MSc Program
Engineering Management

Agile Software Development Methodologies; an Approach to achieve Quality in Software

A Master's Thesis submitted for the degree of
"Master of Science"

supervised by
Professor Peter Kopacek

Amirhassan Aliakbar

0927531

November 2010, Vienna, Austria

Affidavit

I, **AMIRHASSAN ALIAKBAR**, hereby declare

1. that I am the sole author of the present Master's Thesis, "Agile Software Development Methodologies; an Approach to achieve Quality in Software", 84 pages, bound, and that I have not used any source or tool other than those referenced or any other illicit aid or tool, and
2. that I have not prior to this date submitted this Master's Thesis as an examination paper in any form in Austria or abroad.

Vienna,

Signature

ACKNOWLEDGEMENT

I would like to express my sincere gratitude to professor Peter Kopacek for his valuable insights, guidance and direction through the process of writing my theses as my supervisor and overall guidance and continuous support as the Engineering Management Program director. I would like to thank him for the priceless knowledge he provided me with through the program, generously.

To my family, Mom and Dad, who have always been my supports in every stage of my life. Nothing compares to their help and I believe nothing can compensate it.

I also would like to thank all my colleagues and friends in the Engineering Management program. I gratefully enjoyed the collaboration and experience and knowledge exchange them. I need to thank Mr. Franz Kaltseis for his help on giving me a way to his colleagues at Autodesk, in Wels, Austria, and also his colleagues for their indulgent experience sharing.

ABSTRACT

As the computers penetrate more and more in every aspect of man's life, software defects cause more than just inconvenience and latency. Software solutions are used in bigger solutions, therefore the solution itself has grown explosively in size and complexity. Lack of proper methods and management models result in inefficient products with significant absence of quality, so reliability of the software products goes under doubt. As a young topic, software engineering has evolved in a mutant manner through a few decades. Software Development Models were introduced one after another evolutionarily based on the predecessors' shortcomings and problems faced while employing. The Agile Software development methodologies are categorized as one group of methodologies based iterative and incremental model, were introduced in early 00s in the Agile Manifesto for Software Development. Data analysis in this text, based on the surveys and results of employing these methodologies by practitioners of different fields and team and project size world wide, reveals the agile methodologies' success in time and cost reduction and improvement in quality and productivity. The study also shows that these methodologies' success is not limited to the project size, as some of the cons' assumptions and predictions have outlined

Contents

1 INTRODUCTION

Software engineering and its development methodologies as young areas do not have a strong history of trial and error for development like construction industry. However this field is rapidly wide spreading in all areas of human life, its evolution is quickly grown.

The first modern digital computers creation dates back to early 1940s. Firstly, the instructions to operate the machine were wired into it. Like other aspects of this young field of science, which is growing and improving quickly, soon practitioners realized it is necessary to divide this infant into hardware and software.

Then, programming languages were brought to existence in 1950s to deal with scientific, algorithmic and business problems respectively. Ever since programming has been improved to provide solutions to bigger and bigger problems. As the problem gets bigger, the solution it needs gets more complex. So, to manage these complex software solutions, software development methodologies appeared with support of different tools and improvement of programming languages.

It has been almost a decade that, nowadays called, Agile Software Development Methodologies, gained the name agile after publication of the Agile Manifesto. Since then they were the hot topic of software development forums and communities and received appraisal.

Agile methodologies are said to have all the good impacts on the software products, which are developed, based on them. It is also so apparent that the population of proponents of these methodologies is of a way bigger number compared to the opposite group.

One of the impacts of agile methodologies is on quality and quality improvement. This thesis will discuss it through literature, individual surveys and survey results currently available in the literature.

1.1 Thesis Structure

This thesis is divided in three sections:

- Fundamental of Quality and Software Quality
- Software Quality Systems definition, characteristics and Implementation
- Software Development Models
- Discussion on Agile Development Methodologies

1.2 Motivation

As a graduate and fan of software engineering, related issues especially in project management and development approach model were of my interest. Many projects fail to release on time or totally fail to release. This can be caused by lack of robust management and/or shortcomings of methodologies employed and/or usage of wrong method for specific projects.

After a brief review of approaches mainly used in the industry these days, I noted agile methodologies are widely used and uniquely successful.

Since quality in software products is one of my concerns I decided to dedicate this thesis to it in connection with the mostly used methodologies.

2 PROBLEM DESCRIPTION

2.1 Research Approach

This thesis starts with quality definitions and its importance in general meaning and specifically in software industry. Then it moves on to software quality systems and the respective characteristics.

It continues to implementation of such systems and the methods used.

At last, it focuses on the Agile methodologies and discusses their impact on quality and productivity by the means stated before.

2.2 Data Collection

All the statistical data provided in the thesis is referenced. The numbers provided are either in absolute values or percentages that are calculated based on the absolute values quoted from the respective references.

3 QUALITY: DEFINITION AND IMPORTANCE

3.1 Quality: The Definition

Quality is a perceptual, conditional and subjective attribute and can be described differently across different times, situations and individuals. Customers mostly focus on the specification quality of a product and usually compare it to competitors' products. Producers measure the conformance quality or degree to which the product was produced correctly. The American Society for Quality defines quality as: 'A subjective term for which each person has his or her own definition'.

There are typically two types of view of quality: popular and professional. A popular view of quality is an intangible trait which can be discussed, felt, and judged, but cannot be weighed or measured. Terms such as good quality, bad quality, and quality of life exemplify how people talk about something vague, which they don't intend to define. This view reflects the fact that people perceive and interpret quality in different ways. The implication is that quality cannot be controlled and managed, nor can it be quantified. This view is in vivid contrast to the professional view held in the discipline of quality engineering that quality can, and should, be operationally defined, measured, monitored, managed, and improved. Another popular view is that quality connotes luxury, class, and taste. Expensive, elaborate, and more complex products are regarded as offering a higher level of quality than their humbler counterparts. Therefore, a surround-sound hi-fi system is a quality system, but a single-speaker radio is not. According to this view, quality is restricted to a limited class of expensive products with sophisticated functionality and items that have a touch of class. Simple, inexpensive products can hardly be classified as quality products.

Numerous definitions and methodologies have been modified and various techniques and concepts have evolved in order to effectively improve the management of product or service quality. There are a number of quality-related functions within a business of which the most common ones are quality assurance, which is the prevention of defects, and quality control, which is the detection of defects, most commonly associated with testing.

Although, the definition may vary across different fields and through times, there are a number of popular definitions in literature and practice. These definitions are accepted and used commonly in different fields. Some of them are as follow:

- "Conformance to requirements". (Crosby, 1979)
- "Fitness for use". (Juran, 1974)
- "Must-be quality" and "attractive quality." The former is near to "fitness for use" and the latter is what the customer would love, but has not yet thought about. (Kano, 1984)
- "Quality is an intangible trait; it can be discussed, felt, and judged, but cannot be weighed or measured", described as the "popular" view in contrast with "professional" view. (Stephen H. Kan, 2003)
- "Quality in a product or service is not what the supplier puts in. It is what the customer gets out and is willing to pay for." (Drucker, 1985)
- "Degree to which a set of inherent characteristics fulfills requirements." (ISO 9000:2005)

The misconceptions and vagueness of the popular views do not help the quality improvement effort in the industries. To that end, quality must be described in a workable definition. The two definitions by Crosby and Juran are related and consistent. These definitions of quality have been adopted and used by many quality professionals.

"Conformance to requirements" implies that requirements must be clearly stated such that they cannot be misunderstood. Then, in the development and production process, measurements are taken regularly to determine conformance to those requirements. The non-conformances are regarded as defects, the absence of quality. As an example, one requirement or specification for a certain radio may be that it must be able to receive certain frequencies more than X miles away from the source of broadcast. If the radio fails to do so, then it does not meet the quality requirements and should be rejected.

The “fitness for use” definition takes customers' requirements and expectations into account, which involve whether the products or services fit their uses. Since different customers may use the products in different ways, it means that products must possess multiple elements of fitness for use. According to Juran, each of these elements is a quality characteristic and all of them can be classified into categories known as parameters for fitness for use. The two most important parameters are quality of design and quality of conformance.

Quality of design in popular terminology is known as grades or models, which are related to the spectrum of purchasing power. The differences between grades are the result of intended or designed differences. Like in automobile industry, all cars provide to the user the service of transportation. However, models differ in size, comfort, performance, style, economy, and status. In contrast, quality of conformance is the extent to which the product conforms to the intent of the design. In other words, quality of design can be regarded, as the determination of requirements and specifications and quality of conformance is conformance to requirements. The two definitions of quality therefore, are essentially similar. The difference is that the fitness for use concept implies a more significant role for customers' requirements and expectations.

In industry it is commonly stated “Quality drives productivity.” Improved productivity is a source of greater revenues, employment opportunities and technological advances. Most discussions of quality refer to a finished part, wherever it is in the process. Inspection, which is what quality insurance usually means, is historical, since the work is done. The best way to think about quality is in process control. If the process is under control, inspection is not necessary.

However, there is one characteristic of modern quality that is universal. In the past, efforts to improve quality, typically defined as producing fewer defective parts, it was done at the expense of increased cost, increased task time, longer cycle time, etc. However, when modern quality techniques are applied correctly to business, engineering, manufacturing or assembly processes, all aspects of quality, customer satisfaction and fewer defects or errors, cycle time and task time, productivity and total cost, etc.- must all improve or, if one of these aspects does not improve, it must at least stay stable and not decline. So modern quality has the characteristic that it creates AND-based benefits, not OR-based benefits.

In other hand, another view of quality is that it is defined entirely by the customer or end user, and is based upon that person's evaluation of his or her entire customer experience. The customer experience is defined as the aggregate of all the interactions that customers have with the company's products and services.

All in all, the fact is organizations are at different stages on the journey towards perfection, and at each stage they will have unique definitions of quality. Therefore, the best definition is which is accurately defines quality, at each stage, for any organization undertaking the continual improvement journey. It is necessary to reflect the fact that, during each stage of the journey, the organization will be aiming for something different from what was aimed for previously. So the definition of quality needs to be variable, not fixed; this is in line with the spirit of Institute of Quality Assurance’s own definition of quality: “A degree of excellence”.

3.2 Quality: The Importance

The simplest idea about the importance of quality in a business is quality product or service is what customers are looking for. Like the definition of quality, which is not absolute, there is not just one reason for importance of quality. Quality can be the key to success in the competitive market but not only by gaining bigger number of customers but also by reduction in costs of any possible kind and therefore more efficiency and profitability.

Man instinctively responds to good quality product. A company that is reputed to consistently provide quality products is bound to have a bigger share of the market and this means high patronage and profits. The approach to achieve constant quality is profitable itself. A quality product does not come out of an arbitrary system. The whole life cycle must be managed through a quality management system.

Quality management is a new phenomenon in production and service. However it can be tracked back to the old age, the new concept was introduced in the 20th century.

Traditionally, efforts to improve quality have centered on the end of the product development cycle by emphasizing the detection and correction of defects. On the contrary, the new approach to enhancing quality encompasses all phases of a product development process from a requirements analysis to the final delivery of the product to the customer and even further to after sales service. Every step in the development process must be performed to the highest possible standard.

The relationship between quality and productivity may seem conflicting to many individuals. Despite many managers think quality and improvement is almost impossible without reduction in productivity and increase in costs, the modern approach to quality is a way to reduce costs and increase productivity and therefore profitability.

