesto a statement of twelve principles of agile software (table 3.4). To see more about the agile manifesto please check the bibliography [42].

”We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

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

That is, while there is value in the items on the right, we value the items on the left more.“

Authors:

Kent Beck	James Grenning	Robert C. Martin	Mike Beedle
Jim Highsmith	Andrew Hunt	Arie van Bennekum	Ron Jeffries
Ken Schwaber	Alistair Cockburn	Jon Kern	Jeff Sutherland
Ward Cunningham	Brian Marick	Dave Thomas	Martin Fowler
Steve Mellor			

Table 3.3: Agile Manifesto

"We follow these principles:

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

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

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

Business people and developers must work together daily throughout the project.

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

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

Working software is the primary measure of progress.

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

Continuous attention to technical excellence and good design enhances agility.

Simplicity–the art of maximizing the amount of work not done–is essential.

The best architectures, requirements, and designs emerge from self-organizing teams.

At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly. "

Table 3.4: The Twelve principles behind the Agile Manifesto

3.5.2 Scrum

Scrum in a known rugby formation and it was the theme for a work from Takeuchi and Nonaka in an article [43] on Harvard Business Review. The analogy with the rugby formation technique, where the whole team achieves distance by passing the ball back and forth as a a single unit, was made from the fact that the approach to deliver a new product was made by a cross-functional team going through overlapping phases. The analogy used to describe the *old* approach was a relay race where the different teams are passing the baton to the next group.
The cross-functional team should be able to self organize within the project, with the proper control being performed subtly and should be able to learn different areas taking advantage of the knowledge resources within the organization where the project is being held. In consideration to the development it should be performed in overlapping phases and with the called built-in instability. The built-in instability consists in creating a tension to promote creativity, the goals for the new product should not be very strict neither should hand out clear-cut concept of the new product. By that instability teams have a wide measure of freedom and challenging goals.
In the early 1990's Ken Schawber, Jeff Sutherland, John Scumniotales and Jeff Mckenna start exploring this kind of approach in the industrial software. Schawber and Sutherland then presented a paper describing Scrum. They started working together in refining the methodology, by their experience in the software industry and come up with the methodology that is now widely known as Scrum. Ken Schwaber with the help of Mike Beedle then formalized the methodology in a book in 2001 [36].

Roles

Scrum has distinctive roles within the teams [35], they divide the roles into two different kinds: the *pig roles* and the *chicken roles.* The division of the roles is based on a joke about the creation of a restaurant by a chicken and pig:

The roles in Scrum are composed by:

- **Scrum Master**: The Scrum Master is responsible to ensure that the project is enrolled according to the principles of Scrum, and to remove the obstacles to keep the team focused on the tasks and to organize and coach the team in fulfilling the Scrum principles. The Scrum Master is

A chicken and a pig are together when the chicken says, "Let's start a restaurant!"
The pig thinks it over and says, "What would we call this restaurant?"
The chicken says, "Ham n' Eggs!"
The pig says, "No thanks, I'd be committed, but you'd only be involved!"

Table 3.5: Pig and Chicken roles: the joke

not the leader of the team neither manages the team, the team is self-organizing, the Scrum Master just helps the team understand and use the principles of self-organization and cross-functionality. The Scrum Master is also responsible for choosing and teaching the Product Owner to perform his work according to Scrum. The Scrum Master can also be a member of the Scrum Team but it can never be the Product Owner.

- **Product Owner**: The Product Owner is entitled to assure the quality of the product and is responsible for managing the Product Backlog, where a set of priorities is defined, and to make it visible to all the team. This set of priorities is only defined by the Product Owner, he and only he can change the Product Backlog, if some member of the team thinks some item need a change of priority he has to convince the Product Owner, and the team should only work according to the priorities defined by the Product Owner. The Product Owner can be a team member, but it can never be the Scrum Master. This role is very visible and requires responsibility, being a demanding nevertheless rewarding role.

- **The Team**: The team is responsible, with the input from the Product Owner (Product Backlog), to put the requirements in a deliverable in each iteration (called sprint in Scrum). Team members can have specialized tasks and activities but cross-functionality is required and they should learn from each other, and even perform tasks they are not used to if required. The team has no titles within, and no one should refuse to do a task because it's not their job. Scrum strongly refuses the creation of sub-teams with specific tasks, and the team should not be very big and as a optimal size of seven people (plus or minus two), Scrum Master and Product Owner are not included in this count, unless they are also part of the team. The team should be self-organized and not even the Scrum Master should tell the team how to convert the Product Backlog into a deliverable or an increment by the end of the sprint. Changes within membership in the team are not advisable, because can affect the self-organization of the team and therefore affect the productivity and these changes should be done in the end of a sprint.

Using the previous analogy of the chicken and pig, the Scrum Team members are pigs and all other roles are chickens. The Product Owner and Scrum Master are considered to be chicken, but how was said before, they can also be a Scrum team member, so they can also be pigs.
One of the rules in Scrum, is that chickens cannot tell pigs how to do their work, by then, only if the Product Owner and/or Scrum Master are involved in the deliver of the product as part of the Scrum Team (pigs), may they interfere in the way pigs perform their tasks [35].
There are also other roles considered within the scope of Scrum, considered to be chickens, such as managers who will be responsible for creating the conditions for the project to be held and the clients and users of the product normally designated as stakeholders.

Practices and Life Cycle

Scrum can be divided into three major phases *Pre-Game*, *Development* and *Post-Game* that will be now analyzed with more detail.

1. Pre-Game

In the pre-game phase the planning and conceptual architecture are defined. This phase consists in preparing the requirements analysis that will be used to construct the Product Backlog. In this phase, are also chosen the tools that are going to be used to develop the product, the composition of the project members and their roles. Also, a risk analysis and global costs are calculated to asset the viability of the project. This phase involves the called chicken stakeholders. Each requirement defined goes into the Product Backlog, the document that will be managed by the Product Owner, defining the priorities for each one of the requirements. This requirement list will be continuously updated during the development phase. The pre-game is meant to provide the input for the development phase, that works like a black box, the stakeholders define the team, requirements and tools to the development phase and by the end of the development phase a product will be released.
The conceptual architecture is also defined during this phase. A meeting with different approaches should be taken and a conceptual approach should be chosen and this architecture should be based in the requirements that are present in the Product Backlog at this point.
The creation of the **Product Backlog** consists on defining an priority and effort estimation for each requirement, this should happen in the Release Planning Meeting.

2. Development

The development phase is the phase where the development is actually performed. This phase is considered to be a black box, in which changes of requirements are predictable, and where the called pig elements of the project appear. The development phase is divided into iterations called **Sprints**. Each sprint, with a time-box between 2 and 4 weeks, should end with an some increment to the product. A sprint starts with a **Sprint Planning Meeting**, time-boxed to 8 hours for a 4 week sprint (with the rule of 5% for the total amount of the sprint). This meeting is divided in two parts: the *what* part and the *how* part, the meeting is conducted by the Scrum Master and the Product Owner presents the priority list as in the Product Backlog. The parts are time-boxed to half the time of the meeting (although the parts can be combined). The input for the meeting is the Product Backlog, previous sprints increment, the capacity and previous performance of the team. The team chooses an amount of the Product Backlog and a **Sprint Goal** is crafted defining the objectives for the sprint, which is represents a subset of the release goal, this is normally done in the *what* part of the meeting. In the *how* part of the meeting the team decomposes the part of the Product Backlog chosen into tasks and activities that should be performed in order to obtain the Sprint Goals (converting the subset of the Product Backlog into *tangible* software). The task list is called the **Sprint Backlog**. The assigning part can be performed during the meeting or during the sprint it self, it's up to the team, advocating the self-organization of the Scrum Team. The Product Owner should be involved in the meeting to clarify the Product Backlog and to help in the negotiation of trade-offs. The Product Owner may also reconsider the priorities in the Product Backlog.
The sprint is then started and during the sprints, there are meetings, time-boxed to 15 minutes, called **Daily Scrums**, in which each member explains:

- What he/she has accomplished since the last meeting;
- What he/she is going to do before the next meeting;
- What obstacles are in his/her way.

These meetings should take place at the same time and in the same place every day. The Scrum Master conducts the meeting and assures that it happens and happens within the correct time-box. The purpose of these meeting is to improve the communication in the team and to eliminate impediments to the development.
During the sprint are also some artifacts that are present to the all team:

- **Release Burn-down graph**: represents the amount of the Product

Backlog estimated effort across the time. The units are usually Sprints, but it's up to the team to decide how it should be measured.

- **Sprint Burn-down graph**: As the above it represents the amount of the Sprint Backlog to be done, it's advised that the Sprint Burn-down graph should be a physical one, that it should be visible in the project area, for motivational reasons.

In the end of the sprint, a **Sprint Review** meeting is held, with a time-box of 4 hours meeting for a month sprint (should be half the time of the Sprint Planning Meeting). In this meeting the stakeholders should also be present to asset the increment and the Product Owner should explain what has been done according to the Product Backlog and all the items that are done, and the revisions that have been performed to the Product Backlog and estimation of the completion dates and velocity assumptions (how much Product Backlog effort can a team handle for each Sprint). The team also discuss the problems they had, how they solved it and what went well. The team then presents the work done, and the meeting should provide the input for the next Sprint Planning Meeting that will begin the next sprint. Before the next Sprint and after the Sprint Review, a **Sprint Retrospective** meeting is also held, for the Scrum Team to revise their organization and development practices. This meeting is time-boxed to 3 hours and discuss not only technical aspects but all the aspects that can contribute for an increase of effectiveness for the next sprint.

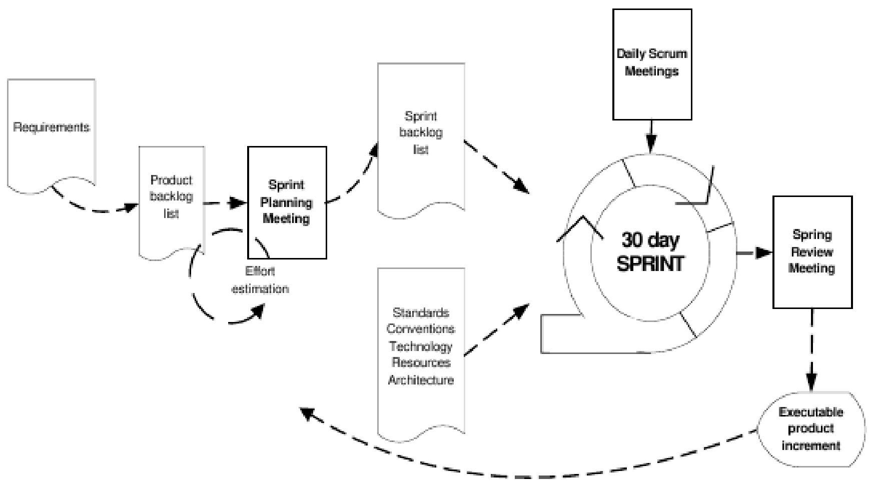


Figure 3.4: The Scrum Sprint [1]

3. Post-Game

The final phase starts when all the goals defined in the Pre-Game phase are achieved, the preparation and creation of the final release, including its tests, manuals and learning material. In this phase all stakeholders should be informed of the end of the project. The tests to perform should include integration and system testing. The definition of done is defined by the Project Owner, meaning the conclusion of development, that can include, tests, documentation and other kind of requirements such as internationalization issues, and it is the Product Owner responsibility to clarify what should be done before releasing the product to the final user.

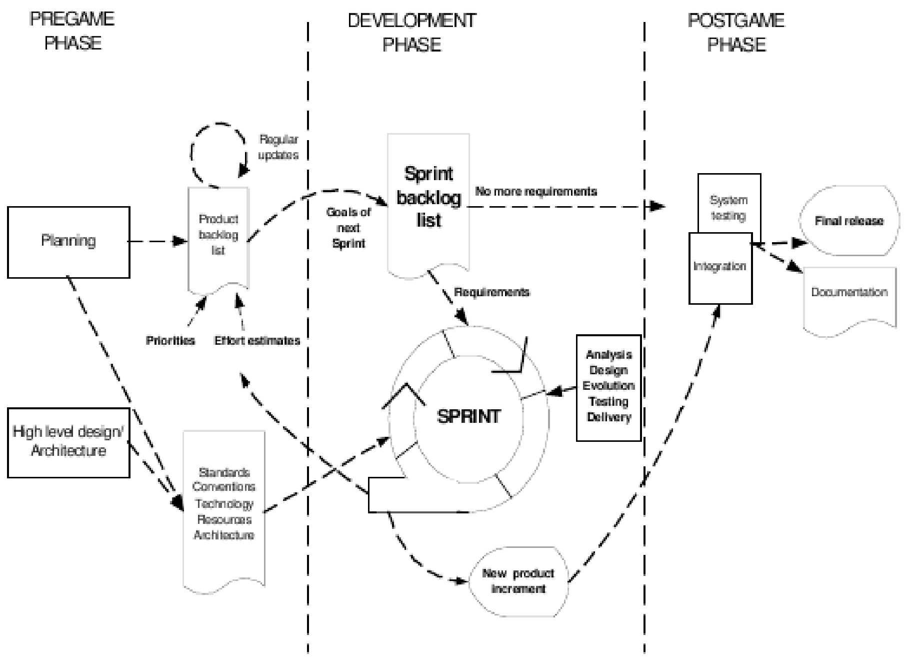


Figure 3.5: The Scrum Life Cycle [1]

Critic to the Scrum Methodology

Scrum, as the other agile methodologies, introduced a lightweight strategy overcoming some of the problems of rigid methodologies when adapting to change of requirements or new requirements. Scrum also introduced some key elements of software engineering practices to adapt to change and to speed up the devel-

opment without putting so much effort in the bureaucracy.
The main criticism brought by Scrum is the fact that the self-management of the Team, may lead to chaos development. Some authors state that Scrum does not consider the Human factor, and that people inside teams tend to gain the *control* of the team, specially when different levels of seniority are present in the team.
Also specially when addressing inexperienced team members can lead to ad-hoc management and/or *cowboy coding* being somewhat difficult to by stand with the philosophy of the methodology.