Production can be drawn as a measure of output from a production process, per unit of input. Production is a process of combining various inputs in order to make something for consumption, which is the output. The input resources needed to produce a bad product is just equal to those needed to make a good one. The more rework needed for the defects increases the value for the input so the productivity faces reduction. If the product is made right first time, no rework is needed therefore productivity grows. Quality improvement is a potential medium to increase productivity by reducing defective outputs and resources used for rework.

3.3 Software Quality: The Definition

The question “What is software quality?” evokes many different answers. From the previous discussion, quality is a complex concept in general itself, it means different things to different people, and it is highly context dependent. Garvin (1984) Analysis in “What Does “Product Quality” Really Mean” reveals how software quality is perceived in different ways in different domains, such as philosophy, economics, marketing, and management. Kitchenham and Pfleeger's (1996) article, "Software Quality: The Elusive Target.", gives a succinct exposition of software quality. They discuss five views of quality in a comprehensive manner as follows:

- *Transcendental View:* It envisages quality as something that can be recognized but is difficult to define. The transcendental view is not specific to software quality alone but has been applied in other complex areas of everyday life.

- *User View:* It perceives quality as fitness for purpose. According to this view, while evaluating the quality of a product, one must ask the key question: "Does the product satisfy user needs and expectations?"

- *Manufacturing View:* Here quality is understood as conformance to the specification. The quality level of a product is determined by the extent to which the product meets its specifications.

- *Product View:* In this case, quality is viewed as tied to the inherent characteristics of the product. A product's inherent characteristics, that is, internal qualities, determine its external qualities.

- *Value-Based View:* Quality, in this perspective, depends on the amount a customer is willing to pay for it.

The concept of software quality and the efforts to understand it in terms of measurable quantities date back to middle of 1970s. McCall, Richards, and Walters' (1977) "Factors in Software Quality" was the first study on the concept of software quality in terms of quality factors and quality criteria. A quality factor represents a behavioral characteristic of a system. Some examples of high-level quality factors are correctness, reliability, efficiency, testability, maintainability, and reusability. A quality criterion is an attribute of a quality factor that is related to software development. Various software quality models have been proposed to define quality and its related attributes. The most influential ones are the ISO 9126(Quality Management Systems, ISO 9004:2000) and the CMM.

The ISO 9126 quality model was developed by an expert group under the aegis of the International Organization for Standardization. The document ISO 9126 defines six broad, independent categories of quality characteristics: functionality, reliability, usability, efficiency, maintainability, and portability.

The CMM was developed by the Software Engineering Institute (SEI) at Carnegie Mellon University. In the CMM framework, a development process is evaluated on a scale of 1 to 5, commonly known as level 1 through level 5. For example, level 1 is called the initial level, whereas level 5, named optimized, is the highest level of process maturity.

In terms of software engineering, software quality defines how well software is designed, and how well the software conforms to that design. Whereas quality of conformance is concerned with implementation, quality of design measures how valid the design and requirements are in creating a worthwhile product.

Software quality may be defined as conformance to explicitly defined functional and performance requirements, explicitly documented development standards and implicit characteristics that are expected of all professionally developed software.

Software requirements are the foundations from which quality is measured. Lack of conformance to requirement is lack of quality. Specified standards define a set of development criteria that guide the manager is software engineering. If criteria are not followed lack of quality will almost result. A set of implicit requirements often goes unmentioned, like for example ease of use, maintainability etc. If software confirms to its explicit requirement but fails to meet implicit requirements, software quality is suspected.

A definition by Steve McConnell (1993) in his "Code Complete" divides software into two pieces: internal and external quality characteristics. External quality characteristics are those parts of a product that face its users, where internal quality characteristics are those that do not.

Conformance to requirements, completeness, absence of bugs and fault-tolerance, and the most important, reliability, can be named as external quality characteristics. Among the internal characteristics, source code quality is one of the most important internal factors. Reliability is a high important facet, which increases software quality importance in a significant pace as will be discussed in the next sections.

Although a computer has no concept of "well-written" source code, from a human point of view, source code can be written in a way that has an effect on the effort needed to comprehend its behavior. Many source code programming style guides which often stress readability and usually language-specific conventions are aimed at reducing the cost of source code maintenance. Readability, low complexity, low resource consumption and robust error handling are some of characteristics of a quality source code.

3.4 Software Quality: The Importance

With software embedded into many devices today, software failure has caused more than inconvenience. Software errors have caused chaos and disorders in places like train stations, production lines, services like banking, telecommunications and even have caused human fatalities. The causes have ranged from poorly designed user interfaces to direct programming errors.

As an example, between June 1985 and January 1987, a computer controlled radiation therapy machine, called the Therac-25, massively overdosed six people. Therac-25 was produced by Atomic Energy of Canada Limited (AECL) based on the previous machines, Therac-6 and Therac-20 with some advantages and improvements. It needed less mechanism required to accelerate the electrons and it was also more economical to produce. Compared to Therac-20, Therac-25 is more compact, more versatile, and arguably easier to use. The customer could also gain economic advantages, since only one machine was required for both treatment modalities: electrons and photons.

In addition, Therac-25 software had more responsibility for maintaining safety than the software in the previous machines. Therac-20 had independent protective circuits for monitoring the electron-beam scanning plus mechanical interlocks for policing the machine and ensuring safe operation. Therac-25 relied more on software for these functions. AECEL took advantage of the computer's abilities to control and monitor the hardware and decided not to duplicate all the existing hardware safety mechanisms and interlocks. The same Therac-6 package was used by the software developers at AECL to start the Therac-25 software. Therac-20 and Therac-25 software programs were done independently starting from a common base. The reuse of Therac-6 design features or modules explain some of the problematic aspects of the Therac-25 software design. The whole story about the disaster made by Therac-25 is well argued in Dr. Nancy Leveson's (1993) paper, Medical Devices: The Therac-25.

From the case Therac-25, the importance of reliability can be effortlessly derived. One of the most significant quality issues can be reliability. Software reliability is an important facet of software quality. It is defined as "the probability of failure-free operation of a computer program in a specified environment for a specified time". One of reliability's distinguishing characteristics is that it is objective, measurable, and can be estimated, whereas much of software quality is subjective criteria. This distinction is especially important in the discipline of Software Quality Assurance. These measured criteria are typically called software metrics.

The need for a means to objectively determine software reliability comes from the desire to apply the techniques of contemporary engineering fields to the development of software. That desire is a result of the common observation, by both laypersons and specialists, that computer software does not work the way it ought to. In other words, software is seen to exhibit undesirable behaviors, up to and including outright failure, with consequences for the data which is processed, the machinery on which the software runs, and by extension the people and materials which those machines might negatively affect. The more critical the application of the software to economic and production processes, or to life-sustaining systems, the more important is the need to assess the software's reliability.

Regardless of the criticality of any single software application, it is also more and more frequently observed that software has penetrated deeply into most every aspect of modern life through the technology that is used. It is only expected that this infiltration will continue, along with an accompanying dependency on the software by the systems, which maintain the societies. As software becomes more and more crucial to the operation of the systems on which man depends, the argument goes, it only follows that the software should offer a concomitant level of dependability. In other words, the software should behave in the way it is intended, or even better, in the way it should.

The fundamental problem in the issue of measuring software reliability, which is the difficulty of determining, in advance, exactly how the software is intended to operate. The problem seems to stem from a common conceptual error in the consideration of software, which is that software in some sense takes on a role, which would otherwise be filled by a human being. This is a problem on two levels. Firstly, most modern software performs work, which a human could never perform, especially at the high level of reliability that is often expected from software in comparison to humans. Secondly, software is fundamentally incapable of most of the mental capabilities of humans, which separate them from mere mechanisms: qualities such as adaptability, general-purpose knowledge, a sense of conceptual and functional context, and common sense.

Nevertheless, most software programs could safely be considered to have a particular, even singular purpose. If the possibility can be allowed that said purpose can be well or even completely defined, it should present a means for at least considering objectively whether the software is, in fact, reliable, by comparing the expected outcome to the actual outcome of running the software in a given environment, with given data. Unfortunately, it is still not known whether it is possible to exhaustively determine either the expected outcome or the actual outcome of the entire set of possible environment and input data to a given program, without which it is probably impossible to determine the program's reliability with any certainty.

However, various attempts are in the works to attempt to rein in the vastness of the space of software's environmental and input variables, both for actual programs and theoretical descriptions of programs. Such attempts to improve software reliability can be applied at different stages of a program's development. These stages principally include: requirements, design, programming, testing, and runtime evaluation.

Beside all discussed about software quality, it may seem to some that it is costly. As mentioned in the section related to importance of product quality in general, quality is equal to more cost to many. But in the modern concepts, quality management leads to lower cost and input with better output and bigger profit. As software industry itself is a young field, injecting new concept of quality was done in its early ages. Many standards and methods and techniques have been, and some are still, used to manage the development of quality software and improving it. These methods and techniques, just like methods in other industries, reduce defect rate, cost and time and increase efficiency and quality of the software.

3.4.1 The Economic Value of Software Quality

The economic value of software quality is not well covered in the soft- ware engineering literature. There are several reasons for this problem. One major reason is the rather poor measurement practices of the software engineering domain. Many cost factors such as unpaid overtime are routinely ignored. In addition, there are frequent gaps and omissions in software cost data, such as omission of project management costs and the omission of part-time specialists such as technical writers. In fact, only the effort and costs of coding have fairly good data available. Everything else, such as requirements, design, inspections, testing, quality assurance, project offices, and documentation tend to be underreported or ignored.

The software engineering literature depends too much on vague and unpredictable definitions of quality. The unscientific definitions slow down research on software quality economics.

Two other measurement problems also affect quality economic studies. These problems are the usage of two invalid economic measures: cost per defect and lines of code. Cost per defect penalizes quality and achieves its lowest costs for the buggiest applications. Lines of code penalizes high-level programming languages and disguises the value of high-level languages for studying either quality or productivity.

Software quality does have value, and the value increases, as application sizes get bigger. In fact, without excellence in quality control, even completing a large software application is highly unlikely. Completing it on time and within budget in the absence of excellent quality control is essentially impossible.

4 SOFTWARE QUALITY SYSTEM

Starting a software quality program from scratch is time consuming and a task often doomed to failure before it is begun. Inadequate preparation, misused terms, lack of planning, and failure to recognize the roles of all individuals in the organization are only a few of the pitfalls waiting for the overanxious practitioner.

There are two goals of software quality systems. The first goal is to build quality into the software from the beginning. This means assuring that the problem or need to be addressed is clearly and accurately stated, and that the requirements for the solution are properly defined, expressed, and understood. Nearly all the elements of software quality systems are oriented toward requirements validity and satisfaction.

In order for quality to be built into the software system from its inception, the software requirements must be clearly understood and documented. Until the actual requirements, and the needs of the user that they fulfill, are known and understood, there is little likelihood that the user will be satisfied with the software system that is delivered. Whether they are all known before the start, or some will be learned as it goes, all requirements must be known and satisfied before getting through the project.

The second goal of software quality systems is to keep that quality in the software throughout the software life cycle.

The elements of software quality systems are standards, reviewing, testing, defect analysis, configuration management, security, safety, risk management.

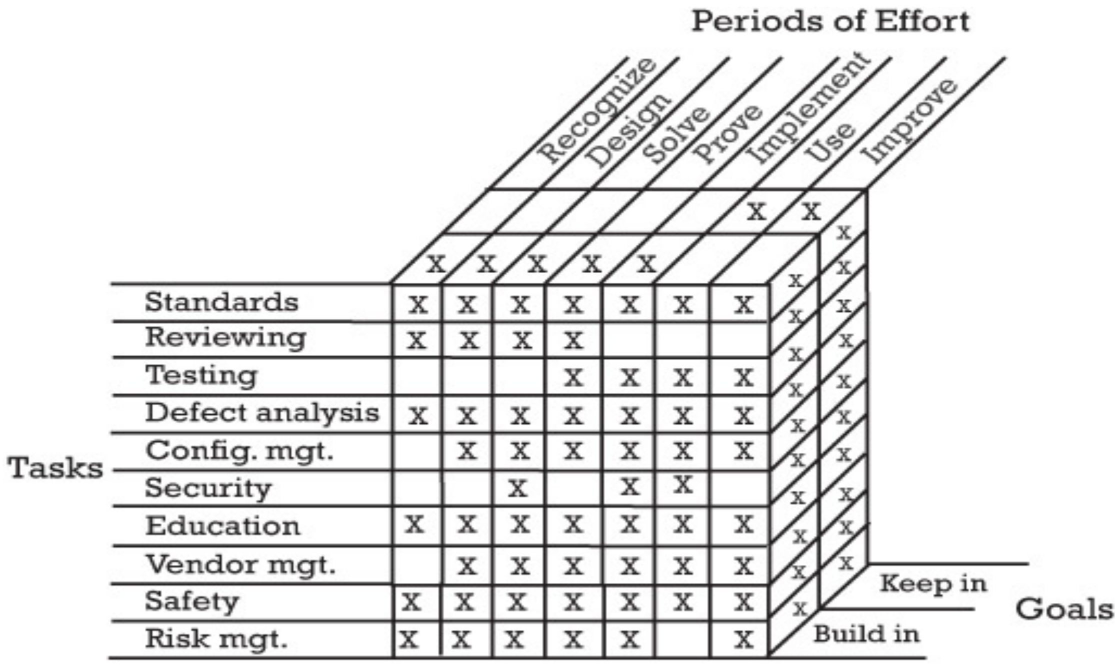
While each element can be shown to contribute to both goals, there are heavier relationships between some elements and one or the other of the two goals.

Every software life cycle model has divisions, or periods of effort, into which the work of developing and using the software is divided. These divisions or periods are given various names depending on the particular life-cycle paradigm being applied. For this discussion, the following periods of effort, together with their common names, are defined:

- Recognition of a need or problem.
- Definition of the software solution to be applied.
- Development of the software that solves the problem or satisfies the need.
- Proving that the solution is correct.
- Implementing the solution.
- Using the solution.
- Improving the solution.

Regardless of their names, each division represents a period of effort directed at a particular part of the overall life cycle. They may be of various lengths and be applied in various sequences, but they all exist in successful projects.

There are also associations between certain elements and the various divisions or periods of the software life cycle. Figure 4.1 displays the ten elements as a cube supporting the goals of software quality and the periods of the software life cycle with which each element is most closely associated.



Tasks	Recognize	Design	Solve	Prove	Implement	Use	Improve
Standards	X	X	X	X	X	X	X
Reviewing	X	X	X	X			
Testing				X	X	X	X
Defect analysis	X	X	X	X	X	X	X
Config. mgt.		X	X	X	X	X	X
Security			X		X	X	
Education	X	X	X	X	X	X	X
Vendor mgt.		X	X	X	X	X	X
Safety	X	X	X	X	X	X	X
Risk mgt.	X	X	X	X	X		X

Figure 4.1: Quality tasks, life-cycle periods, and goals (Horch, 2003)

4.1 Standards

The old days of free-form creativity in the development of software are gradually giving way to more controlled and scientific approaches. Software is moving from an arcane art to a visible science.

Standards are intended to provide consistent, rigorous, uniform, and enforceable methods for software development and operation activities. The development of standards, whether by professional societies such as the Institute of Electrical and Electronics Engineers (IEEE), international groups such as International Organization for Standardization/International Electrotechnical Commission Joint Technical Committee One (ISO/IEC JTC1), industry groups, or software development organizations for themselves, is recognizing and furthering that movement.

Standards cover all aspects of the software life cycle, including the very definition of it itself. More, probably, than any of the other elements, standards can govern every phase of the life cycle. Standards can describe considerations to be covered during the concept exploration phase. They can also specify the format of the final report describing the retirement of a software system that is no longer in use.

Standards come into being for many reasons. They might document experience gained in the day-to-day running of a computer center, and the most efficient methods to be used. Laws and government regulations often impose standard procedures on business and industry. Industries can band together to standardize interfaces between their products such as in the communications areas. Contracts often specify standard methods of performance. And, in many cases, standards arise out of good common sense.

Standards have several characteristics such as necessity; because no standard will be observed for long if there is no real reason for its existence, feasibility; as common sense states, if it is not possible to comply with the tenets of a standard, then it will be ignored and, measurability, It must be possible to demonstrate that the standard is being followed.

Software standards should be imposed so that the developer of a software product or component can pay attention to the technical aspects of the task, rather than to the routine aspects that may be the same for every task. Standards, such as those for document formats, permit the producer to concentrate on technical issues and content rather than format or layout details.

Standards, while worthwhile, are less than fully effective if they are not supported by policies that clearly indicate their imposition. Specific practices for standard implementation are often useful. In this way, adherence to the standard may be more uniform.

4.2 Reviewing

Reviews permit ongoing visibility into the software development and installation activities.

Product reviews, also called technical reviews, are examinations of products and components throughout the development phases of the life cycle. They are conducted throughout the software development life cycle. Informal reviews generally occur during development life cycle phases, while formal reviews usually mark the ends of the phases.

4.3 Testing

Tests provide increasing confidence and, ultimately, a demonstration that the software requirements are being satisfied. Test activities include planning, design, execution, and reporting.

Test planning begins during the requirements phase and parallels the requirements development. As each requirement is generated, the corresponding method of test for that requirement should be a consideration. A requirement is faulty if it is not testable. By starting test planning with the requirements, non-testability is often avoided. In the same manner that requirements evolve and change throughout the software development, do the test plans evolve and change. This emphasizes the need for early, and continuing, configuration management of the requirements and test plans.

Test design begins as the software design begins. Here, a parallel effort with the software development is appropriate. As the design of the software takes form, the test cases, scenarios, and data are developed that will exercise the designed software. Each test case also will include specific expected results so that a pass-fail criterion is established. As each requirement must be measurable and testable, so must each test be measurable. A test whose completion is not definitive tells little about the subject of the test. Expected results give the basis against which the success or failure of the test is measured.

Actual testing begins with the debugging and early unit and module tests conducted by the programmer. Formal test execution generally begins with integration tests in which modules are combined into subsystems for functional testing. In larger systems, it is frequently advisable to begin formal testing at the module level after the programmer has completed his or her testing and is satisfied that the module is ready for formal testing.

4.4 Defect analysis

Defect analysis is the combination of defect detection and correction, and defect trend analysis. Defect detection and correction, together with change control, presents a record of all discrepancies found in each software component. It also records the disposition of each discrepancy, perhaps in the form of a software problem report or software change request.

Each needed modification to a software component, whether found through a walk-through, review, test, audit, operation, or other means is reported, corrected, and formally closed. A problem or requested change may be submitted by anyone with an interest in the software. The situation will be verified by the developers, and the configuration activity manager will agree to the change. Verification of the situation is to assure that the problem or need for the change actually exists. Configuration manager may wish to withhold permission for the change or delay it until a later time; perhaps because of concerns such as interference with other software, schedule and budget considerations, the customer's desires, and so on. Once the change is completed and tested, it will be reported by configuration manager to all concerned parties, installed into the operational software by the developers or operations staff, and tested for functionality and compatibility in the full environment.

4.5 Configuration management

Configuration management is a three-fold discipline. Its intent is to maintain control of the software, both during development and after it is put into use and changes begin.

Configuration management is, in fact, three related activities: identification, control, and accounting. Each of the activities has a distinct role to play. As system size grows, so does the scope and importance of each of the activities. As systems grow and become more complex, or as changes to the system become more important, each activity takes on a more definite role in the overall management of the software and its integrity.

4.6 Security

Another frequent damager of the quality of output of an otherwise high-quality software system is data that has been unknowingly modified. If the data on which the system is operating has been made inaccurate, whether intentionally or by accident, the results of the software will not be correct. To the user or customer, this appears to be inadequate software.

Additionally, though not really a software quality issue per se, is the question of theft of data. The security of stored or transmitted data is of paramount concern in most organizations. From the theft of millions of dollars by interception of electronic funds transfers to an employee who just changes personnel or payroll records, data security is a major concern.

Finally, the recent onslaught of hackers and software attackers and the burgeoning occurrences of viruses also need to be considered. These threats to software quality must be recognized and countered.

The software quality practitioner is responsible for alerting management to the absence, or apparent inadequacy, of security provisions in the software. In addition, the software quality practitioner must raise the issue of data center security and disaster recovery to management's attention.

4.7 Education

Education assures that the people involved with software development, and those people using the software once it is developed, are able to do their jobs correctly.

It is important to the quality of the software that the producers be educated in the use of the various development tools at their disposal. Different programming languages, the use of operating systems, data modeling techniques, debugging tools, special workstations, and test tools must be taught before they can be applied beneficially.

The proper use of the software once it has been developed and put into operation is another area requiring education. In this case, the actual software user must be taught proper operating procedures, data entry, report generation, and whatever else is involved in the effective use of the software system's capabilities. This is one of the issues got challenged in Dr. Nancy Leveson’s (1993) paper, Medical Devices: The Therac-25.

The data center personnel must be taught the proper operating procedures before the system is put into full operation. Loading and initializing a large system may not be a trivial task. Procedures for recovering from abnormal situations may be the responsibility of data center personnel. Each of the many facets of operating a software system must be clear so that the quality software system that has been developed may continue to provide quality results.

The software quality practitioner is not usually the trainer or educator. These functions are normally filled by some other group or means. The role of the software quality practitioner is, as always, to keep management attention focused on the needs surrounding the development and use of a quality software system. In this case, the software quality practitioner is expected to monitor the requirements for, and the provision of, the education of the personnel involved in the software life cycle.

Lastly, the support personnel surrounding software development must know their jobs. The educators, configuration managers and software quality practitioners, security and database administrators, and so on must be competent to maintain an environment in which quality software can be built, used, and maintained.

4.8 Vendor management

When software is purchased, the buyer must be aware of, and take action to gain confidence in, its quality. Not all purchased software can be treated in the same way, as will be demonstrated here. Each type of purchased software will have its own software quality system approach, and each must be handled in a manner appropriate to the degree of control the purchaser has over the development process used by producer. The following are three basic types of purchased software (Horch, 2003):

- Off-the-shelf;
- Tailored shell;
- Contracted.

Off-the-shelf software is a package that is bought at the store. Microsoft Office, Adobe Photoshop, virus checkers, and the like are examples. These packages come as they are with no warrantee that they will do what you need to have done. They are also almost totally outside the buyer's influence with respect to quality.

The second category may be called the tailored shell. In this case, a basic, existing framework is purchased and the vendor then adds specific capabilities as required by the contract. This is somewhat like buying a stripped version of a new car and then having the dealer add a stereo, sunroof, and other extras. The only real quality influence is over the custom-tailored portions.