3.5.3 eXtreme Programming

The eXtreme Programming, normally designated XP, was created by Kent Beck, during a project for the auto-mobile industry, to develop a payroll system. XP is based in five values, the fifth was added in the 2nd edition of the publication of Extreme Programming explained by Kent Beck [5]:

- **Comunication**: It's a crucial factor when developing software, some authors considered inefficient communication to be the core of most of the problems in software projects. In traditional development methodologies, communication is formalized into documents, XP considers communication between the team, without the need of elaborated document, should be simple, mainly verbal communication, with the use of metaphors and simple designs that speed up the process and respond efficiently to changes.

- **Simplicity**: Designs should be as simple as possible and they should not try to prevent all problems that will surge, because they cannot be all predicted. So, the up front design should be a simple solution with no great detail, so it can be easily adaptable and extra functionality (the cherry in the top of the cake) can be designed and developed later, after everything that is essential is done. The buzz word for this value is *"great is enemy of good"*. Simplicity is also strongly connected with communication, a simpler design is easier to understand than a very complex one, and if the code developed is simpler all programmers will understand it quickly.

- **Feedback**: Frequent and on time feedback provides a proper reaction to problems and issues. Client also needs to receive feedback to avoid tension and rupture. This value is also related to the previous, being feedback a part of communication it should also be simple, one way of a very simple feedback from system is performing unit and integration testing of the increments added to the solution. Clients can also, periodically provide feedback when testing the functional requirements (called user stories) of the system.

- **Courage**: Acts should be taken with courage. Team members should have the courage to admit failure, and to provide the proper feedback without fearing the consequences, no programmer likes to throw away code that as cost him or her a lot of effort and time, but they should have the courage to throw it away when needed and to admit they failed. Courage is needed to respect the values above, and to stay strict to them.

- **Respect**: Team members should respect each others and their work should rely on that. When submitting a piece a code, a member of the team should take into account that it can affect the work of the others, so it should assure that the respect for others members work is guaranteed. They accomplish that by performing unit and integration testing between other tasks. Respect for the values is also required for the team members, and if the values are respected is a good start for respecting the other team members.

Roles

- **Programmer**: The programmer is the central figure of the methodology, as its name indicates. Its role is to translate the user story of the client into software. The programmer is also responsible for conducting unit testing (once XP uses test driven development as it will be explained further).He or she is also responsible for reviewing other programmers work (as it uses pair-programming, also explained further) and to re-factor code. The programmer has to respect all the values stated by XP.

- **Customer**: The customer is part of the team and is responsible for creating user stories, that will define the requirements for the programmers to implement, and to perform the acceptance test of the functionalities implemented. The customer knows what to be implemented and the programmer how, so it's the customer job to define which tests should be used to test the acceptance of the release.

- **Tester**: The tester helps the customer in writing the tests and to execute tests already implemented to assure that re-factoring and new functionalities don't compromise the previous implementations.

- **Tracker**: The tracker is in charge of retrieving informations about the evolution of the project and to deliver an opinion on effort estimations to implement each on of the functionalities, even that the effort estimation should be performed by the programmer, the tracker adjust them and try to keep them as exact as possible. The tracker is also responsible for keeping a log of all test results, errors and problems as well to inform the responsible to perform the corrections and tests with all the information required to do it. The tracker as to be capable of perform these activities as exact as possible and without being affected by the eventual the pressure that programmers feel towards changes and corrections, because it can lead to omissions of facts and possible problems.

- **Coach**: The coach is the person in charge of the project and to guide the rest of the team during the process. The coach, as to know the details of the process and to be capable of which practices can mitigate the problems that appear in the project.
- **Consultant**: This is an optional element of the team, and is a person that can bring specialized skills that the team does not have. The consultant has to guide the team members in solving the problems of the specialization that brought him to the project.
- **Big Boss**: The big boss is the project manager responsible to allocate and to deliver all the resources needed and it's responsible to intervene, if needed, to assure the success of the project.

Practices and Life Cycle

XP was named after the extreme approach when considering agile and lean development, this demonstrates how XP is strict with the use of some practices. In this section the life cycle of the methodology and the use of theses practices during each phase of the process are described.
The XP methodology is divided in 6 phases:

1. Exploration

The first phase of XP is characterized by the developers team getting used to the technologies and tools that they will use to develop the product and the customer developing an essential artifact to the implementation of XP, the **User Story**. User Stories defines, in a very succinct way, a functionality or functional requirement, to be implemented in the final product. The user story is built using **Story Cards**, which is a paper where the user describes the functionality in just a few words, is advised to think of a story card as a reminder to the developer to have a conversation with the customer. The developers then pick up the user stories and built up a very rudimentary system, exploring the technologies that will be used in the development of the system. During this phase programmers should develop in different ways presenting alternatives in concern of the architecture. In this stage, as in the rest of the process, programmers should work using **Pair-Programming**. In pair-programming, two programmers work in the same workstation analyzing the same task, one of them regarding the coding detail and the other with the functionality and reviewing the code. Programmers often trade roles, and it's advised that the pairs switch often, so everyone is familiar with what others are doing.

2. Planning

In this phase, characterized by the practice called **The Planning Game**, customers work with the programmers in order to define priorities and estimate effort for each user story and define which user stories are going to be implemented in the the next iteration. This phase can be split in two phases, being the first with the interaction of the user, to define the priorities and estimate the effort, called the **Release Planning** and a second phase, where only programmers are included defining the activities and tasks for each pair of programmers, who will commit them selves into a deadline for the task to be complete. During this phase also unit tests are developed, prior to the actual development, this approach is called the **Test Driven Development** is one of the practices advocated by XP

3. Iterations

The iteration phase can take from one to four weeks long and considers all the tasks normally stated in software development: analysis, design and testing. It also adds a planning for testing when the developers gather to discuss how the tests should be performed. The development how was explained above is developed using pair-programming.
XP recommends the practices of code sharing, where all the team is able to view and modify the code the called **Collective Code Ownership**. The use of **Coding Standards** is advised in order to simplify the mixture of the programmers and for everyone to understand the code as easy as possible. Developers, according to the values of XP, should develop simple designs and should use common metaphors so everyone can easily understand when someone is referring to some specific element of the requirements.
During the process of development besides the **Continuous Review** from part of one member of the pair, they should always work with the latest version of software, advocating the called **Continuous Integration**, and should perform **Re-factoring** of the code produced. This means removing ambiguity and redundancy from the code. As was referred in the previous phase, XP uses test

driven development, which means that tests should be continuous and even precede the development of code.
The customer is considered as part of the team as it should be involved during the development of the code, and should perform functional tests to the releases. This relates to two more practices that XP refers, **Customer On-Site** and **Small Releases**. Customer on-site, means that the client is not considered as to be external and should be in the same physical space as the developers, and the small releases serve to the customer to test each increment as soon as possible, these releases should run independently and they are not intended to go live.
When regarding the programmer welfare, the XP is very strict, and defines that no programmer should work more than **40 hours per week**, as tired programmers are more likely to produce errors. In the need of extra hours of work in one week from a developer, this developer should not work overtime in the next week.

4. Productionizing

In this phase the performance is analysed and together with the released produced in the previous phase the customer revises the release and can suggest modifications, the customer can demand new acceptance tests and performance to prove the validation of the release. Modifications can be pointed out, and they are then evaluated to go to the maintenance phase or being included in a next iteration if the cost and objectives suggest it.

5. Maintenance

In this phase the modifications suggested by the customer are implemented and an extra caution to integration is required. All uses cases should be re-run after the modifications are done and an update of the previous release is presented to the customer and it's considered to be in production.

6. Death

The dead of the project happens when there are no more user stories to implement and performance and stability of the system are approved by the customer. In this phase all documentation required by the customer is then produced, considering a simple description of the of the architecture and functionalities implemented.

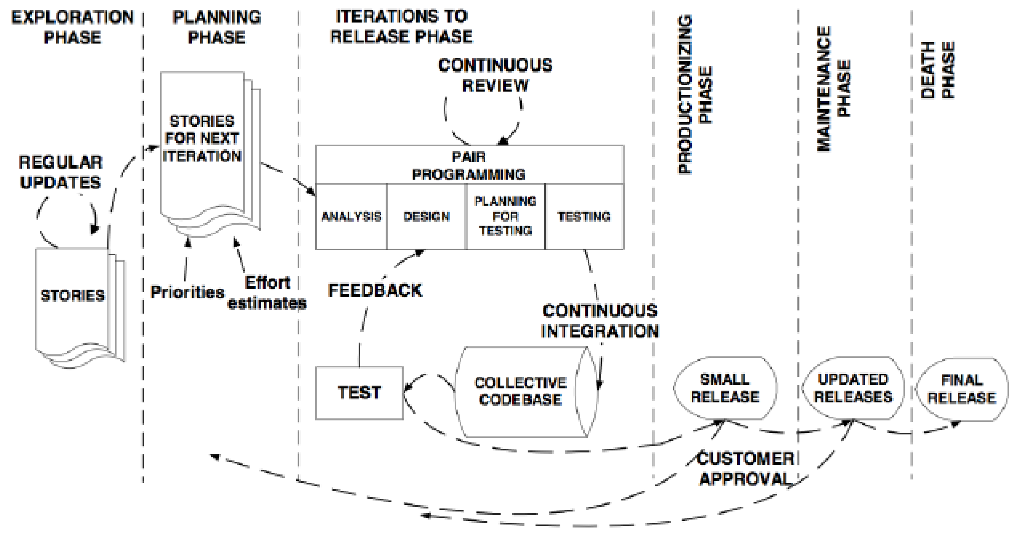


Figure 3.6: The Extreme Programming Life Cycle [1]

Critic to the XP Methodology

The extreme Programming methodology brought the simplicity commitment to software development, and a very big effort on tests. Being a agile methodology it was also a big cut with the previous monolithic approach.

One of the main critics to eXtreme programming comes from the design process being too simple, which can cause additional problems further in the end of the process, the rudimentary design address in the begin of the phase is considered by some author to be insufficient.

About pair programming, the critic made is concerned with lack of concentration coming from the work constantly done by two people, without having time to think in silence, when programming is a work, specially when addressing very complex algorithms or problems, that requires concentration and focus on the problem. Other privacy issues may cause also discomfort within the place of work.

The same concern is addressed when dealing with inexperienced programmers can cause the project to decline to chaos, like the criticism performed to Scrum, and almost all the agile development methodologies.

Chapter 4

Results

4.1 Method

To analyze SDMs using an empirical approach will imply an influence on the outcome, of using the SDM, and even in the adherence of the team to SDM's definition. This influence can appear by the characteristics of the team, its experience and to specific issues of the project given the context in which it is enrolled. This influence reflects subjectivity, and when performing an evaluation/classification subjectivity can imply that the results are not accurate. This kind of approach has been used by attempting to analyze and quantify the qualities of the product delivered when using the evaluated SDM, some examples can be found in the bibliography in [4] and [22].
Other approach, more informal, is to analyze an SDM and then try to determine how is better or worst by its advantages and criticisms.The problem with this approach is that the subjectivity is on the person who evaluates and by the sources that were used. Some examples of this kind of approaches can be found in the bibliography in [40],[38], [39] and [1]).
The approach taken in this dissertation was not to perform an empirical research, therefore removing the subjectivity of the specific experiences of using the SDM, and to perform an analysis of the SDM but removing the subjectivity by confronting them against facts instead of identifying advantages or criticisms informally.
The first step was to define an ideal use of the methodology, that is present in the state of the art sections 3.2, 3.4.2, and 3.5. Then the ideal methodology is confronted against the SWEBOK's knowledge areas by grading its adherence to these KAs. With this approach both subjectivity flaws stated previously are being avoided. Of course that the grading is performed by someone who analyzed the SDMs and also researched about the advantages and criticisms of

the SDM and it can still be biased by this research. Although, it will have to confront the SDM directly against a fact, prove that is adherent or not to this fact, and grade it with a satisfaction scale, so in the end subjectivity is lessened. The framework here presented it also allows to inflect subjectivity into the final results. It may sound confusing, because subjectivity is not in benefit of the classification but it can be useful when someone wishes to add their experience and their context to the choice that they are doing. These addition can be made by weighting the knowledge areas (or more detailed into the sub-knowledge areas). Again, because this addition is made after the classification is not adding subjectivity to the process of grading the SDM but to the process of choosing one.

An important issue when defining, the previous mentioned, ideal methodologies, was the extrapolation made from the models, specifically when addressing to waterfall and spiral model (sections 3.2.1 and 3.3.2). This extrapolation was risky but the intention was to address two very common approaches in the actual industry where methodologies derived from these models are still very used. This extrapolation used descriptions of the use of the models to projects defining an implicit methodology by both idealistic and averages uses of the methodology.

The analysis that is presented in this chapter is the, already mentioned, confrontation of the definition of the methodologies against the knowledge areas. To provide quantifiable metrics, the use of satisfaction points was given with the justification. A quantification is made, to provide a more easy way of addressing the results, by defining 0 to no satisfaction, 1 to partial satisfaction and 2 to full satisfaction.

A visual aid is also delivered for each methodology, by gathering the results in a bar chart with the percentage of satisfaction obtained in each knowledge area, this is calculated by dividing the actual satisfaction by the maximum possible of satisfaction.

This analysis should not be understood as an attempt of choosing which SDM is better, but to provide an overview of each methodology according to the software engineering methodologies, and to provide an aid when choosing a methodology, according to the required KA concerns.

4.2 Waterfall

Waterfall is still one of the most used methodologies (or methodologies based on this model) in the world, so it was a necessary presence in this analysis. The software engineering knowledge areas have evolved to fit the new context

engineering areas, some of the satisfaction points were low because of its monolithic approach, without space for change and and effective process evaluation or adaptation. The design process is very sustained in documents rather than in practical solutions that can provide a more effective design when defining the different kind of design levels. Quality assurance and requirements analysis have very high satisfaction points and software construction was penalized for not committing to simplicity and coding for verification.
The tables and bar chart for the waterfall methodology can be seen below.