The third category is contracted software. This is software that is contractually specified and provided by a third-party developer. In this case, the contract can also specify the software quality activities that the vendor must perform and which the buyer will audit. The software quality practitioner has the responsibility in each case to determine the optimum level of influence to be applied, and how that influence can be most effectively applied. The purchaser's quality practitioners must work closely with the vendor's quality practitioners to assure that all required steps are being taken.

Attention to vendor quality practices becomes extremely important when the developer is offshore or remote.

4.9 Safety

As computers and software grow in importance and impact more and more of our lives, the safety of the devices becomes a major concern. The literature records overdoses of medicines, lethal doses of radiation, space flights gone astray, and other catastrophic and near-catastrophic events. Every software project must consciously consider the safety implications of the software and the system of which it is a part. The project management plan should include a paragraph describing the safety issues to be considered. If appropriate, a software safety plan should be prepared.

4.10 Risk management

There are several types of risk associated with any software project. Risks range from the simple, such as the availability of trained personnel to undertake the project, to more threatening, such as improper implementation of complicated algorithms, to the deadly, such as failure to detect an alarm in a nuclear plant. Risk management includes identification of the risk; determining the probability, cost, or threat of the risk; and taking action to eliminate, reduce, or accept the risk. Risk and its treatment is a necessary topic in the project plan and may deserve its own risk management plan.

It is hard to achieve quality software products without a quality system within an environment.

5 SOFTWARE QUALITY SYSTEM IMPLEMENTATION

The elements of a software quality system must be assembled into a manageable whole. As it begins to implement the individual elements into the software quality system, each organization must select the method and order of implementation and ensure that sufficient support is present for a successful implementation and that the software quality system will become part of the new quality culture.

The planning of a software quality system should involve consideration of all the elements discussed. Prior to beginning any actual implementation, careful consideration must be given to each step that will be taken. Those software quality system elements that are already in place or that are partially implemented must be recognized and built on to the maximum extent compatible with the overall system. Each activity must be assigned to the appropriate organizational entity for execution. Inclusion of each group to be monitored in the planning process will benefit the overall system by instilling a sense of system ownership in the whole organization.

The actual implementation of the software quality system plan requires careful planning and scheduling. Starting with the definition of the charter of the software quality practitioners and ending with the software quality system implementation strategy and execution, each step must be laid out and accomplished with the maximum involvement of the affected groups.

There are several strategies for implementing a software quality system. Probably the least effective methods are the ones that impose the software quality system on the whole development organization without regard to which stage each project is at in its software development life cycle.

First is the all-at-once approach. Each project is expected to stop what it is doing and to bring the project in line with the new software quality system requirements, whether or not every requirement is meaningful. The result is usually a period of confusion and a corresponding antagonism toward the software quality system and the software quality group. Faced with this negative attitude, the software quality group has a very difficult time establishing itself and often fails and is disbanded.

Another poor method is the one-element-at-a-time approach. In this case, a particular element is chosen for organization-wide implementation, again without regard to the status of the various ongoing projects. Since there is varied success based on the position of each project in its software life cycle, the element tends to fade away due to decreasing application. When it is realized that element is ineffective, the decision is made to try one of the others. It fails eventually. As each element is tried in turn, each faces the same fate. Finally, the decision is made to scrap the software quality system because it is obviously not effective.

Both these implementation methods can work if consideration is given to each project to which they will be applied. There must be recognition that each project will be in a different portion of its life cycle and thus will have differing abilities, or needs, to comply with a new software quality system. Provisions for deviations from, or waivers of, specific requirements of the software quality system based on the projects' needs must be allowed, which will make either method of implementation much more likely to succeed.

The all-at-once approach can be successful when the software quality system is to be applied only to new projects.

A combination of the two methods can be the best answer in most cases. As is the case in any discussion of methods or approaches, there is no single, always correct situation.

The single-project and single-element approaches are clearly the extremes of the implementation method spectrum. The single-project approach would be successful in the information systems organization that had no ongoing development projects to consider. The single-element approach could be the best answer if there is no new project activity. Neither of these situations is likely to be the case in most organizations. The answer, obviously, is to fit the implementation method, or combination of implementation methods, to the actual experience of the particular organization and to the specific projects being affected.

For new projects, it is almost always best to implement as much of the total system as possible. Only those elements that, in a given organization, would conflict with ongoing projects should be delayed. An example might be a new form of database security system that would seriously impact an ongoing development effort. In most cases, however, new projects can be started using the full software quality system with little or no impact on the rest of the development activity.

Ongoing projects can be the subject of various subsets of the full software quality system, depending on their status and needs. Projects late in the development life cycle probably would be unaffected by the imposition of new programmer training but could benefit from increased user training requirements. A project early in the development life cycle can be placed under more stringent configuration management procedures without much impact on completed work. Each project must be evaluated against the full software quality system, and those elements that are feasible should be implemented.

As the software quality system is implemented and experience is gained with it, it should be evaluated and modified as appropriate. The experiences of each project should be considered and changes, additions, and deletions made. Provisions for deviations and waivers will make the actual implementation of each element to each project as smooth as possible. A study of the waivers and deviations will show the modifications that may be needed in the overall system.

Once a quality system is implemented, there is a need in lower levels to inject quality in developments of the software products.

Software metrics and models cannot be discussed in a vacuum; they must be referenced to the software development process.

An overview of and discussion on the well-known process models seems necessary in order to derive a better approach. The waterfall process life-cycle model, the prototyping approach, the spiral model, the object-oriented development process and the iterative development process, Extreme Programming and Scrum are as follows.

5.1 The Waterfall Process Model

In the 1960s and 1970s software development projects were characterized by massive cost overruns and schedule delays; the focus was on planning and control. The emergence of the waterfall process to help tackle the growing complexity of development projects was a logical event. The Waterfall Process Model was first introduced in an article written by Winston Royce (1970), primarily intended for use in government projects. It encourages the development team to specify what the software is supposed to do, in other means, gather and define system requirements, before developing the system. It then breaks the complex mission of development into several logical steps with intermediate deliverables that lead to the final product. To ensure proper execution with good quality deliverables, each step has validation, entry, and exit criteria.

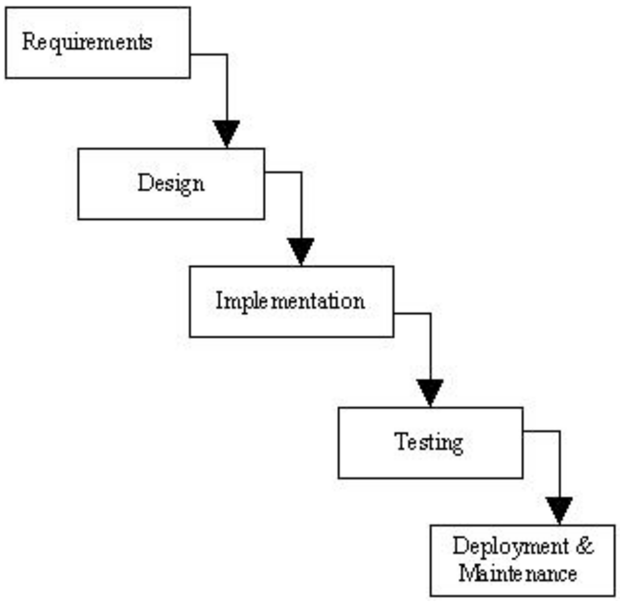


Figure 5.1 - The Waterfall Process Model.

The divide-and-conquer approach of the waterfall process has several advantages. It enables more accurate tracking of project progress and early identification of possible slippages. It forces the organization that develops the software system to be more structured and manageable. This structural approach is very important for large organizations with large, complex development projects. It demands that the process generate a series of documents that can later be used to test and maintain the system. The bottom line of this approach is to make large software projects more manageable and delivered on time without cost overrun. Experiences of the past several decades show that the waterfall process is very valuable. Many major developers, especially those who were established early and are involved with systems development, have adopted this process. This group includes commercial corporations, government contractors, and governmental entities.

Although a variety of names have been given to each stage in the model, the basic methodologies remain more or less the same. Thus, the system-requirements stages are sometimes called system analysis, customer-requirements gathering and analysis, or user needs analysis; the design stage may be broken down into high-level design and detail-level design; the implementation stage may be called code and debug; and the testing stage may include component-level test, product-level test, and system-level test.

Figure 5.2 shows an implementation of the waterfall process model for a large project. The requirements stage is followed by a stage for architectural design. When the system architecture and design are in place, design and development work for each function begins.

This consists of high-level design, low-level design, code development, and unit testing. Despite the waterfall concept, parallelism exists because various functions can proceed simultaneously. As shown in the figure, the code development and unit test stages are also implemented iteratively. Since unit testing is an integral part of the implementation stage, it makes little sense to separate it into another formal stage.

Before the completion of the high-level design, low-level design, and code, formal reviews and inspections occur as part of the validation and exit criteria. These inspections are called I0, I1, and I2 inspections, respectively.

When the code is completed and unit tested, the subsequent stages are integration, component test, system test, and early customer programs.

The final stage is release of the software system to customers.

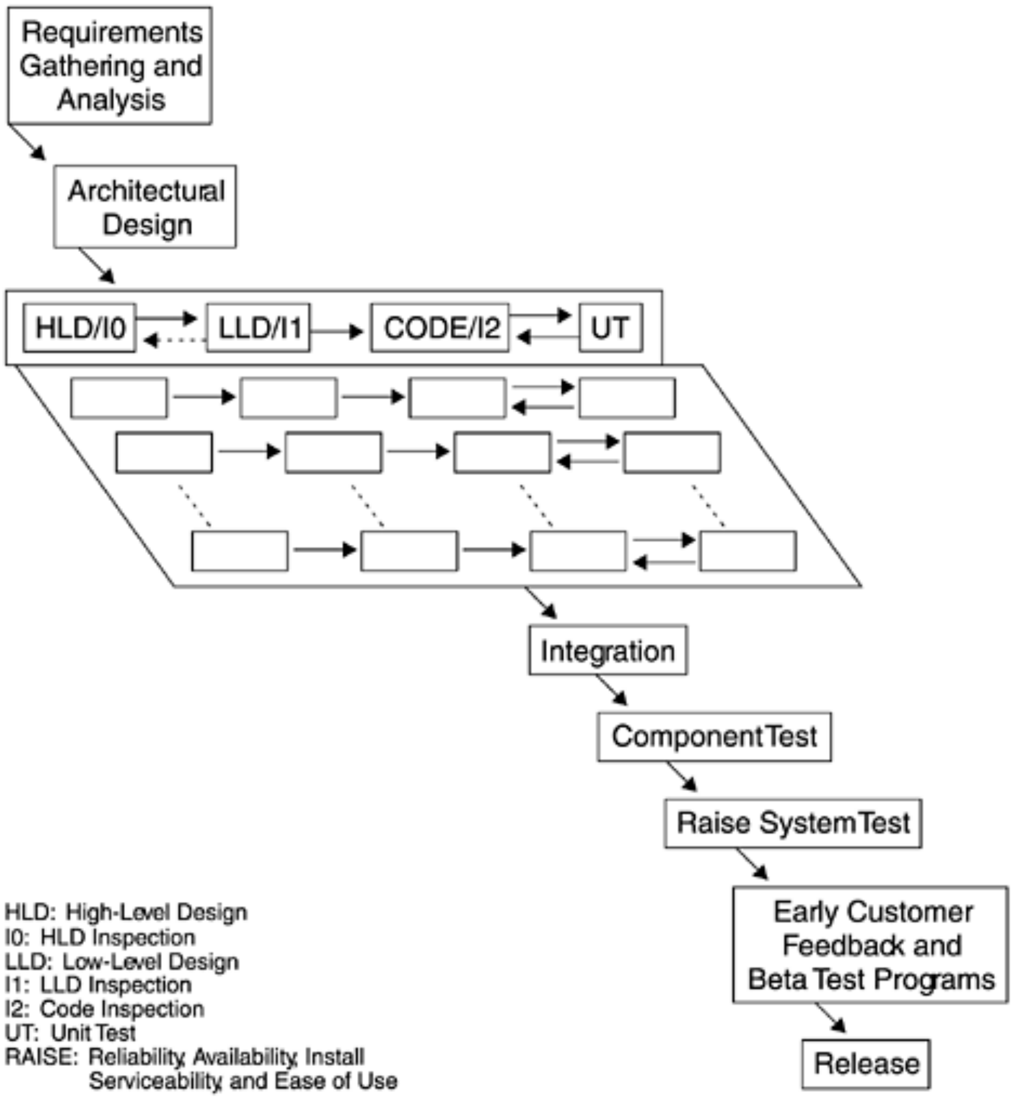


Figure 5.2, An Example of the Waterfall Process Model. (Kan, 2002)

The following sections describe the objectives of the various stages from high-level design to early customer programs.

5.1.1 <u>High-Level Design</u>

High-level design is the process of defining the externals and internals from the perspective of a component. Its objectives are as follows:

- Develop the external functions and interfaces, including:

    - External user interfaces
    - Application programming interfaces
    - System programming interfaces: intercomponent interfaces and data structures.

- Design the internal component structure, including intracomponent interfaces and data structures.
- Ensure all functional requirements are satisfied.
- Ensure the component fits into the system/product structure.
- Ensure the component design is complete.
- Ensure the external functions can be accomplished or possibility of doing the requirements

5.1.2 *Low-Level Design*

Low-level design is the process of transforming the high-level design into more detailed designs from the perspective of a part (modules, macros, includes, and so forth). Its objectives are as follows:

- Finalize the design of components and parts (modules, macros, includes) within a system or product.
- Complete the component test plans.
- Give feedback about high-level design and verify changes in it.

5.1.3 *Code Stage*

The coding portion of the process results in the transformation of a function's low-level design to completely coded parts. The objectives of this stage are as follows:

- Code parts (modules, macros, includes, messages, etc.).
- Code component test cases.
- Verify changes in high-level design and low-level design.

5.1.4 Unit Test

The unit test is the first test of an executable module. Its objectives are as follows:

- Verify the code against the component's
  - High-level design and
  - Low-level design.
- Execute all new and changed code to ensure
  - All branches are executed in all directions,
  - Logic is correct, and
  - Data paths are verified.
- Exercise all error messages, return codes, and response options.
- Give feedback about code, low-level design, and high-level design.

The level of unit test is for verification of limits, internal interfaces, and logic and data paths in a module, macro, or executable include. Unit testing is performed on nonintegrated code and may require scaffold code to construct the proper environment.

5.1.5 Component Test

Component tests evaluate the combined software parts that make up a component after they have been integrated into the system library. The objectives of this test are as follows:

- Test external user interfaces against the component's design documentation, user requirements.
- Test intercomponent interfaces against the component's design documentation.
- Test application program interfaces against the component's design documentation.
- Test function against the component's design documentation.
- Test intracomponent interfaces (module level) against the component's design documentation.
- Test error recovery and messages against the component's design documentation.
- Verify that component drivers are functionally complete and at the acceptable quality level.
- Test the shared paths (multitasking) and shared resources (files, locks, queues, etc.) against the component's design documentation.
- Test ported and unchanged functions against the component's design documentation.

5.1.6 System-Level Test

The system-level test phase comprises the following tests:

- System test
- System regression test
- System performance measurement test

- Usability tests

The system test follows the component tests and precedes system regression tests. The system performance test usually begins shortly after system testing starts and proceeds throughout the system-level test phase. Usability tests occur throughout the development process, it can be prototyping during design stages, formal usability testing during system test period.

- System test objectives
    - Ensure software products function correctly when executed concurrently and in stressful system environments.
    - Verify overall system stability when development activity has been completed for all products.
- System regression test objective
    - Verify that the final programming package is ready to be shipped to external customers.
    - Make sure original functions work correctly after functions were added to the system.
- System performance measurement test objectives
    - Validate the performance of the system.
    - Verify performance specifications.
    - Provide performance information to marketing.
    - Establish base performance measurements for future releases.
- Usability tests objective
    - Verify that the system contains the usability characteristics required for the intended user tasks and user environment.

5.1.7 Early Customer Programs

The early customer programs include testing of the following support structures to verify their readiness:

- Service structures
- Development fix support
- Electronic customer support
- Market support
- Ordering, manufacturing, and distribution

In addition to these objectives, a side benefit of having production systems installed in a customer's environment for the early customer programs is the opportunity to gather customers' feedback so developers can evaluate features and improve them for future releases. Collections of such data or user opinion include:

- Product feedback: functions offered, ease of use, and quality of online documentation
- Installability of hardware and software
- Reliability
- Performance which is measure throughput under the customer's typical load
- System connectivity
- Customer acceptance

As the preceding lists illustrate, the waterfall process model is a disciplined approach to software development. It is most appropriate for systems development characterized by a high degree of complexity and interdependency. Although expressed as a cascading waterfall, parallelism and some amount of iteration among process phases often exist in actual implementation. During this process, the focus should be on the intermediate deliverables like design document, interface rules, test plans, and test cases rather than on the sequence of activities for each development phase. In other words, it should be entity-based instead of step-by-step based. Otherwise the process could become too rigid to be efficient and effective.

The essence of waterfall model is that complex software systems can be built in a sequential, phase-wise manner where all of the requirements are gathered at the beginning, all of the design is completed next, and finally the master design is implemented into production quality software. This approach holds that complex systems can be built in a single pass, without going back and revisiting requirements or design ideas in light of changing business or technology conditions.

It equates software development to a production line conveyor belt. Requirements analysts compile the system specifications until they pass the finished requirements specification document to software designers who plan the software system and create diagrams documenting how the code should be written. The design diagrams are then passed to the developers who implement the code from the design.

Under the waterfall approach, traditional IT managers have made valiant efforts to craft and adhere to large-scale development plans. These plans are typically laid out in advance of development projects using Gantt or PERT charts to map detailed tasks and dependencies for each member of the development group months or years down the line. However, studies of past software projects show that only 9% to 16% are considered on-time and on-budget. (Standish Group International Inc, 1994) This article attempts to summarize current thinking among computer scientists on why waterfall fails in so many cases.

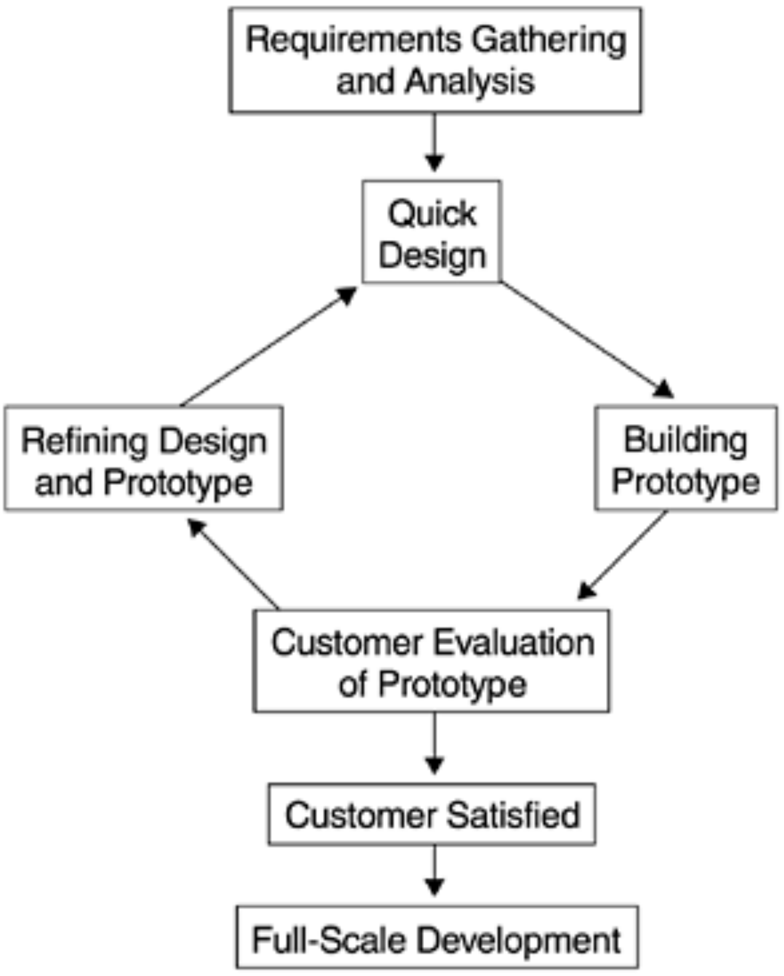
5.2 The Prototyping Approach

The first step in the waterfall model is the gathering and analysis of customers' requirements. When the requirements are defined, the design and development work begins. The model assumes that requirements are known, and that once requirements are defined, they will not change or any change will be insignificant. This may well be the case for system development in which the system's purpose and architecture are thoroughly investigated. However, if requirements change significantly between the times the system's specifications are finalized and when the product's development is complete, the waterfall may not be the best model to deal with the resulting problems. Sometimes the requirements are not even known. In the past, various software process models have been proposed to deal with customer feedback on the product to ensure that it satisfied the requirements. Each of these models provides some form of prototyping, of either a part or all of the system. Some of them build prototypes to be thrown away; others evolve the prototype over time, based on customer needs.

A prototype is a partial implementation of the product expressed either logically or physically with all external interfaces presented. The potential customers use the prototype and provide feedback to the development team before full-scale development begins. Seeing is believing, and that is really what prototyping intends to achieve. By using this approach, the customers and the development team can clarify requirements and their interpretation.

As Figure 5.3 shows, the prototyping approach usually involves the following steps:

- Gather and analyze requirements.
- Do a quick design.
- Build a prototype.
- Customers evaluate the prototype.
- Refine the design and prototype.
- If customers are not satisfied with the prototype, loop back to step 5.
- If customers are satisfied, begin full-scale product development.



Requirements Gathering and Analysis
Quick Design
Refining Design and Prototype
Building Prototype
Customer Evaluation of Prototype
Customer Satisfied
Full-Scale Development

Figure 5.3 – The prototyping approach. (Kan, 2002)

The critical factor for success of the prototyping approach is quick turnaround in designing and building the prototypes. Several technologies can be used to achieve such an objective. Reusable software parts could make the design and implementation of prototypes easier. Formal specification languages could facilitate the generation of executable code. Fourth-generation languages and technologies could be extremely useful for prototyping in the graphical user interface domain. These technologies are still emerging, however, and are used in varying degrees depending on the specific characteristics of the projects.

The prototyping approach is most applicable to small tasks or at the subsystem level. Prototyping a complete system is difficult. Another difficulty with this approach is knowing when to stop iterating. In practice, the method of time boxing is being used. This method involves setting arbitrary time limits for each activity in the iteration cycle and for the entire iteration and then assessing progress at these checkpoints.