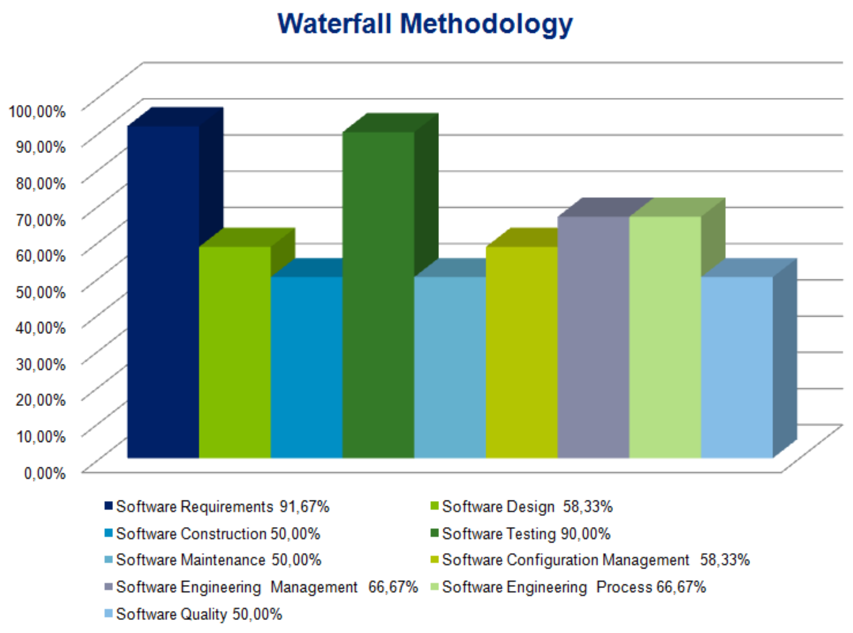
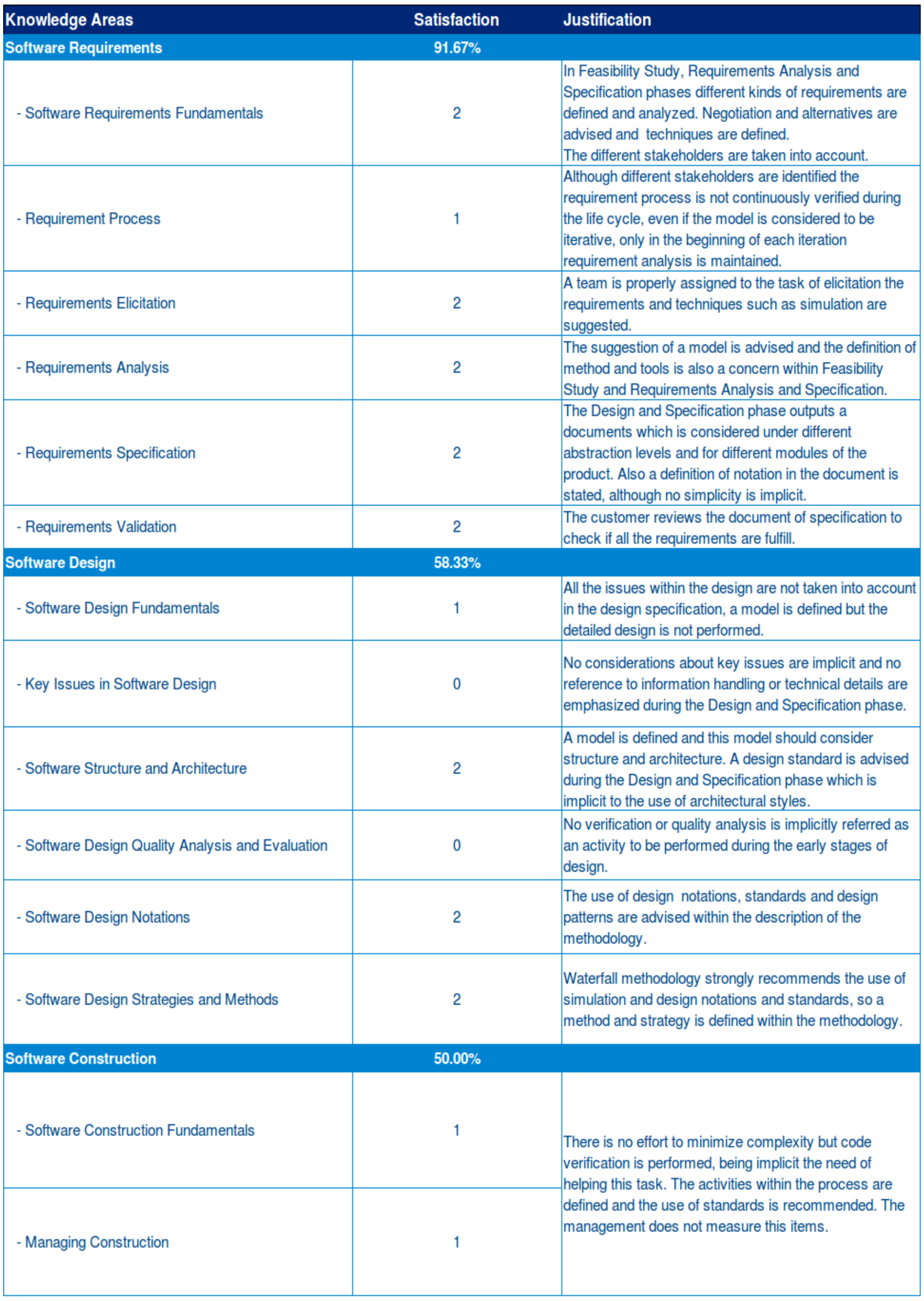


Figure 4.1: Waterfall: Software KA satisfaction bar chart

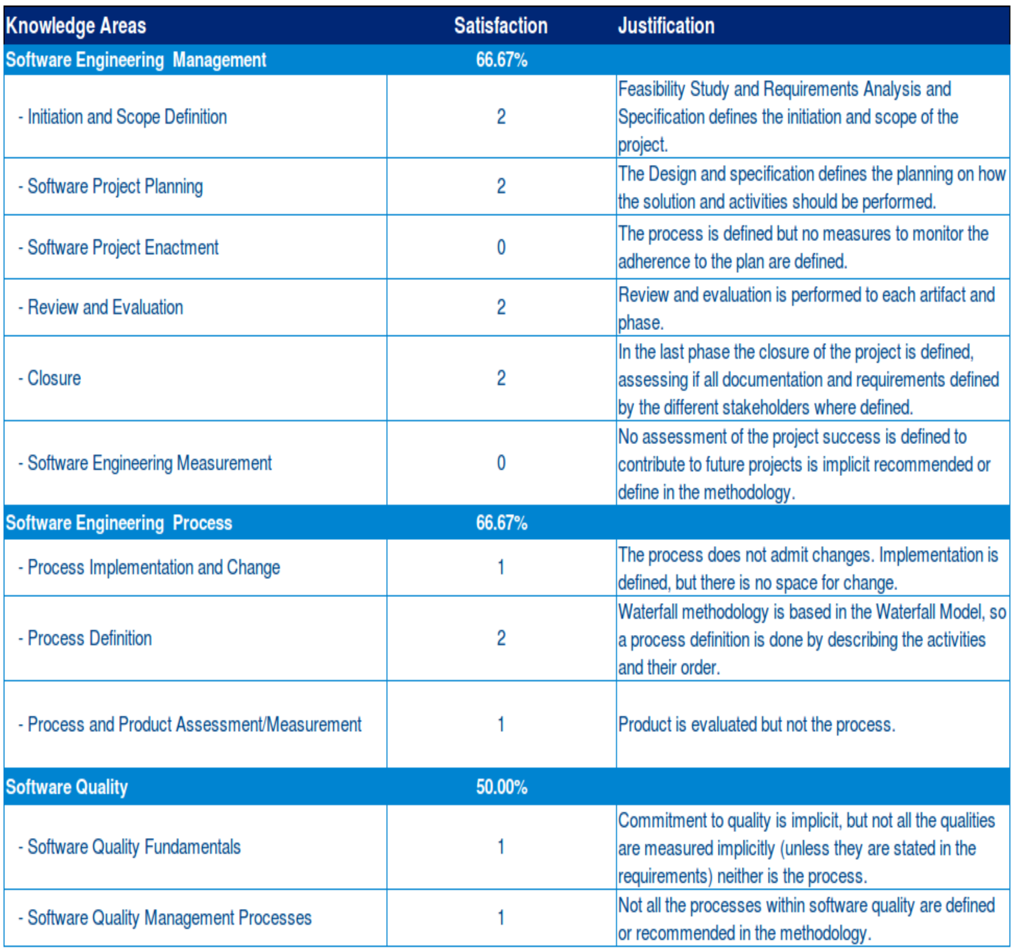


Knowledge Areas	Satisfaction	Justification
Software Requirements	**91.67%**	
- Software Requirements Fundamentals	2	In Feasibility Study, Requirements Analysis and Specification phases different kinds of requirements are defined and analyzed. Negotiation and alternatives are advised and techniques are defined. The different stakeholders are taken into account.
- Requirement Process	1	Although different stakeholders are identified the requirement process is not continuously verified during the life cycle, even if the model is considered to be iterative, only in the beginning of each iteration requirement analysis is maintained.
- Requirements Elicitation	2	A team is properly assigned to the task of elicitation the requirements and techniques such as simulation are suggested.
- Requirements Analysis	2	The suggestion of a model is advised and the definition of method and tools is also a concern within Feasibility Study and Requirements Analysis and Specification.
- Requirements Specification	2	The Design and Specification phase outputs a documents which is considered under different abstraction levels and for different modules of the product. Also a definition of notation in the document is stated, although no simplicity is implicit.
- Requirements Validation	2	The customer reviews the document of specification to check if all the requirements are fulfill.
Software Design	**58.33%**	
- Software Design Fundamentals	1	All the issues within the design are not taken into account in the design specification, a model is defined but the detailed design is not performed.
- Key Issues in Software Design	0	No considerations about key issues are implicit and no reference to information handling or technical details are emphasized during the Design and Specification phase.
- Software Structure and Architecture	2	A model is defined and this model should consider structure and architecture. A design standard is advised during the Design and Specification phase which is implicit to the use of architectural styles.
- Software Design Quality Analysis and Evaluation	0	No verification or quality analysis is implicitly referred as an activity to be performed during the early stages of design.
- Software Design Notations	2	The use of design notations, standards and design patterns are advised within the description of the methodology.
- Software Design Strategies and Methods	2	Waterfall methodology strongly recommends the use of simulation and design notations and standards, so a method and strategy is defined within the methodology.
Software Construction	**50.00%**	
- Software Construction Fundamentals	1	There is no effort to minimize complexity but code verification is performed, being implicit the need of helping this task. The activities within the process are defined and the use of standards is recommended. The management does not measure this items.
- Managing Construction	1	

Table 4.1: Waterfall: Software Requirements, Software Design and Software Construction KA

Knowledge Areas	Satisfaction	Justification
Software Testing	**90.00%**	
- Software Testing Fundamentals	2	Test is carried out during the development and the issues are addressed with the interaction of users.
- Test Levels	2	Unit testing and integration testing is defined in the methodology and its use is recommended.
- Test Techniques	2	Testing is advised to be performed within the levels previously defined and the use of standards is advised. The definition of different types of each level of testing is also defined, which creates a technique definition.
- Test Related Measures	1	The definition of correction is stated, but no testing measures are properly defined.
- Test Process	2	The methodology fulfill the different activities addressed in this knowledge area.
Software Maintenance	**50.00%**	
- Software Maintenance Fundamentals	2	Software Maintenance is defined within it's scope in delivery and maintenance phase.
- Key Issues in Software Maintenance	0	None of the issues is addressed in the methodology.
- Maintenance Process	2	Considers all the activities within the process of maintenance.
- Techniques for Maintenance	0	Does not describe any technique to perform maintenance.
Software Configuration Management	**58.33%**	
- Management of the SCM Process	1	During Feasibility Study and Design and Specification, concerns about the configuration management are taken but not during the product's life.
- Software Configuration Identification	2	The elements within the configuration management are identified in the Feasibility Study and in the Design and Specification phases.
- Software Configuration Control	0	No control is performed according to the description of the methodology, within the scope of Software Configuration Management.
- Software Configuration Status Accounting	1	During Feasibility Study the configuration should be defined in documentation but not during the rest of the life cycle .
- Software Configuration Auditing	1	Only some quality assurance is performed, but not within the complete scope of configuration management.
- Software Release Management and Delivery	2	The proper documentation is delivered and the configuration of the environment should be performed according to what was defined in the feasibility study.

Table 4.2: Waterfall: Software Testing, Software Maintenance and Software Configuration Management KA



Knowledge Areas	Satisfaction	Justification
Software Engineering Management	66.67%	
- Initiation and Scope Definition	2	Feasibility Study and Requirements Analysis and Specification defines the initiation and scope of the project.
- Software Project Planning	2	The Design and specification defines the planning on how the solution and activities should be performed.
- Software Project Enactment	0	The process is defined but no measures to monitor the adherence to the plan are defined.
- Review and Evaluation	2	Review and evaluation is performed to each artifact and phase.
- Closure	2	In the last phase the closure of the project is defined, assessing if all documentation and requirements defined by the different stakeholders where defined.
- Software Engineering Measurement	0	No assessment of the project success is defined to contribute to future projects is implicit recommended or define in the methodology.
Software Engineering Process	66.67%	
- Process Implementation and Change	1	The process does not admit changes. Implementation is defined, but there is no space for change.
- Process Definition	2	Waterfall methodology is based in the Waterfall Model, so a process definition is done by describing the activities and their order.
- Process and Product Assessment/Measurement	1	Product is evaluated but not the process.
Software Quality	50.00%	
- Software Quality Fundamentals	1	Commitment to quality is implicit, but not all the qualities are measured implicitly (unless they are stated in the requirements) neither is the process.
- Software Quality Management Processes	1	Not all the processes within software quality are defined or recommended in the methodology.