5.2.1 Rapid Throwaway Prototyping

The rapid throwaway prototyping approach of software development, made popular by Gomaa and Scott (1981), is now used widely in the industry, especially in application development. It is usually used with high-risk items or with parts of the system that the development team does not understand thoroughly. In this approach, "quick and dirty" prototypes are built, verified with customers, and thrown away until a satisfactory prototype is reached, at which time full-scale development begins.

Some people are of the opinion that rapid prototyping is not effective because they believe it fails in replication of the real product or system. It could so happen that some important developmental steps could be omitted to get a quick and cheap working model.

Another disadvantage of rapid prototyping is one in which many problems are overlooked resulting in endless rectifications and revisions. Suitability of this approach is challenged and under question for large size applications.

5.2.2 Evolutionary Prototyping

In the evolutionary prototyping approach, a prototype is built based on some known requirements and understanding. The prototype is then refined and evolved instead of thrown away. Whereas throwaway prototypes are usually used with the aspects of the system that are poorly understood, evolutionary prototypes are likely to be used with aspects of the system that are well understood and thus build on the development team's strengths. These prototypes are also based on prioritized requirements, sometimes referred to as chunking in application development. For complex applications, it is not reasonable or economical to expect the prototypes to be developed and thrown away rapidly.

As of rapid prototyping, criticism on shortcomings of its evolutionary cousin is illustrated, in some manner can be called fairly illustrated. The two major disadvantages discussed in the literature are the rare possibility to set a release date and also usage of lethal code-and-fix development technique.

5.3 The Spiral Model

The spiral model of software development and enhancement, developed by Barry W. Boehm (1988), is based on experience with various refinements of the waterfall model as applied to large government software projects. Relying heavily on prototyping and risk management, it is much more flexible than the waterfall model. The spiral concept and the risk management focus have gained acceptance in software engineering and project management.

Figure 5.4 shows Boehm's spiral model. The underlying concept of the model is that each portion of the product and each level of elaboration involve the same sequence of steps, or in other words, cycles. Starting at the center of the spiral, one can see that each development phase involves one cycle of the spiral. The radial dimension in Figure 5.4 represents the cumulative cost incurred in accomplishing the steps. The angular dimension represents the progress made in completing each cycle of the spiral. As indicated by the quadrants in the figure, the first step of each cycle of the spiral is to identify the objectives of the portion of the product being elaborated, the alternative means of implementation of this portion of the product, and the constraints imposed on the application of the alternatives. The next step is to evaluate the alternatives relative to the objectives and constraints, to identify the associated risks, and to resolve them. Risk analysis and the risk-driven approach, therefore, are key characteristics of the spiral model, in contrast to the document-driven approach of the waterfall model.

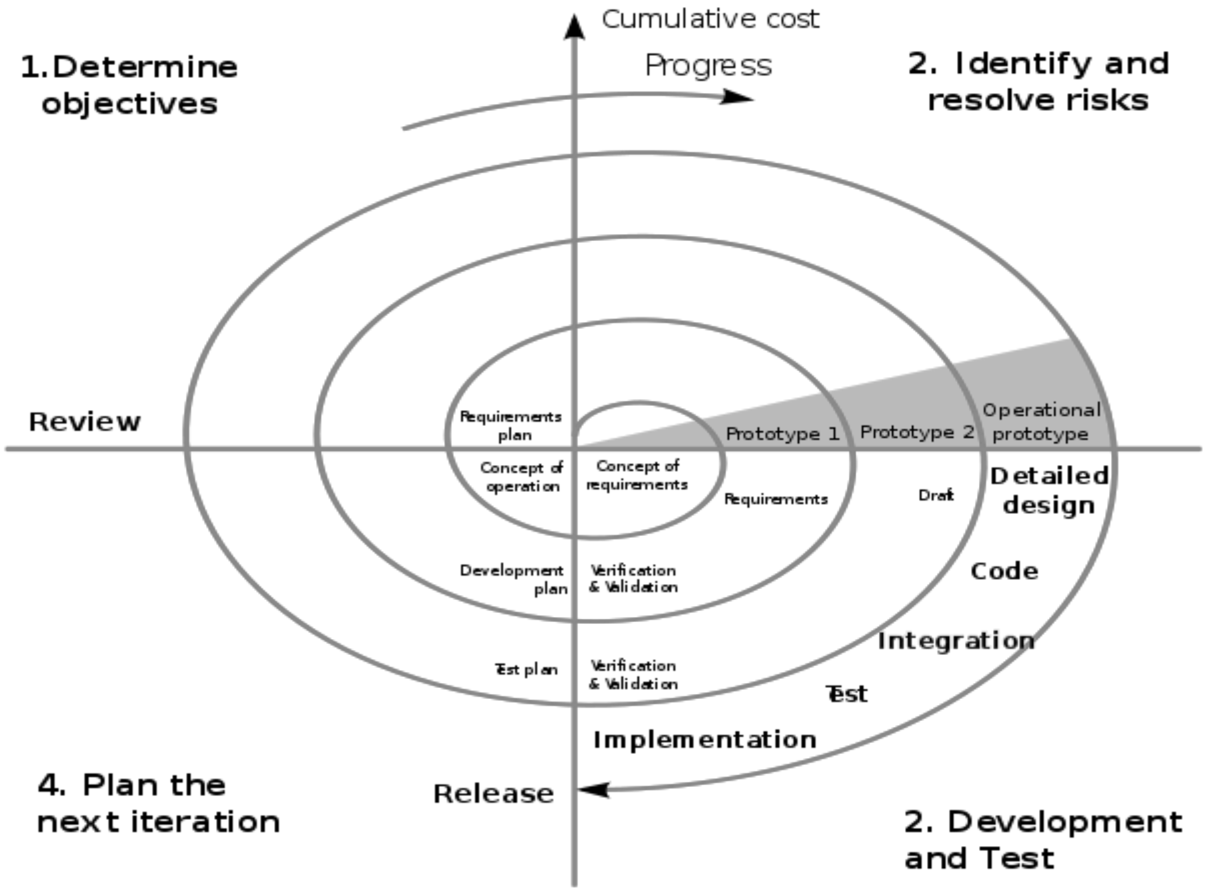


Figure 5.4. The Spiral Process Model. (Boehm, 1988)

In this risk-driven approach, prototyping is an important tool. Usually prototyping is applied to the elements of the system or the alternatives that present the higher risks. Unsatisfactory prototypes can be thrown away; when an operational prototype is in place, implementation can begin. In addition to prototyping, the spiral model uses simulations, models, and benchmarks in order to reach the best alternative. As indicated in the illustration, an important feature of the spiral model, as with other models, is that each cycle ends with a review involving the key members or organizations concerned with the product.

For software projects with incremental development or with components to be developed by separate organizations or individuals, a series of spiral cycles can be used, one for each increment or component. A third dimension could be added to Figure 3.4 to represent the model better.

Boehm provides a candid discussion of the advantages and disadvantages of the spiral model. Its advantages are as follows:

- Its range of options accommodates the good features of existing software process models, whereas its risk-driven approach avoids many of their difficulties. This is the primary advantage. Boehm also discusses the primary conditions under which this model becomes equivalent to other process models such as the waterfall model and the evolutionary prototype model.
- It focuses early attention on options involving the reuse of existing software. These options are encouraged because early identification and evaluation of alternatives is a key step in each spiral cycle. This model accommodates preparation for life-cycle evolution, growth, and changes of the software product.
- It provides a mechanism for incorporating software quality objectives into software product development.
- It focuses on eliminating errors and unattractive alternatives early.
- It does not involve separate approaches for software development and software enhancement.
- It provides a viable framework for integrating hardware-software system development. The risk-driven approach can be applied to both hardware and software.

On the other hand, difficulties with the spiral model include the following:

- Matching to contract software: Contract software relies heavily on control, checkpoint, and intermediate deliverables for which the waterfall model is good. The spiral model has a great deal of flexibility and freedom and is, therefore, more suitable for internal software development. The challenge is how to achieve the flexibility and freedom prescribed by the spiral model without losing accountability and control for contract software.

- Relying on risk management expertise: The risk-driven approach is the backbone of the model. The risk-driven specification addresses high-risk elements in great detail and leaves low-risk elements to be elaborated in later stages. However, an inexperienced team may also produce a specification just the opposite: a great deal of detail for the well-understood, low-risk elements and little elaboration of the poorly understood, high-risk elements. In such a case, the project may fail and the failure may be discovered only after major resources have been invested. Another concern is that a risk-driven specification is people dependent. In the case where a design produced by an expert is to be implemented by non-experts, the expert must furnish additional documentation.

- Need for further elaboration of spiral steps: The spiral model describes a flexible and dynamic process model that can be used to its fullest advantage by experienced developers. For non-experts and especially for large-scale projects, however, the steps in the spiral must be elaborated and more specifically defined so that consistency, tracking, and control can be achieved. Such elaboration and control are especially important in the area of risk analysis and risk management.

5.4 The Object-Oriented Development Process

The object-oriented approach to design and programming, which was introduced in the 1980s, represents a major paradigm shift in software development. Different from traditional programming, which separates data and control, object-oriented programming is based on objects, each of which is a set of defined data and a set of operations that can be performed on that data. Like the paradigm of structural design and functional decomposition, the object-oriented approach has become a major cornerstone of software engineering. In the early days of object-oriented technology deployment, which was from late 1980s to mid 1990s, much of the object oriented literature concerned analysis and design methods, therefore there was little information about its development processes. In recent years the object-oriented technology has been widely accepted and object-oriented development is now so pervasive that there is no longer a question of its viability.

Branson and Herness (1992) proposed an object oriented development process for large-scale projects that centers on an eight-step methodology supported by a mechanism for tracking, a series of inspections, a set of technologies, and rules for prototyping and testing.

The eight-step process is divided into three logical phases:

- The analysis phase focuses on obtaining and representing customers' requirements in a concise manner, to visualize an essential system that represents the users' requirements regardless of which implementation platform (hardware or software environment) is developed.
- The design phase involves modifying the essential system so that it can be implemented on a given set of hardware and software. Essential classes and incarnation classes are combined and refined into the evolving class hierarchy. The objectives of class synthesis are to optimize reuse and to create reusable classes.
- The implementation phase takes the defined classes to completion.

The eight steps of the process are summarized as follows:

1. Model the essential system: The essential system describes those aspects of the system required for it to achieve its purpose, regardless of the target hardware and software environment. It is composed of essential activities and essential data. This step has five sub steps:
    - Create the user view.
    - Model essential activities.
    - Define solution data.
    - Refine the essential model.
    - Construct a detailed analysis.

This step focuses on the user requirements. Requirements are analyzed, dissected, refined, combined, and organized into an essential logical model of the system. This model is based on the perfect technology premise.

2. Derive candidate-essential classes: This step uses a technique known as "carving" to identify candidate-essential classes and methods from the essential model of the whole system. A complete set of data-flow diagrams, along with supporting process specifications and data dictionary entries, is the basis for class and method selection. Candidate classes and methods are found in external entities, data stores, input flows, and process specifications.

3. Constrain the essential model: The essential model is modified to work within the constraints of the target implementation environment. Essential activities and essential data are allocated to the various processors and containers. Activities are added to the system as needed, based on limitations in the target implementation environment. The essential model, when augmented with the activities needed to support the target environment, is referred to as the incarnation model.

4. Derive additional classes: Additional candidate classes and methods specific to the implementation environment are selected based on the activities added while constraining the essential model. These classes supply interfaces to the essential classes at a consistent level.