Table 4.3: Waterfall: Software Engineering Management, Software Engineering Process and Software Quality KA

4.3 Spiral

The spiral methodology, due to the prototyping technique, and the introduction of iterations and the revision in each iteration, provide a great satisfaction in the design and testing KA and a very reasonable one in software configuration management and software engineering process and management.
The lack of metrics and quantifications, besides the risk assessments, and the not commitment for simplicity and code verification are responsible for lower satisfaction points in software construction and software engineering management. Also a non definition of a specific phase or activities concerning maintenance reflected in the lower satisfaction for this methodology.
Next the tables and satisfaction bar chart for the spiral methodology are presented.

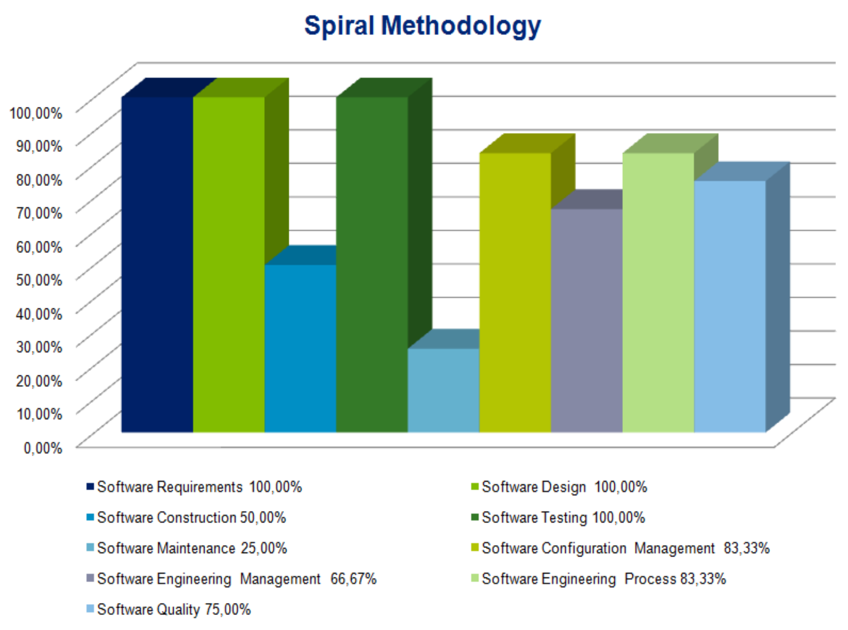
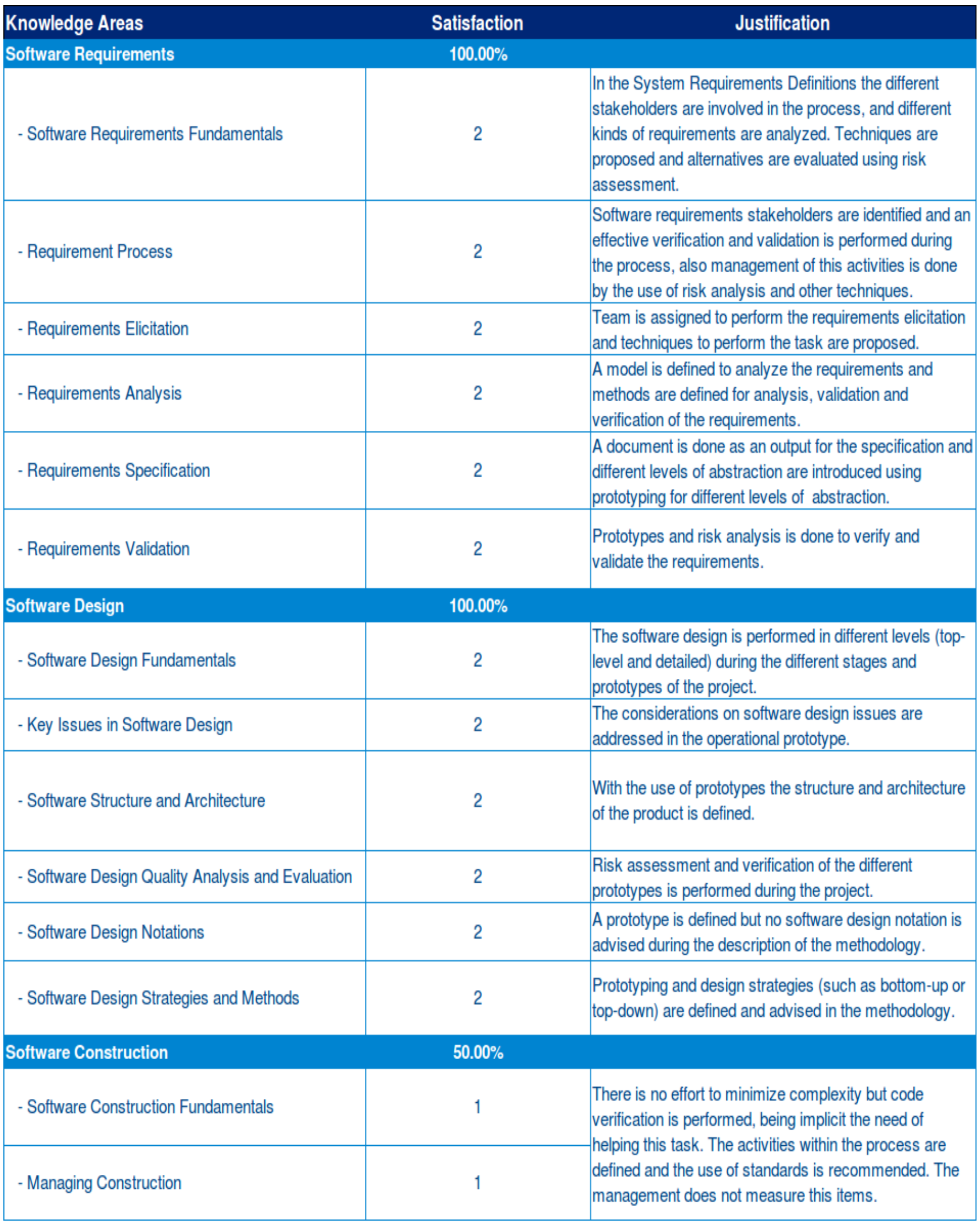


Figure 4.2: Spiral: Software KA satisfaction bar chart

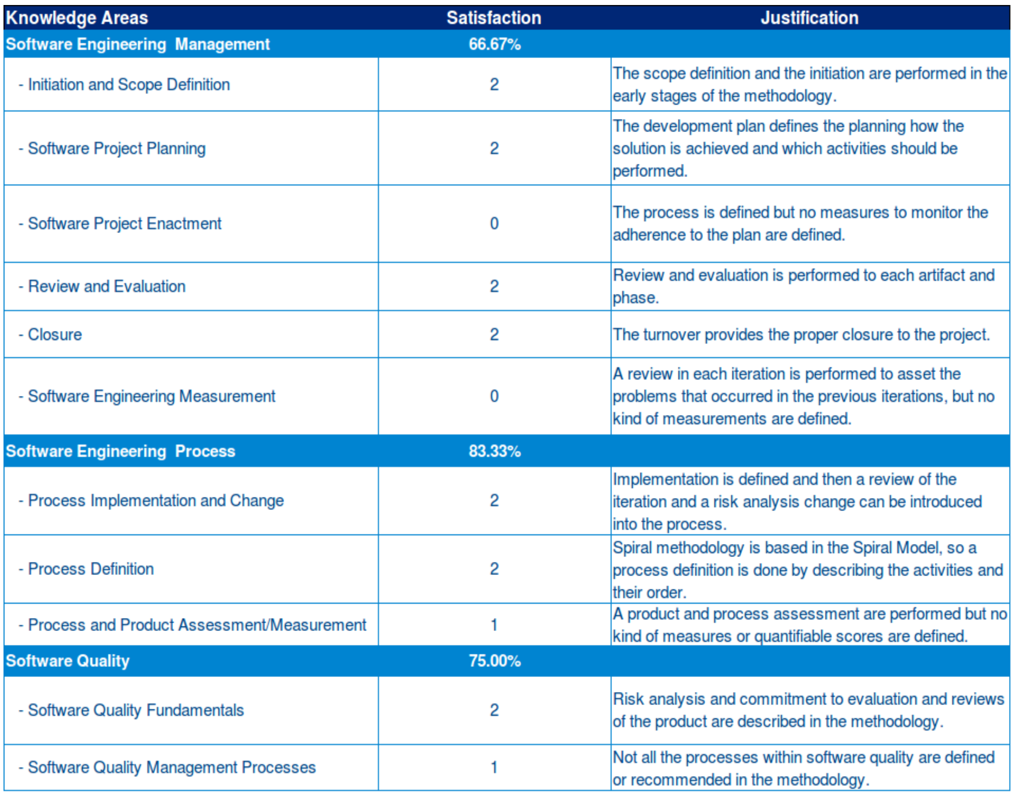


Knowledge Areas	Satisfaction	Justification
Software Requirements	**100.00%**	
- Software Requirements Fundamentals	2	In the System Requirements Definitions the different stakeholders are involved in the process, and different kinds of requirements are analyzed. Techniques are proposed and alternatives are evaluated using risk assessment.
- Requirement Process	2	Software requirements stakeholders are identified and an effective verification and validation is performed during the process, also management of this activities is done by the use of risk analysis and other techniques.
- Requirements Elicitation	2	Team is assigned to perform the requirements elicitation and techniques to perform the task are proposed.
- Requirements Analysis	2	A model is defined to analyze the requirements and methods are defined for analysis, validation and verification of the requirements.
- Requirements Specification	2	A document is done as an output for the specification and different levels of abstraction are introduced using prototyping for different levels of abstraction.
- Requirements Validation	2	Prototypes and risk analysis is done to verify and validate the requirements.
Software Design	**100.00%**	
- Software Design Fundamentals	2	The software design is performed in different levels (top-level and detailed) during the different stages and prototypes of the project.
- Key Issues in Software Design	2	The considerations on software design issues are addressed in the operational prototype.
- Software Structure and Architecture	2	With the use of prototypes the structure and architecture of the product is defined.
- Software Design Quality Analysis and Evaluation	2	Risk assessment and verification of the different prototypes is performed during the project.
- Software Design Notations	2	A prototype is defined but no software design notation is advised during the description of the methodology.
- Software Design Strategies and Methods	2	Prototyping and design strategies (such as bottom-up or top-down) are defined and advised in the methodology.
Software Construction	**50.00%**	
- Software Construction Fundamentals	1	There is no effort to minimize complexity but code verification is performed, being implicit the need of helping this task. The activities within the process are defined and the use of standards is recommended. The management does not measure this items.
- Managing Construction	1	

Table 4.4: Spiral: Software Requirements, Software Design and Software Construction KA

Knowledge Areas	Satisfaction	Justification
Software Testing	**100.00%**	
- Software Testing Fundamentals	2	Test is carried out during the development and the issues are addressed with the interaction of users.
- Test Levels	2	Within the methodology different levels of tests are performed.
- Test Techniques	2	The methodology defines a test plan in which a test technique is implicit.
- Test Related Measures	2	The use of measures in testing is defined in the methodology.
- Test Process	2	The methodology fulfill the different activities addressed in this knowledge area.
Software Maintenance	**25.00%**	
- Software Maintenance Fundamentals	0	Software maintenance is considered as new iteration of the life cycle so it does not fulfill all the kinds of maintenance either its activities.
- Key Issues in Software Maintenance	1	Only technical issues are considered in the methodology.
- Maintenance Process	1	Does not considers all the activities within the process of maintenance, but as maintenance requires some common activities with development implementation and problem analysis can be viewed as an phase of the development.
- Techniques for Maintenance	0	Does not describe any technique to perform maintenance.
Software Configuration Management	**83.33%**	
- Management of the SCM Process	2	With the use of prototypes with it's review and risk analysis, a control and management of the SCM process is done.
- Software Configuration Identification	2	During the early stages the identification of the configuration is performed.
- Software Configuration Control	2	With the risk analysis of the prototypes and the review of the development on each iteration of the spiral, the configuration is controlled.
- Software Configuration Status Accounting	1	The configuration is documented in the beginning of the project and it should evolve during it's development, but no measure of the information is produced.
- Software Configuration Auditing	1	The validation of the conformance with the requirements is developed, but as no measurement is done, it does not fulfill entirely the tasks described in this sub knowledge area.
- Software Release Management and Delivery	2	The proper documentation is delivered and the configuration of the environment should be performed according to what was defined in the requirements analysis.

Table 4.5: Spiral: Software Testing, Software Maintenance and Software Configuration Management KA



Knowledge Areas	Satisfaction	Justification
Software Engineering Management	**66.67%**	
- Initiation and Scope Definition	2	The scope definition and the initiation are performed in the early stages of the methodology.
- Software Project Planning	2	The development plan defines the planning how the solution is achieved and which activities should be performed.
- Software Project Enactment	0	The process is defined but no measures to monitor the adherence to the plan are defined.
- Review and Evaluation	2	Review and evaluation is performed to each artifact and phase.
- Closure	2	The turnover provides the proper closure to the project.
- Software Engineering Measurement	0	A review in each iteration is performed to asset the problems that occurred in the previous iterations, but no kind of measurements are defined.
Software Engineering Process	**83.33%**	
- Process Implementation and Change	2	Implementation is defined and then a review of the iteration and a risk analysis change can be introduced into the process.
- Process Definition	2	Spiral methodology is based in the Spiral Model, so a process definition is done by describing the activities and their order.
- Process and Product Assessment/Measurement	1	A product and process assessment are performed but no kind of measures or quantifiable scores are defined.
Software Quality	**75.00%**	
- Software Quality Fundamentals	2	Risk analysis and commitment to evaluation and reviews of the product are described in the methodology.
- Software Quality Management Processes	1	Not all the processes within software quality are defined or recommended in the methodology.