5. Synthesize classes: The candidate-essential classes and the candidate-additional classes are refined and organized into a hierarchy. Common attributes and operations are extracted to produce superclasses and subclasses. Final classes are selected to maximize reuse through inheritance and importation.

6. Define interfaces: The interfaces, object-type declarations, and class definitions are written based on the documented synthesized classes.

7. Complete the design: The design of the implementation module is completed. The implementation module comprises several methods, each of which provides a single cohesive function. Logic, system interaction, and method invocations to other classes are used to accomplish the complete design for

each method in a class. Referential integrity constraints specified in the essential model are now reflected in the class design.

8. Implement the solution: The implementation of the classes is coded and unit tested.

The analysis phase of the process consists of steps 1 and 2, the design phase consists of steps 3 through 6, and the implementation phase consists of steps 7 and 8. Several iterations are expected during analysis and design. Prototyping may also be used to validate the essential model and to assist in selecting the appropriate incarnation. Furthermore, the process calls for several reviews and checkpoints to enhance the control of the project. The reviews include the following:

- Requirements review after the second substep of step 1 (model essential system)
- External structure and design review after the fourth substep (refined model) of step 1
- Class analysis verification review after step 5
- Class externals review after step 6
- Code inspection after step 8 code is complete

In addition to methodology, requirements, design, analysis, implementation, prototyping, and verification, Branson and Herness assert that the object-oriented development process architecture must also address elements such as reuse, CASE tools, integration, build and test, and project management. The Branson and Herness process model, based on their object-oriented experience at IBM Rochester, represents one attempt to deploy the object-oriented technology in large organizations. It is certain that many more variations will emerge before a commonly recognized object-oriented process model is reached.

Finally, the element of reuse merits more discussion from the process perspective. Design and code reuse gives object-oriented development significant advantages in quality and productivity. However, reuse is not automatically achieved simply by using object-oriented development. Object-oriented development provides a large potential source of reusable components, which must be generalized to become usable in new development environments. In terms of development life cycle, generalization for reuse is typically considered an add-on at the end of the project. However, generalization activities take time and resources. Therefore, developing with reuse is what every object-oriented project is aiming for, but developing for reuse is difficult to accomplish. This reuse paradox explains the reality that there are no significant amounts of business-level reusable code despite the promises object-oriented technology offers, although there are many general-purpose reusable libraries. Therefore, organizations that intend to leverage the reuse advantage of object-oriented development must deal with this issue in their development process.

Henderson-Sellers and Pant (1998) propose a two-library model for the generalization activities for reusable parts. The model addresses the problem of costing and is quite promising. The first step is to put "on hold" project-specific classes from the current project by placing them in a library of potentially reusable components. Thus the only cost to the current project is the identification of these classes. The second library, the library of generalized components, is the high-quality company resource. At the beginning of each new project, an early phase in the development process is an assessment of classes that reside in the these two libraries in terms of their reuse value for the project. If of value, additional spending on generalization is made and potential parts in library of potentially reusable components can undergo the generalization process and quality checks and be placed in library of generalized components. Because the reusable parts are to benefit the new project, it is reasonable to allocate the cost of generalization to the customer.

As the preceding discussion illustrates, it may take significant research, experience, and ingenuity to piece together the key elements of an object-oriented development process and for it to mature. In the late 1990s, the Unified Software Development Process, which was developed by Jacobson, Booch, and Rumbaugh (1998) and is owned by the Rational Software Corporation, was published. The process relies on the Unified Modeling Language (UML) for its visual modeling standard. It is usecase driven, architecture-centric, iterative, and incremental. Usecases are the key components that drive this process model. A usecase can be defined as a piece of functionality that gives a user a result of a value. All the usecases developed can be combined into a usecase model, which describes the complete functionality of the system. The usecase model is analogous to the functional specification in a traditional software development process model. Usecases are developed with the users and are modeled in UML. These represent the requirements for the software and are used throughout the process model. The Unified Process is also described as architecture-centric. This architecture is a view of the whole design with important characteristics made visible by leaving details out. It works hand in hand with the usecases. Subsystems, classes, and components are expressed in the architecture and are also modeled in UML. Last but not least, the Unified Process is iterative and incremental. Iterations represent steps in a workflow, and increments show growth in functionality of the product. The core workflows for iterative development are:

- Requirements
- Analysis
- Design
- Implementation
- Test

The Unified Process consists of cycles. Each cycle results in a new release of the system, and each release is a deliverable product. Each cycle has four phases: inception, elaboration, construction, and transition. A number of iterations occur in each phase, and the five core workflows take place over the four phases.

During inception, a good idea for a software product is developed and the project is started. A simplified usecase model is created and project risks are prioritized. Next, during the elaboration phase, product usecases are specified in detail and the system architecture is designed. The project manager begins planning for resources and estimating activities. All views of the system are delivered, including the usecase model, the design model, and the implementation model. These models are developed using UML and held under configuration management. Once this phase is complete, the construction phase begins. From here the architecture design grows into a full system. Code is developed and the software is tested. Then the software is assessed to determine if the product meets the users' needs so that some customers can take early delivery. Finally, the transition phase begins with beta testing. In this phase, defects are tracked and fixed and the software is transitioned to a maintenance team.

5.5 The Iterative Development Process Model

The iterative enhancement approach, or the iterative development process, was defined to begin with a subset of the requirements and develop a subset of the product that satisfies the essential needs of the users, provides a vehicle for analysis and training for the customers, and provides a learning experience for the developer. Based on the analysis of each intermediate product, the design and the requirements are modified over a series of iterations to provide a system to the users that meets evolving customer needs with improved design based on feedback and learning.

The iterative development process model combines prototyping with the strength of the classical waterfall model. Other methods such as domain analysis and risk analysis can also be incorporated into the iterative development process model. The model has much in common with the spiral model, especially with regard to prototyping and risk management. Indeed, the spiral model can be regarded as a specific iterative development process model, while the term iterative development process is a general rubric under which various forms of the model can exist. The model also provides a framework for many modern systems and software engineering methods and techniques such as reuse, object-oriented development, and rapid prototyping.

Figure 5.5 shows an example of the iterative development process model used by IBM. With the purpose of "building a system by evolving an architectural prototype through a series of executable versions, with each successive iteration incorporating experience and more system functionality," the example implementation contains eight major steps:

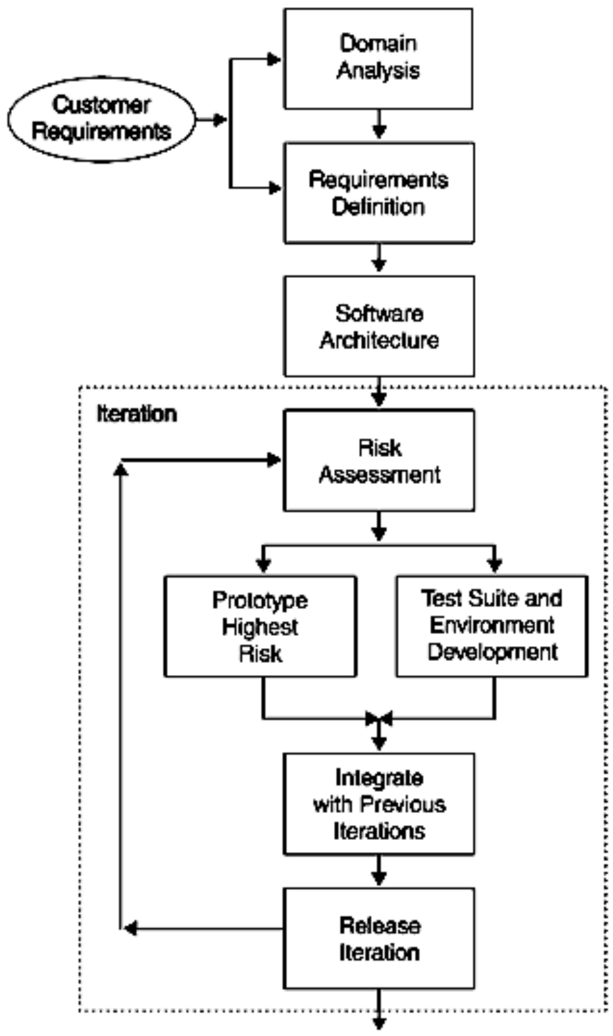


Figure 5.5 An Example of the Iterative Development Process Model. (Luckey, 1992)

1. Domain analysis
2. Requirements definition
3. Software architecture
4. Risk analysis
5. Prototype
6. Test suite and environment development
7. Integration with previous iterations
8. Release of iteration

As illustrated in the figure, the iteration process involves the last five steps; domain analysis, requirements definition, and software architecture are pre-iteration steps, which are similar to those in the waterfall model. During the five iteration steps, the following activities occur:

- Analyze or review the system requirements.
- Design or revise the solution that best satisfies the requirements.
- Identify the highest risks for the project and prioritize them. Mitigate the highest priority risk via prototyping, leaving lower risks for subsequent iterations.
- Define and schedule or revise the next few iterations.
- Develop the iteration test suite and supporting test environment.
- Implement the portion of the design that is minimally required to satisfy the current iteration.
- Integrate the software in test environments and perform regression testing.
- Update documents for release with the iteration.
- Release the iteration.

Test suite development along with design and development is extremely important for the verification of the function and quality of each iteration. Yet in practice this activity is not always emphasized appropriately.

The development of IBM's OS/2 2.0 operating system is a combination of the iterative development process and the small team approach. Different from the last example to some extent, the OS/2 2.0 iterative development process involved large-scale early customer feedback instead of just prototyping. The iterative part of the process involved the loop of subsystem design to subsystem code and test and to system integration to customer feedback and back to subsystem design. Specifically, the waterfall process involved the steps of market requirements, design, code and test, and system certification. The iterative process went from initial market requirements to the iterative loop, then to system certification. Within the one-year development cycle, there were five iterations, each with increased functionality, before completion of the system. For each iteration, the customer feedback involved a beta test of the available functions, a formal customer satisfaction survey, and feedback from various vehicles such as electronic messages on Prodigy, IBM internal e-mail conferences, customer visits, technical seminars, and internal and public bulletin boards. Feedback from various channels was also statistically verified and validated by the formal customer satisfaction surveys. More than 30,000 customers and 100,000 users were involved in the iteration feedback process. Supporting the iterative process was the small team approach in which each team assumed full responsibility for a particular function of the system. Each team owned its project, functionality, quality, and customer satisfaction, and was held completely responsible. Cross-functional system teams also provided support and services to make the subsystem teams successful and to help resolve cross-subsystem concerns.

The OS/2 2.0 and later versions are still used in the professional computing of different businesses and also embedded systems.

It was widely used in Brazilian banks. Banco do Brasil had a peak 10,000 machines running OS/2 Warp in the 1990s. OS/2 was used in automated teller machines until 2006. The workstations and automated teller machines have been migrated to Linux.

OS/2 is still used in the banking industry. Suncorp bank in Australia still ran its ATM network on OS/2 as late as 2002. ATMs in Perisher Blue used OS/2 as late as 2009, and even the turn of the decade.

OS/2 is still used to control the SkyTrain automated light rail system in Vancouver, Canada.

It is also still used by the Stop & Shop supermarket chain and has been installed in new stores as recently as March 2010.