Table 4.6: Spiral: Software Engineering Management, Software Engineering Process and Software Quality KA

4.4 Rapid Application Development

The RAD methodology, with the use of tools and its management approach got a total satisfaction of the software construction KA and with the use of simulation a total satisfaction of the requirements KA.
An insufficient concern with the testing and maintenance phase were reflected in the lowest satisfaction for this methodology. The design lacked a notation and standard definition and the methodology lacked measurements and quantifiable reviews.

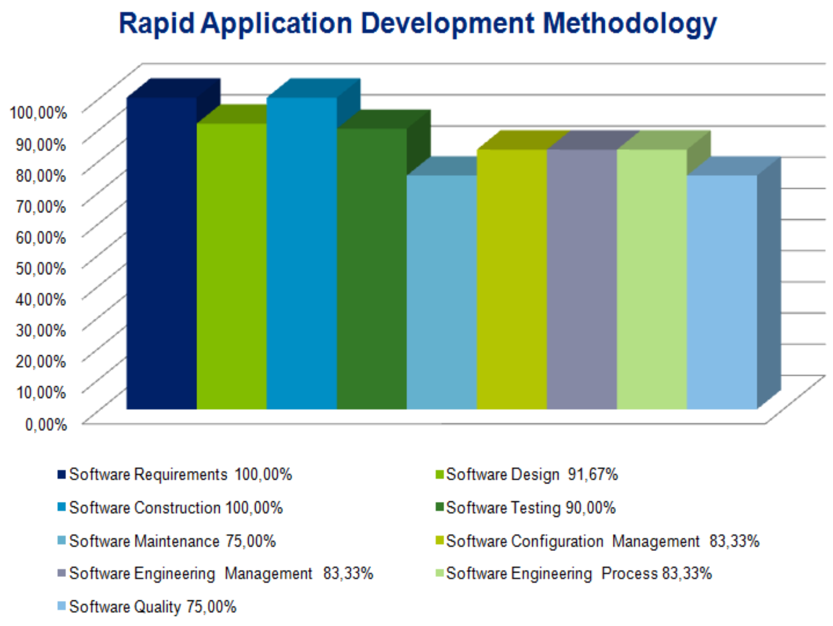
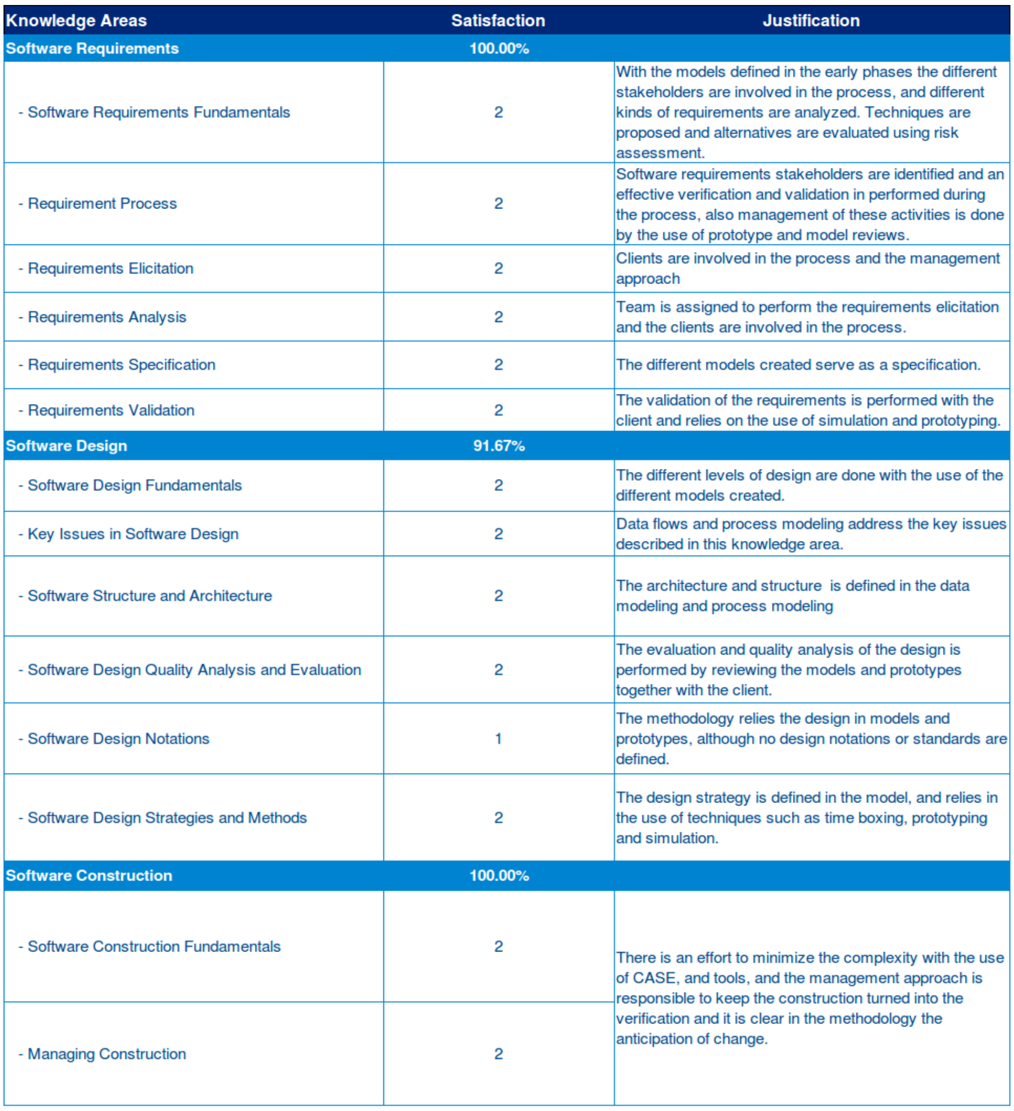


Figure 4.3: RAD: Software KA satisfaction bar chart

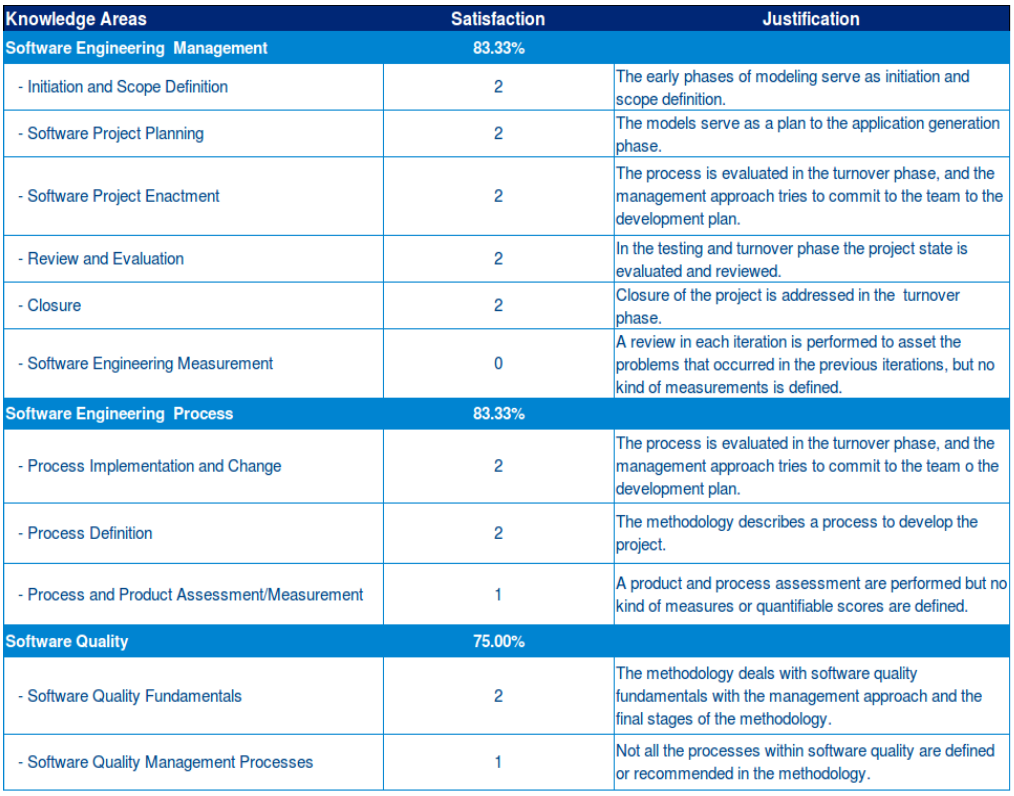


Knowledge Areas	Satisfaction	Justification
Software Requirements	**100.00%**	
- Software Requirements Fundamentals	2	With the models defined in the early phases the different stakeholders are involved in the process, and different kinds of requirements are analyzed. Techniques are proposed and alternatives are evaluated using risk assessment.
- Requirement Process	2	Software requirements stakeholders are identified and an effective verification and validation in performed during the process, also management of these activities is done by the use of prototype and model reviews.
- Requirements Elicitation	2	Clients are involved in the process and the management approach
- Requirements Analysis	2	Team is assigned to perform the requirements elicitation and the clients are involved in the process.
- Requirements Specification	2	The different models created serve as a specification.
- Requirements Validation	2	The validation of the requirements is performed with the client and relies on the use of simulation and prototyping.
Software Design	**91.67%**	
- Software Design Fundamentals	2	The different levels of design are done with the use of the different models created.
- Key Issues in Software Design	2	Data flows and process modeling address the key issues described in this knowledge area.
- Software Structure and Architecture	2	The architecture and structure is defined in the data modeling and process modeling
- Software Design Quality Analysis and Evaluation	2	The evaluation and quality analysis of the design is performed by reviewing the models and prototypes together with the client.
- Software Design Notations	1	The methodology relies the design in models and prototypes, although no design notations or standards are defined.
- Software Design Strategies and Methods	2	The design strategy is defined in the model, and relies in the use of techniques such as time boxing, prototyping and simulation.
Software Construction	**100.00%**	
- Software Construction Fundamentals	2	There is an effort to minimize the complexity with the use of CASE, and tools, and the management approach is responsible to keep the construction turned into the verification and it is clear in the methodology the anticipation of change.
- Managing Construction	2	

Table 4.7: RAD: Software Requirements, Software Design and Software Construction KA

Knowledge Areas	Satisfaction	Justification
Software Testing	**90.00%**	
- Software Testing Fundamentals	2	Test is carried out during the development and the issues are addressed with the interaction of users.
- Test Levels	2	The different levels of testing are performed.
- Test Techniques	2	The methodology emphasizes the use of tools and techniques to improve the testing tasks.
- Test Related Measures	1	There is no specific mention to testing related measures, but as this methodology relies in the use of tools to improve it's quality/cost, this can be addressed as many of the modern tools (such as IDE's) provides test related measures.
- Test Process	2	The methodology fulfill the different activities addressed in this knowledge area.
Software Maintenance	**75.00%**	
- Software Maintenance Fundamentals	1	The maintenance is viewed as changes and correction not describing all the tasks defined in this knowledge area.
- Key Issues in Software Maintenance	2	All issues are covered in the methodology, as the technical skills should be addressed with modeling and the management issues with the management approach defined in the methodology.
- Maintenance Process	2	Considers all the activities within the process of maintenance.
- Techniques for Maintenance	1	No techniques are defined in the methodology, but the use of tools can imply the use of maintenance techniques.
Software Configuration Management	**83.33%**	
- Management of the SCM Process	2	The SCM process is managed with the use of prototypes and the control is performed with the client and the management elements.
- Software Configuration Identification	2	The identification of the configuration is done with the use of models and prototypes.
- Software Configuration Control	2	The control is performed during the turnover phase and the review of models and prototypes.
- Software Configuration Status Accounting	1	The configuration is documented in the beginning of the project and it should evolve during it's development, but no measure of the information is produced.
- Software Configuration Auditing	1	The validation of the conformance with the requirements is developed, but as no measurement it does not fulfill entirely the tasks described in this sub knowledge area.
- Software Release Management and Delivery	2	During the turnover phase the proper documentation is provided and validated with the client.

Table 4.8: RAD: Software Testing, Software Maintenance and Software Configuration Management KA



Knowledge Areas	Satisfaction	Justification
Software Engineering Management	**83.33%**	
- Initiation and Scope Definition	2	The early phases of modeling serve as initiation and scope definition.
- Software Project Planning	2	The models serve as a plan to the application generation phase.
- Software Project Enactment	2	The process is evaluated in the turnover phase, and the management approach tries to commit to the team to the development plan.
- Review and Evaluation	2	In the testing and turnover phase the project state is evaluated and reviewed.
- Closure	2	Closure of the project is addressed in the turnover phase.
- Software Engineering Measurement	0	A review in each iteration is performed to asset the problems that occurred in the previous iterations, but no kind of measurements is defined.
Software Engineering Process	**83.33%**	
- Process Implementation and Change	2	The process is evaluated in the turnover phase, and the management approach tries to commit to the team o the development plan.
- Process Definition	2	The methodology describes a process to develop the project.
- Process and Product Assessment/Measurement	1	A product and process assessment are performed but no kind of measures or quantifiable scores are defined.
Software Quality	**75.00%**	
- Software Quality Fundamentals	2	The methodology deals with software quality fundamentals with the management approach and the final stages of the methodology.
- Software Quality Management Processes	1	Not all the processes within software quality are defined or recommended in the methodology.

Table 4.9: RAD: Software Engineering Management, Software Engineering Process and Software Quality KA

4.5 Scrum

The Scrum methodology due to its role distribution and with the use of artifacts such as backlogs and burn-down charts was the only methodology to fully satisfy the software engineering management and software engineering process KA (although eXtreme programming also fulfilled the software engineering process). Scrum approach on maintenance as another iteration of the process and the lack of effort on minimizing the complexity reflected in the lowest satisfaction for this methodology on software construction and maintenances KA. The tables and bar chart satisfaction for the Scrum methodology are stated below.

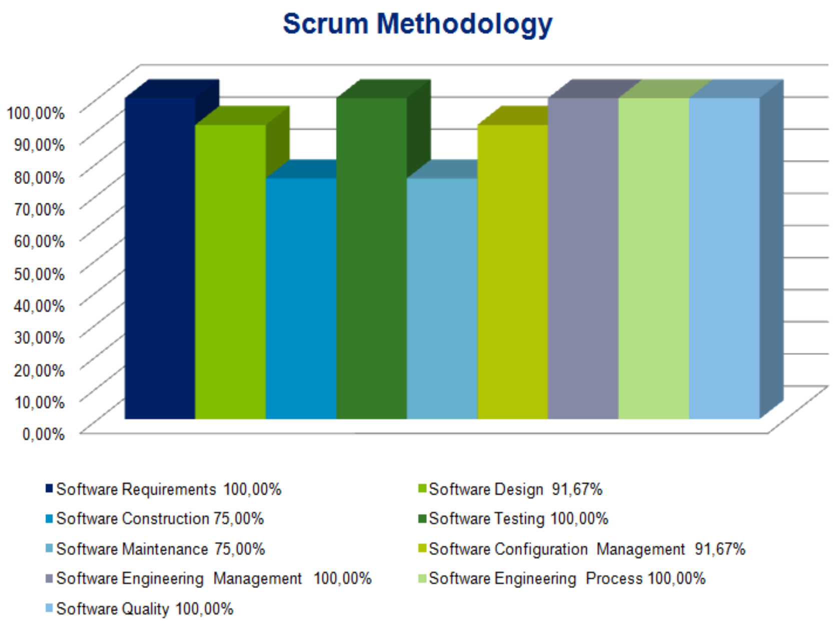
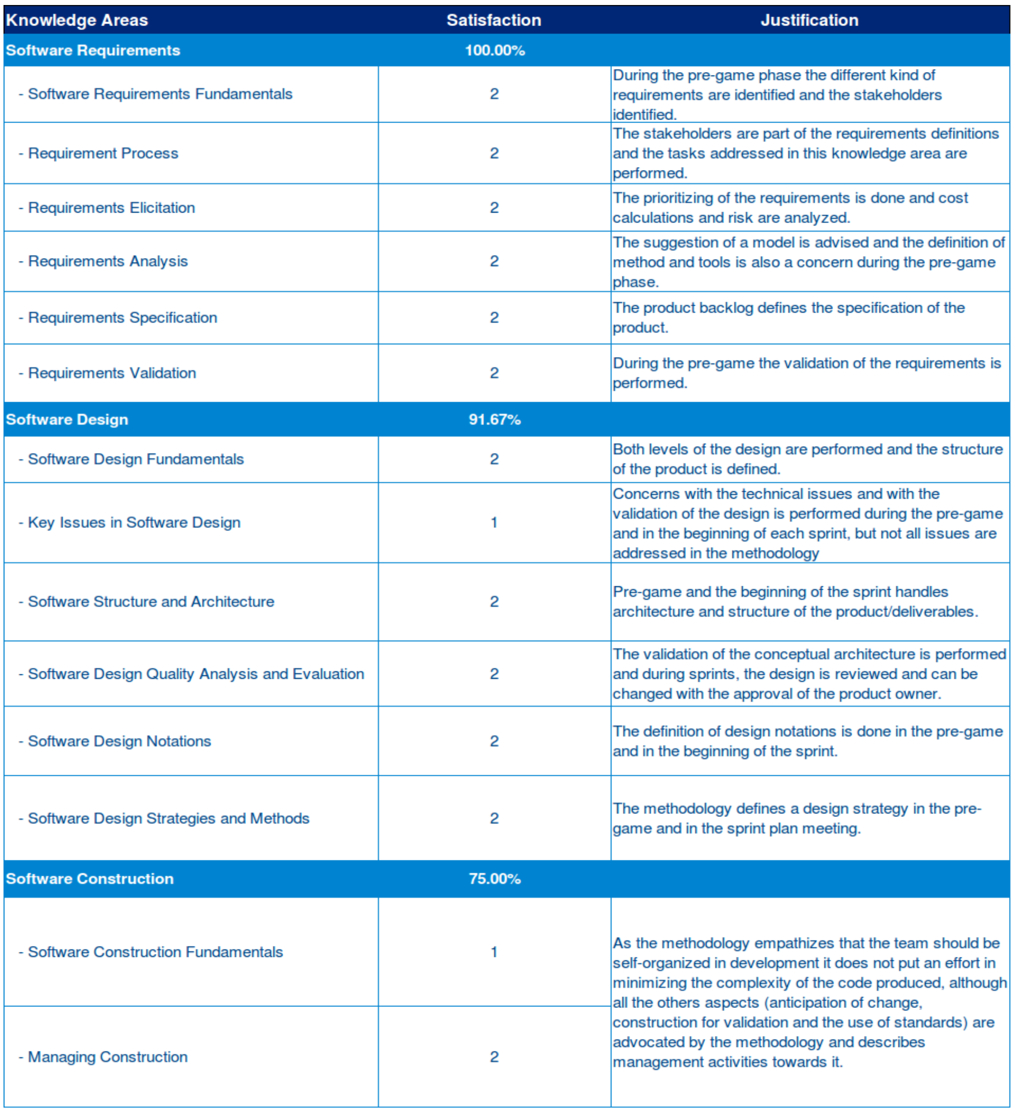


Figure 4.4: Scrum: Software KA satisfaction bar chart

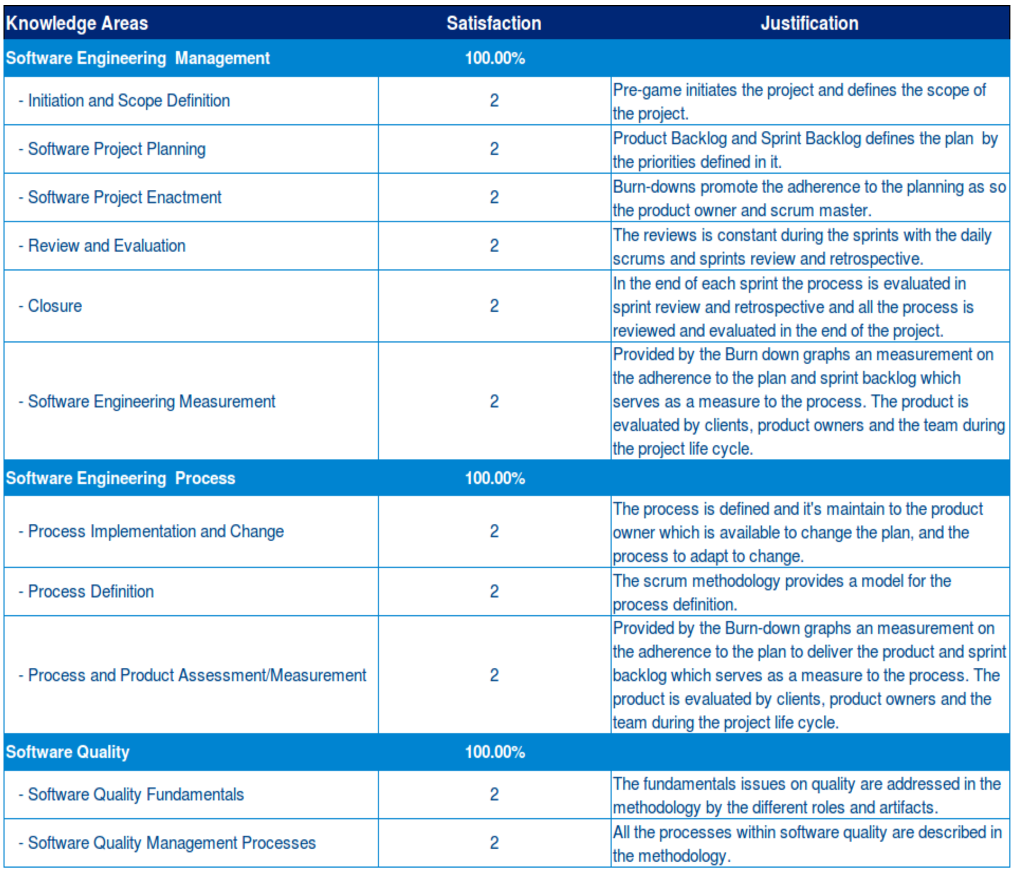


Knowledge Areas	Satisfaction	Justification
Software Requirements	**100.00%**	
- Software Requirements Fundamentals	2	During the pre-game phase the different kind of requirements are identified and the stakeholders identified.
- Requirement Process	2	The stakeholders are part of the requirements definitions and the tasks addressed in this knowledge area are performed.
- Requirements Elicitation	2	The prioritizing of the requirements is done and cost calculations and risk are analyzed.
- Requirements Analysis	2	The suggestion of a model is advised and the definition of method and tools is also a concern during the pre-game phase.
- Requirements Specification	2	The product backlog defines the specification of the product.
- Requirements Validation	2	During the pre-game the validation of the requirements is performed.
Software Design	**91.67%**	
- Software Design Fundamentals	2	Both levels of the design are performed and the structure of the product is defined.
- Key Issues in Software Design	1	Concerns with the technical issues and with the validation of the design is performed during the pre-game and in the beginning of each sprint, but not all issues are addressed in the methodology
- Software Structure and Architecture	2	Pre-game and the beginning of the sprint handles architecture and structure of the product/deliverables.
- Software Design Quality Analysis and Evaluation	2	The validation of the conceptual architecture is performed and during sprints, the design is reviewed and can be changed with the approval of the product owner.
- Software Design Notations	2	The definition of design notations is done in the pre-game and in the beginning of the sprint.
- Software Design Strategies and Methods	2	The methodology defines a design strategy in the pre-game and in the sprint plan meeting.
Software Construction	**75.00%**	
- Software Construction Fundamentals	1	As the methodology empathizes that the team should be self-organized in development it does not put an effort in minimizing the complexity of the code produced, although all the others aspects (anticipation of change, construction for validation and the use of standards) are advocated by the methodology and describes management activities towards it.
- Managing Construction	2	

Table 4.10: Scrum: Software Requirements, Software Design and Software Construction KA

Knowledge Areas	Satisfaction	Justification
Software Testing	**100.00%**	
- Software Testing Fundamentals	2	Test is carried out during the development and the issues are addressed with the supervision of the product owner.
- Test Levels	2	Different levels of testing are performed during the software life cycle. Unit tests during the sprint and system and integration tests at post-game.
- Test Techniques	2	Test techniques are defined in the sprint planning meeting
- Test Related Measures	2	Test related measures are advised and should be recorded into sprint backlog.
- Test Process	2	All activities of the test process are present in the description of the methodology.
Software Maintenance	**75.00%**	
- Software Maintenance Fundamentals	1	The maintenance is viewed as a change or correction not describing all the tasks defined in this knowledge area.
- Key Issues in Software Maintenance	2	With the different roles and the sprint tasks the main issues in maintenance are defined.
- Maintenance Process	2	Considers all the activities within the process of maintenance.
- Techniques for Maintenance	1	No specific techniques are defined in the methodology, but they can (should) be addressed in the pre-game and sprint planning.
Software Configuration Management	**91.67%**	
- Management of the SCM Process	2	The management of the SCM process is done by the inspection of product and sprint backlog and by the meeting held prior and at the end of the sprint.
- Software Configuration Identification	2	The configuration identification should be maintained in the product backlog.
- Software Configuration Control	2	The control is performed with the burn-down, backlogs and reviews.
- Software Configuration Status Accounting	1	The product owner should keep the configuration definition in the product backlog and if it alters the backlog it should reflect those changes, but the documentation is only proper delivered in the end of the project.
- Software Configuration Auditing	2	The auditing is a responsibility of scrum master and product owner their aids are the burn-downs, the reviews and retrospectives done at the configuration baseline.
- Software Release Management and Delivery	2	In the post-game phase all the documentation is produced and the integration is provided.

Table 4.11: Scrum: Software Testing, Software Maintenance and Software Configuration Management KA



Knowledge Areas	Satisfaction	Justification
Software Engineering Management	**100.00%**	
- Initiation and Scope Definition	2	Pre-game initiates the project and defines the scope of the project.
- Software Project Planning	2	Product Backlog and Sprint Backlog defines the plan by the priorities defined in it.
- Software Project Enactment	2	Burn-downs promote the adherence to the planning as so the product owner and scrum master.
- Review and Evaluation	2	The reviews is constant during the sprints with the daily scrums and sprints review and retrospective.
- Closure	2	In the end of each sprint the process is evaluated in sprint review and retrospective and all the process is reviewed and evaluated in the end of the project.
- Software Engineering Measurement	2	Provided by the Burn down graphs an measurement on the adherence to the plan and sprint backlog which serves as a measure to the process. The product is evaluated by clients, product owners and the team during the project life cycle.
Software Engineering Process	**100.00%**	
- Process Implementation and Change	2	The process is defined and it's maintain to the product owner which is available to change the plan, and the process to adapt to change.
- Process Definition	2	The scrum methodology provides a model for the process definition.
- Process and Product Assessment/Measurement	2	Provided by the Burn-down graphs an measurement on the adherence to the plan to deliver the product and sprint backlog which serves as a measure to the process. The product is evaluated by clients, product owners and the team during the project life cycle.
Software Quality	**100.00%**	
- Software Quality Fundamentals	2	The fundamentals issues on quality are addressed in the methodology by the different roles and artifacts.
- Software Quality Management Processes	2	All the processes within software quality are described in the methodology.

Table 4.12: Scrum: Software Engineering Management, Software Engineering Process and Software Quality KA

4.6 eXtreme Programming

A dedicated phase to maintenance, and a concern with activities that extend the development issues, as the philosophy stating the principles of simplicity, a very detailed review and roles reflecting those principle turned into eXtreme Programming being the only methodology to totally satisfy the software maintenance and software construction KA. Adding the roles of tester and tracker the proper Software Configuration Management was also only fully satisfied by this methodology.
It was also interesting noticing that one of the main critics to this methodology being the design process, it was also, together with software engineering management and process KA, the lowest satisfied KA for this methodology. Although getting a very high satisfaction on all of them. The fact of not considering the different levels of design and not providing the measures or assessments of the procedures and process was the reason for those satisfaction levels.
More detailed descriptions for each sub-KA can be found in the tables and bar charts for this methodology that are now presented.

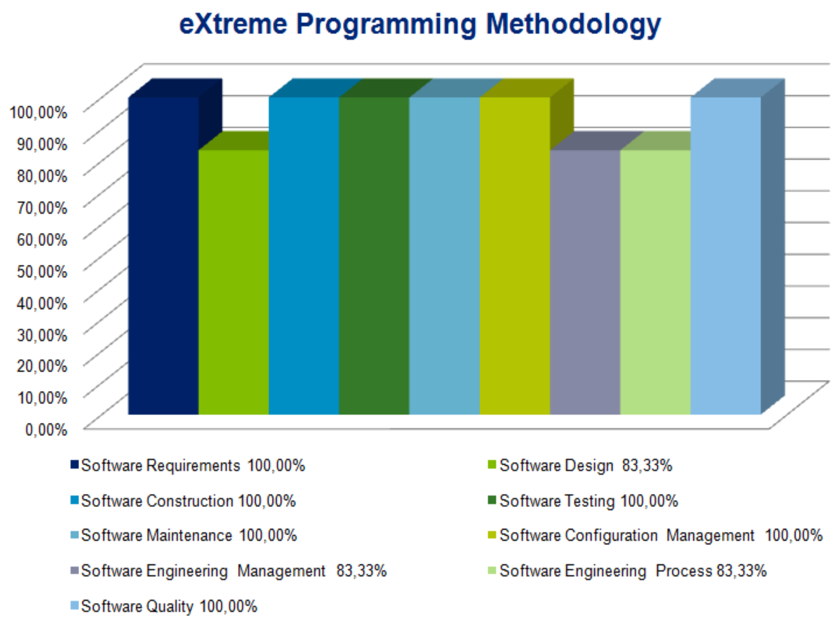
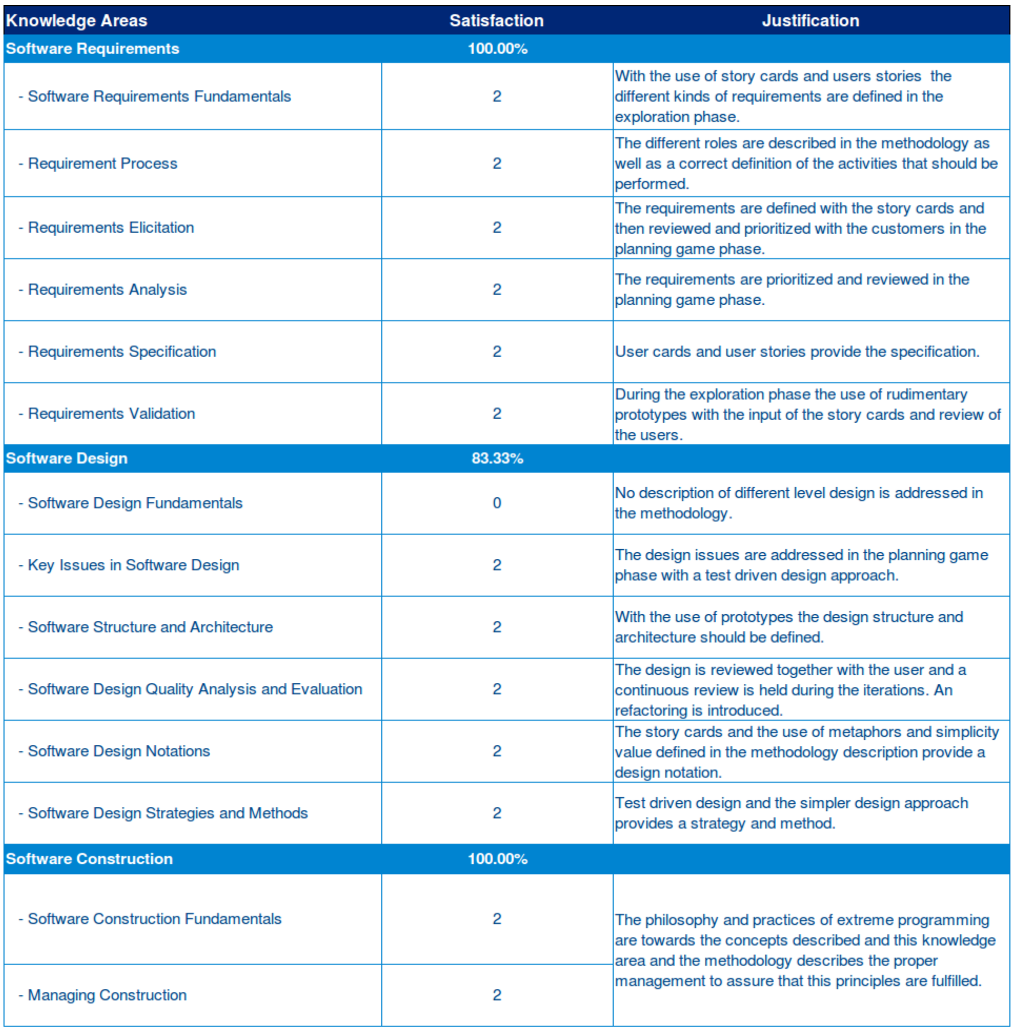
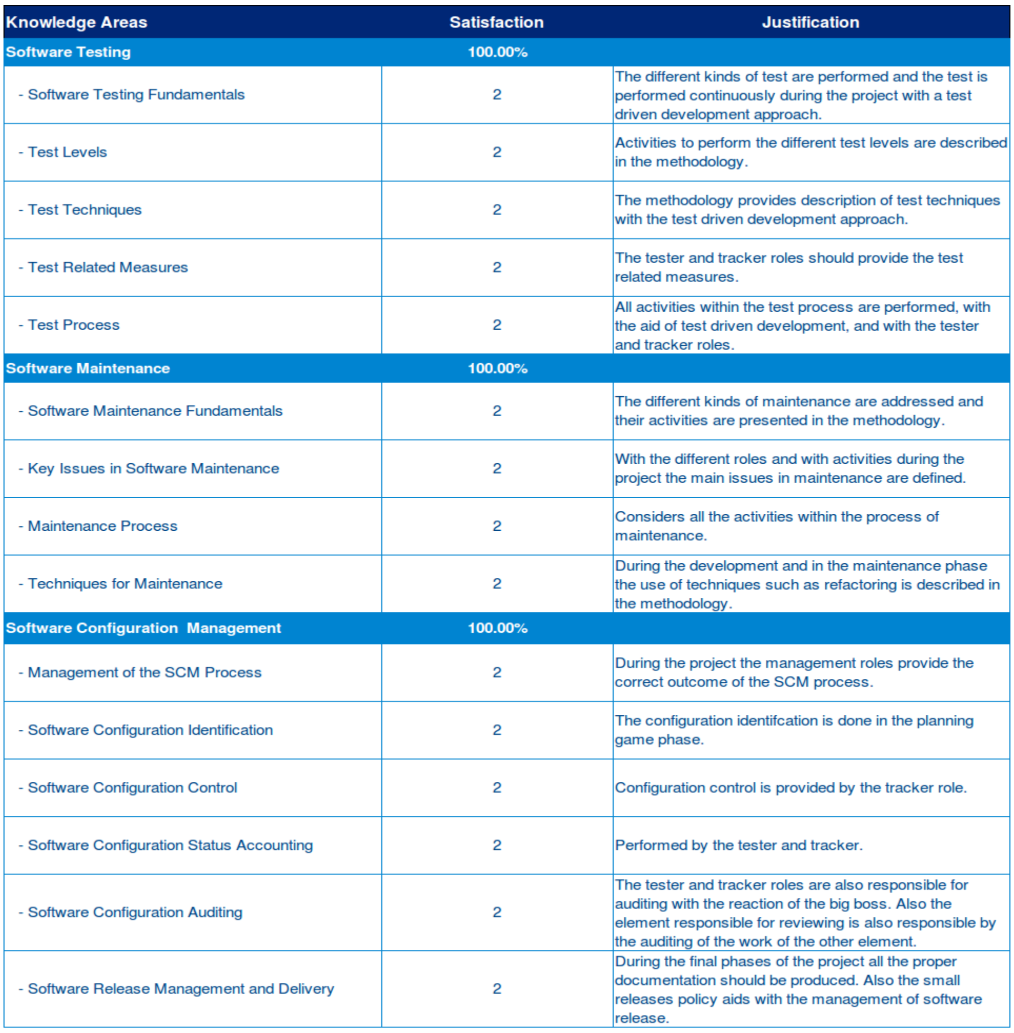


Figure 4.5: XP: Software KA satisfaction bar chart



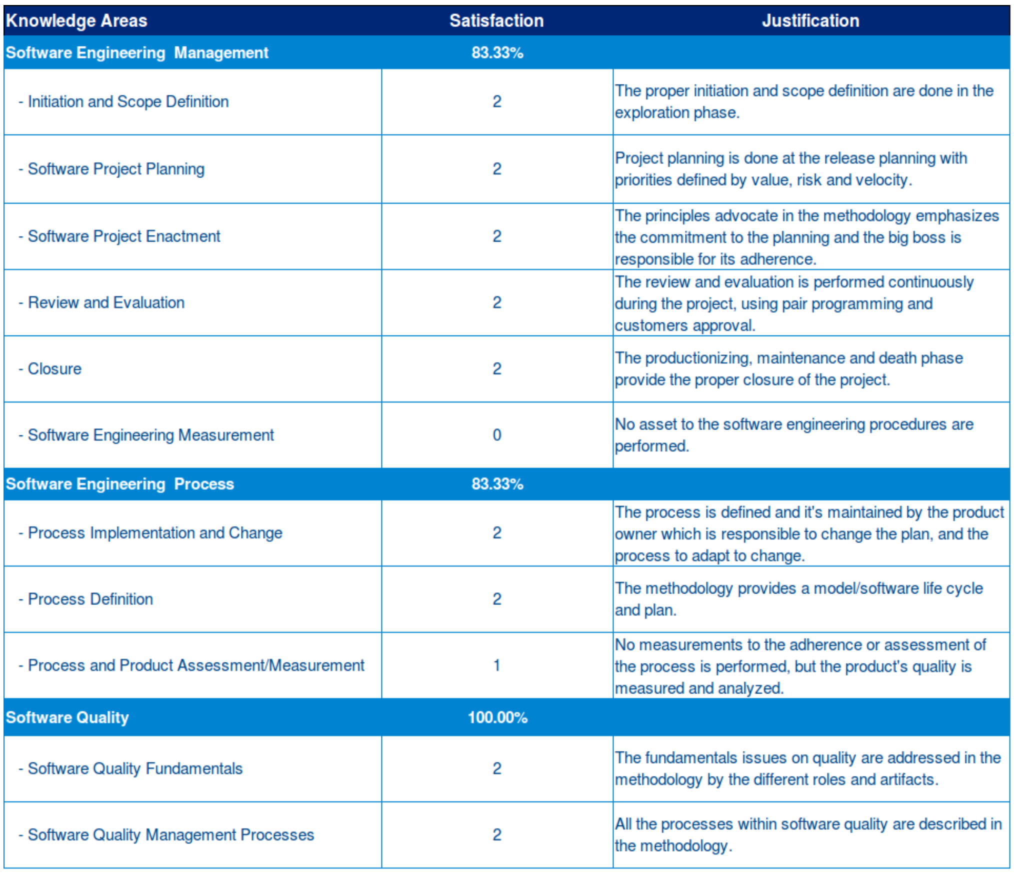
Knowledge Areas	Satisfaction	Justification
Software Requirements	**100.00%**	
- Software Requirements Fundamentals	2	With the use of story cards and users stories the different kinds of requirements are defined in the exploration phase.
- Requirement Process	2	The different roles are described in the methodology as well as a correct definition of the activities that should be performed.
- Requirements Elicitation	2	The requirements are defined with the story cards and then reviewed and prioritized with the customers in the planning game phase.
- Requirements Analysis	2	The requirements are prioritized and reviewed in the planning game phase.
- Requirements Specification	2	User cards and user stories provide the specification.
- Requirements Validation	2	During the exploration phase the use of rudimentary prototypes with the input of the story cards and review of the users.
Software Design	**83.33%**	
- Software Design Fundamentals	0	No description of different level design is addressed in the methodology.
- Key Issues in Software Design	2	The design issues are addressed in the planning game phase with a test driven design approach.
- Software Structure and Architecture	2	With the use of prototypes the design structure and architecture should be defined.
- Software Design Quality Analysis and Evaluation	2	The design is reviewed together with the user and a continuous review is held during the iterations. An refactoring is introduced.
- Software Design Notations	2	The story cards and the use of metaphors and simplicity value defined in the methodology description provide a design notation.
- Software Design Strategies and Methods	2	Test driven design and the simpler design approach provides a strategy and method.
Software Construction	**100.00%**	
- Software Construction Fundamentals	2	The philosophy and practices of extreme programming are towards the concepts described and this knowledge area and the methodology describes the proper management to assure that this principles are fulfilled.
- Managing Construction	2	

Table 4.13: XP: Software Requirements, Software Design and Software Construction KA



Knowledge Areas	Satisfaction	Justification
Software Testing	**100.00%**	
- Software Testing Fundamentals	2	The different kinds of test are performed and the test is performed continuously during the project with a test driven development approach.
- Test Levels	2	Activities to perform the different test levels are described in the methodology.
- Test Techniques	2	The methodology provides description of test techniques with the test driven development approach.
- Test Related Measures	2	The tester and tracker roles should provide the test related measures.
- Test Process	2	All activities within the test process are performed, with the aid of test driven development, and with the tester and tracker roles.
Software Maintenance	**100.00%**	
- Software Maintenance Fundamentals	2	The different kinds of maintenance are addressed and their activities are presented in the methodology.
- Key Issues in Software Maintenance	2	With the different roles and with activities during the project the main issues in maintenance are defined.
- Maintenance Process	2	Considers all the activities within the process of maintenance.
- Techniques for Maintenance	2	During the development and in the maintenance phase the use of techniques such as refactoring is described in the methodology.
Software Configuration Management	**100.00%**	
- Management of the SCM Process	2	During the project the management roles provide the correct outcome of the SCM process.
- Software Configuration Identification	2	The configuration identifcation is done in the planning game phase.
- Software Configuration Control	2	Configuration control is provided by the tracker role.
- Software Configuration Status Accounting	2	Performed by the tester and tracker.
- Software Configuration Auditing	2	The tester and tracker roles are also responsible for auditing with the reaction of the big boss. Also the element responsible for reviewing is also responsible by the auditing of the work of the other element.
- Software Release Management and Delivery	2	During the final phases of the project all the proper documentation should be produced. Also the small releases policy aids with the management of software release.

Table 4.14: XP: Software Testing, Software Maintenance and Software Configuration Management KA



Knowledge Areas	Satisfaction	Justification
Software Engineering Management	**83.33%**	
- Initiation and Scope Definition	2	The proper initiation and scope definition are done in the exploration phase.
- Software Project Planning	2	Project planning is done at the release planning with priorities defined by value, risk and velocity.
- Software Project Enactment	2	The principles advocate in the methodology emphasizes the commitment to the planning and the big boss is responsible for its adherence.
- Review and Evaluation	2	The review and evaluation is performed continuously during the project, using pair programming and customers approval.
- Closure	2	The productionizing, maintenance and death phase provide the proper closure of the project.
- Software Engineering Measurement	0	No asset to the software engineering procedures are performed.
Software Engineering Process	**83.33%**	
- Process Implementation and Change	2	The process is defined and it's maintained by the product owner which is responsible to change the plan, and the process to adapt to change.
- Process Definition	2	The methodology provides a model/software life cycle and plan.
- Process and Product Assessment/Measurement	1	No measurements to the adherence or assessment of the process is performed, but the product's quality is measured and analyzed.
Software Quality	**100.00%**	
- Software Quality Fundamentals	2	The fundamentals issues on quality are addressed in the methodology by the different roles and artifacts.
- Software Quality Management Processes	2	All the processes within software quality are described in the methodology.

Table 4.15: XP: Software Engineering Management, Software Engineering Process and Software Quality KA

Chapter 5

Conclusions and further work

5.1 Conclusions

This dissertation provides a framework to compare SDMs and to correctly choose one when embracing a new software project. This framework is backed up by SWEBOK and its knowledge areas, providing that the classification and comparison of the SDM is objective and based on facts. Subjectivity and experience can also be added to the framework by weighting the KA according to what is believed to be the KA more important for the project in cause. One of the main problems with software development projects is that normally this activity is not done at all, normally a team or a company uses the same SDM for all the projects and use them until something goes wrong or they decide is time to use another methodology. The fact is that software development projects should use an SDM that suits their demands and not the one that is normally used. Even though experience using an SDM can influence the decision it should not be the only factor. So the result of this dissertation should not be only the framework delivered but a concern with the importance that a correct choice of an SDM can have in the final product. This dissertation also contributes with an analysis of software engineering evolution during the years and the contribution it has given to software development. With the the results of this dissertation is possible to effectively improve the quality of the software produce and the process of building it. In order to achieve it is necessary to maintain the framework that have been delivered by increasing the number of SDM and feeding the analysis with previous experiences, by giving weights not only to KA but also to SDM them selves and updating the classification with

the input of the experience in using those SDM.
The choice of the SDM to include in the framework delivered was made according to the evolution of the SDM in time, confronting different SDM philosophies and to compare the approaches that are being used in industry. These confront led to choosing a set of most used methodologies of each approach/philosophy and analyze them according to the previously explained methodology. This approach helps to mitigate and dismiss some generalizations and prejudices that exist with the SDMs and that supporters of these philosophies tend to have regarding the *"opposite"* philosophies. One of the objectives of this dissertation is also do demonstrate that most of the times a methodology is not better than other but is more appropriate than other given a determined context. In other words, the choice of an SDM is not choosing the *"best SDM ever"* but is actually using this framework for assessing what are the qualities of an SDM by performing the classification against the SWEBOK knowledge areas and putting the classification in context by giving weights to these KA regarding the predicted risks and the environment variables that the project will be involved in.
When concerning the results, and when confronting to critics and similar works was interesting to find out that most of the results were similar, even though different approaches have been taken. The results seem to point that when considering the software engineering process and management the more modern methodologies satisfy these KA without disregarding the KA that the TDM were mainly concerned with. Although the main critic pointed to the ADM is that they should work in theory but fail when actual implementations occur and that the methodologies tend to degrade into an exaggerated concern with the way those activities are performed disregarding that the actual activities, turning the project into chaos where the final product is be delivered with several problems. Translating into the terminology of this dissertation, regarding more the KA dedicated to software engineering process and management but failing into correctly outputting the needs addressed by the KA of software construction, requirements, design and testing. This kind of critic it cannot be supported or declined by the work done in this dissertation, for *proving* the work done here and to clarify some of the doubts and opinions addressed by the different authors an statistical work, trying to assess and to ask for input over different projects using different methodologies, should be done as a support to this dissertation. Of course as was stated previously, this work should be done, taking in consideration the specific constraints of the project, team and stakeholders involved.
Another important aspect when reviewing the work done in this document is to address the subjectivity of the results. The analysis is performed by a person, who carries opinions, experiences and interpretations, although an effort

on minimizing this by trying to provide factual verifications and analysis was done. When reviewing this work and confronting to similar works, even though with different approaches, scopes and goals, was good to verify that the results, even that they differ in the metric, they tend to be similar in the overview (the main goal of this dissertation) given for the same KA/SDM.
In overall, the necessary description of the SDM and KA was provided to understand the results and its justifications. When regarding the extension an intentional confront between different approaches of SDM (TDM and ADM) with the inclusion of a methodology that would not fit into neither of the group representing the link between them, of course some very popular ADM could also be described (e.g. DSDM (Dynamic systems development method), FDD (Feature Driven Development) or Agile Modeling), but the popularity factor of Scrum and eXtrememe Programming was chosen among other factors. The confrontation of TDM/ADM was/is very common in several papers, forums, books and other ways of communication within the software engineering community and were a must have confrontation in this dissertation. Other confrontations could be done and will be addressed in further work section.

5.2 Further Work

Although the objectives of the dissertation were fulfilled and the framework delivered can help to choose an SDM, there are more variables that can be included in the work developed.
The first addition that must be addresses is the scope of SDMs that have been analyzed and used in the classification process. As it was mention previously the choice of this SDMs was made by confronting different approaches and philosophies regarding the evolution of the SDMs in time and the popularity of each SDM in academic and industrial contexts. A group of SDM that was not included in this dissertation is a confrontation that is now in vogue and that could lead to interesting conclusions. These confrontations, more modern, is the increase of discussion between the lean software development methodologies and agile development methodologies and it will be considered as further work.
In the same topic is to broaden the SDM for each group including some popular SDMs that have not been included in this work but were taken in consideration when making the choice. Some examples of these SDMs are: DSDM (Dynamic systems development method), FDD (Feature Driven Development) and Agile Modeling.
The other addition that could increase the quality of the framework delivered is the classification it self, in this work the software engineering approach was developed by using the SWEBOK but it would be also interesting to develop an

additional classification regarding the view of project management by using the same approach with PMBOK and its knowledge areas. This insight could lead to additional weights in the classification and it could broaden the scope of the classification and provide insight of the problems and risks that project managers face in the view some what disconnected from the software development it self and more focused in management giving a different input on the subject.
This dissertation also approached the sense of what software quality means by explaining the different qualities that can be measured and that a software can have. It would also be interesting to analyze how an SDM can influence those particular qualities and to build up a classification for each SDM regarding the qualities it enforces when developing software. This approach is more difficult to explore, keeping in mind the objectivity that was referenced has being one of the goals of that the framework should retrieve. To measure how an SDM can influence the quality of software would not be an easy task, because it could imply analyzing the results of the SDM which brings the subjectivity of the context to the classification. Other approaches could lessen the subjectivity and that would be an important work to add to this framework.
The support to this framework is also considered as further work by providing a tool that can help people using the framework, the ideal scenario would be to build an web application that could be enriched with SDM descriptions and classifications and an front-end that would allow a user to wheight KA and deliver graphical input with a set of choosen SDM. This would be an important work as it should help to materialize the work developed in this dissertation.

Bibliography

[1] Pekka Abrahamsson, Jussi Ronkainen, and Juhani Warsta. Agile development methods - review and analysis. Technical report, University of Oulu, 2002.

[2] D. E. Avison and G. Fitzgerald. *Information Systems Development: Methodologies, Techniques, and Tools*. McGraw-Hill Higher Education, 2nd edition, 1998.

[3] Victor R. Basili. Quantitative evaluation of software methodology. Technical report, Deparment of Computer Science, University of Maryland, 1985. www.cs.umd.edu/~basili/publications/proceedings/P29.pdf.

[4] Victor R. Basili and Robert Reiter JR. A controlled experiment quantitatively comparing software development approaches. *IEEE Transactions on Software Engeneering*, VOL. SE-7(NO. 3), May 1981. http://www.cs.umd.edu/~basili/publications/journals/J10.pdf.

[5] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 2nd edition, 2005.

[6] Barry Boehm. A spiral model of software development and enhancement. VOL 5:61–72, 1988.

[7] Barry Boehm. Tutorial on software risk management. 1989.

[8] Frederick P. Brooks. No silver bullet. *Proceedings of the IFIP Tenth World Computing Conference*, 1986.

[9] Common framework for algebraic specification. http://www.informatik.uni-bremen.de/cofi/wiki/index.php/CoFI.

[10] Edward R. Comer. Alternative software life cycle models. VOL 2:289–299, 1997.

[11] Edgsger W. Dijkstra. Notes on structured programming. Technical report, Technological University Eindhoven,The Netherlands, 1969. http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF.

[12] Edsger Dijkstra. The humble programmer. ACM Turing Lecture.

[13] William R. Duncan, editor. *Guide to Project Management Body of Knowledge*. Project Management Institute - Standards Committee, 1996.

[14] Gamma and all. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[15] Carlo Ghezzi, Mehdi Jazayero, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice-Hall International Editions, 1991.

[16] Kai Gilb. *EVO - Evolutionary Management and Product Developement*. Manuscript from www.gilb.com, 2007.

[17] Tom Gilb. *Principles Of Software Engineering Management*. Addison-Wesley, 1988.

[18] IEEE, editor. *Guide to Software Engineering Body of Knowledge*. IEEE Computer Society - Professional Practices Committee, 2004.

[19] Frederick P. Brooks Jr. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, 1995.

[20] Stephen H. Kan. *Metrics and models in software quality engineering*. Pearson Education, 2nd edition, 2009.

[21] James M. Kerr and Richard Hunter. *Inside RAD: How to Build a Fully Functional Computer System in 90 Days or Less*. McGraw-Hill, 1994.

[22] Riaan Klopper, Stefan Gruner, and Derrick G. Kourie. Assessment of a framework to compare software development methodologies. *Proceeding of SAICSIT 2007, South African institute of computer scientists and information technologists on IT research in developing countries*, 2007.

[23] Henrik Kniber. *Scrum and XP from the trenches - How we do Scrum*. InfoQ - Enterprise Software Development Series, 1999. http://infoq.com/minibooks/scrum-xpfrom-the-trenches.

[24] Craig Larman. *Agile and iterative development: a manager's guide*. Addison-Wesley Professional, 2004.

[25] Nancy Leveson. Medical devices: The therac-25. Appendix of: Safeware: System Safety and Computers.

[26] Lidwell and all. *Universal Design Principles*. Rockport, 1993.

[27] James Martin. *Information Engineering: Desing and Construction*. Prentice-Hall, 1990.

[28] James Martin. *Rapid Application Development*. Prentice-Hall, 1991.

[29] Robert Martin. *Agile Software Development, Principles, Patterns, and Practices*. Prentice-Hall, 2002.

[30] Lampert Meertens. Category theory for program construction by calculation. 1995.

[31] José Nuno Oliveira. Especificação e desenvolvimento formal de programas. 1995.

[32] D.L. Parnas. On the criteria to be used in decomposing systems into modules. 1972.

[33] D.L. Parnas and D.M. Weiss. Active design reviews: principles and practices. 1987.

[34] Brian Randell. The 1968/69 nato software engineering reports. http://homepages.cs.ncl.ac.uk/brian.randell/NATO/NATOReports/index.html.

[35] Ken Schwaber. Scrum guide. 2009.

[36] Ken Schwaber and Mike Beedle. *Agile Software Development with Scrum*. Prentice Hall, 2001.

[37] Ken schwaber google talks. http://video.google.com/videoplay?docid=-7230144396191025011.

[38] Henk Gerard Sol. *A feature analysis of information systems design methodologies: Methodological considerations*. 1983.

[39] Xiping Song and Leon J. Osterweil. Toward objective, systematic design-method comparisons. *IEEE Software Journal*, VOL 9 issue 3, 1992.

[40] Reed Sorensen. A comparison of software development methodologies.

[41] Matt Stephens and Doug Rosenberg. *Extreme Programming Refactored: The Case Against XP*. A! Press, 2003.

[42] Various subscribers. Agile manifesto, 2001. http://agilemanifesto.org.

[43] Hirotaka Takeuchi and Ikujiro Nonaka. The new new product development game. Jan 1986.

[44] Unified modeling language. http://www.uml.org.

[45] Vienna development method. http://www.vdmtools.jp/en/.