5.6 The Extreme Programming

One very controversial object oriented process that has gained recognition and generated vigorous debates among software engineers is Extreme Programming proposed by Kent Beck (2000). This lightweight, iterative and incremental process has four cornerstone values: communication, simplicity, feedback, and courage. With this foundation, extreme programming advocates the following practices:

- The Planning Game: Development teams estimate time, risk, and story order. The customer defines scope, release dates, and priority.
- System metaphor: A metaphor describes how the system works.
- Simple design: Designs are minimal, just enough to pass the tests that bound the scope.
- Pair programming: All design and coding is done by two people at one workstation. This spreads knowledge better and uses constant peer reviews.
- Unit testing and acceptance testing: Unit tests are written before code to give a clear intent of the code and provide a complete library of tests.
- Refactoring: Code is refactored before and after implementing a feature to help keep the code clean.
- Collective code ownership: By switching teams and seeing all pieces of the code, all developers are able to fix broken pieces.
- Continuous integration: The more code is integrated, the more likely it is to keep running without big hang-ups.
- On-site customer: An onsite customer is considered part of the team and is responsible for domain expertise and acceptance testing.
- 40-hour week: Stipulating a 40-hour week ensures that developers are always alert.
- Small releases: Releases are small but contain useful functionality.

- Coding standard: Coding standards are defined by the team and are adhered to.

According to Beck, because these practices balance and reinforce one another, implementing all of them in concert is what makes Extreme Programming extreme. With these practices, a software engineering team can "embrace changes." Unlike other evolutionary process models, it discourages preliminary requirements gathering, extensive analysis, and design modeling. Instead, it intentionally limits planning for future flexibility, which emphasizes fewer classes and reduced documentation. It appears that the Extreme Programming philosophy and practices may be more applicable to small projects. For large and complex software development, some of its principles become harder to implement and may even run against traditional wisdom that is built upon successful projects. Beck stipulates that to date Extreme Programming efforts have worked best with teams of ten or fewer members.

5.7 The Scrum

Scrum is an iterative, incremental methodology for project management. Although Scrum was intended for management of software development projects, it can be used to run software maintenance teams, or as a general project management approach.

Hirotaka Takeuchi and Ikujiro Nonaka (1986) described a new holistic approach that would increase speed and flexibility in commercial new product development. They compared this new holistic approach, in which the phases strongly overlap and the whole process is performed by one cross-functional team across the different phases, to rugby, where the whole team "tries to go the distance as a unit, passing the ball back and forth". The case studies came from the automotive, photo machine, computer, and printer industries.

DeGrace and Stahl (1991), in "Wicked Problems, Righteous Solutions", referred to this approach as Scrum, a rugby term mentioned in the article by Takeuchi and

Nonaka. In the early 1990s, Ken Schwaber (1995) used an approach that led to Scrum at his company, Advanced Development Methods. At the same time, Jeff Sutherland, John Scumniotales, and Jeff McKenna (1995) developed a similar approach at Easel Corporation and were the first to call it Scrum.

Later, Sutherland and Schwaber (1995) jointly presented a paper, describing Scrum, its first public appearance. Schwaber and Sutherland collaborated during the following years to merge the above writings, their experiences, and industry best practices into what is now known as Scrum.

Although the word is not an acronym, some companies implementing the process have been known to spell it with capital letters as SCRUM. This may be due to one of Ken Schwaber's (2004) early papers, which capitalized Scrum in the title.

Scrum is a process skeleton that contains sets of practices and predefined roles. The main roles in Scrum are:

- The Scrum Master, who maintains the processes, typically the project management.
- The Product Owner, who represents the stakeholders and the business
- The Team, a cross-functional group of about 7 people who do the actual analysis, design, implementation, testing, etc.

As depicted in figure 5.6, during each sprint, typically a two to four week period, with the length being decided by the team, the team creates a potentially shippable product increment. The set of features that go into a sprint come from the product backlog, which is a prioritized set of high level requirements of work to be done. Which backlog items go into the sprint is determined during the sprint planning meeting. During this meeting, the Product Owner informs the team of the items in the product backlog that he or she wants completed. The team then determines how much of this they can commit to complete during the next sprint. During a sprint, no one is allowed to change the sprint backlog, which means that the requirements are frozen for that sprint. Development is time-boxed such that the sprint must end on time; if requirements are not completed for any reason they are left out and returned to the product backlog. After a sprint is completed, the team demonstrates how to use the software.

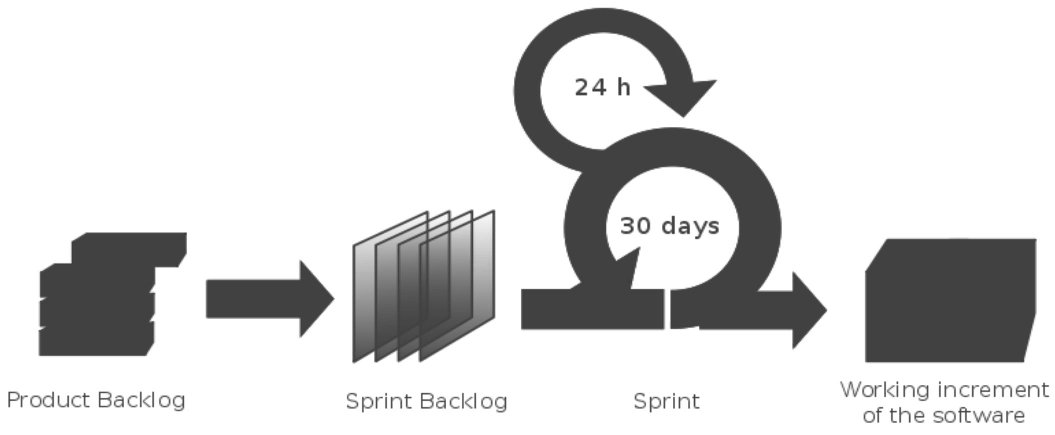


Figure 5.6 – The Scrum Model. (en.wikipedia.org, 2010)

Scrum enables the creation of self-organizing teams by encouraging co-location of all team members, and verbal communication across all team members and disciplines that are involved in the project.

A key principle of Scrum is its recognition that during a project the customers can change their minds about what they want and need, often called requirements churn, and that unpredicted challenges cannot be easily addressed in a traditional predictive or planned manner. As such, Scrum adopts an empirical approach, accepting that the problem cannot be fully understood or defined, focusing instead on maximizing the team’s ability to deliver quickly and respond to emerging requirements.

Scrum can be implemented through a wide range of tools. Many companies use universal software tools, such as spreadsheets to build and maintain artifacts such as the sprint backlog. There are also open-source and proprietary software packages dedicated to management of products under the Scrum process. Other organizations implement Scrum without the use of any software tools, and maintain their artifacts in hard-copy forms such as paper, whiteboards, and sticky notes.

6 THE REVOLUTIONARY APPROACH

All these development process models are precious and useful. As time passes, models and methods evolve. It was implicitly derived from the few models reviewed that they are founded upon the lessons and experiments learned from using the predecessor ones. The evolutions happened to improve the overall efficiency of the models to improve the products in as many aspects as possible. They all intend to reduce the time, cost, risk and increase the quality and reliability, in other words, reduce the bad and increase the good.

Among these models, Scrum and Extreme Programming sound reasonably efficient and they can be named as the most efficient models, but they still have some shortcomings that applicability and their success is not implementable for any possible software development project in any environment.

There are many critics on pair programming issue in extreme programming.

“The only constraint that extreme programming puts on you is that any production code has to be [sic] written by a pair. Your preferences and comfort do not supersede the delivery of quality to the project, or your participation [sic] in the team.” As Robert C. Martin (2001) says.

“Having a number of years of programming experience, I place a pretty high value on peace, quiet, and space to think in. According to a study from IBM’s SantaTeresaLaboratory, putting programmers in private offices with doors that closed instead of cubicles resulted in a huge boost to productivity. So the idea of all the programmers in a big, noisy room seems like it would be a huge detriment to productivity. It would figure to drive many people nuts.” (Stephens and Rosenberg, 2003)

“The affordability of pair programming is a key issue. If it is much more expensive, managers simply will not permit it. Skeptics assume that incorporating pair programming will double code development expenses and critical manpower needs.” As Alistair Cockburn and Laurie Williams (2003) state in their paper, “The Costs and Benefits of Pair Programming.”

Scrum model is not safe from critics' complaints. It has been brought to challenge in many issues. It can be one of the leading causes of scope creep. If there is no definite end date, therefore the project management stakeholders may be tempted to keep demanding new functionality is delivered.

In other way, if a task is not well defined, estimating project costs and time will not be accurate. In such a case, the task can be spread over several sprints.

Many critics believe scrum is only suitable for teams of experienced members. Lack of enough experience causes the project to not be completed on time.

Scrum and Extreme Programming and some other iterative methodologies like Crystal Clear, Feature Driven Development and Dynamic Systems Development method are now typically referred as Agile Methodologies since 2001 that Agile Manifesto published.

However these critics are in some manners fair, but in practice success and benefits from using these two models are significantly ahead the failures and shortcomings. In the last chapter research and surveys reveal this point out based on literature using large scale of organizations' responses.

6.1 The Manifesto for Agile Software Development

In 2001, 17 software developers gathered in Utah, The United States, to discuss lightweight development methodologies. They published the Manifesto for Agile Software Development to define the approach now known as agile software development. As stated in the Agile Manifesto it contains 12 principles, which are as follow:

- 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

Many development methodologies are considered as agile development methods, including previously discussed, Scrum and Extreme Programming. Agile software development methodologies are widely accepted these days. Traditional software development methods are not efficient enough to convene with the rapid change in requirements and short iterations that are required for efficient product delivery.

Agile software practices achieve agility by promoting self-organizing teams, customer collaboration, higher quality, less documentation and reduced time to market. Traditional software development practices relied heavily on documentation. Project managers were of the view that by getting as much specification as possible early in the planning and design phase would save a lot of time, cost and resources for the software project as changes later on in the development process would result in setbacks and increase in cost, time and scope. The traditional practices did not allow changes late in the implementation and verification phases, but even then, many software projects failed to meet their objectives due to the quantum of required specifications and bulk of documentation demand. Agile software practices solve these problems by encouraging changes integration and close customer interaction with software developing teams. Agile supports iterations that can integrate any change in the user requirements during the implementation phase.

Agile development methodologies are transforming the way development teams work. Agile development enables organizations to deliver products to market faster and to respond more rapidly to changing market priorities by enabling more effective processes. In fact, Agile helps turn software development organizations into software delivery organizations.

6.2 Impact of Agile Development Methodologies on Quality and Productivity

Many studies and literatures on agile methodologies identify the parameters, which impact on productivity and quality of software projects. These parameters are identified based on surveys, research and interviews with experienced professionals practicing agile. A few of the parameters from the current literature are discussed in this text.

6.2.1 Knowledge Sharing

Knowledge sharing is an activity in software projects by which information and knowledge is shared amongst members. It is a constituent part of agile development environment, normally done with the help of a knowledge management system.

Discussion with some of the senior professionals in the field of software engineering has revealed that there are regular knowledge sharing session in organizations practicing agile and senior and experienced resources are provided with a chance to share their knowledge and experience. Knowledge sharing amongst team members helps in solving tough problems, rather spending time individually. Individuals who are new to the team get an opportunity to learn from knowledge sharing sessions.

Survey results have shown that almost all organizations practice knowledge sharing as an essential constituent of agile methodologies.

6.2.2 Active Participation of Stakeholders

Agile methodologies emphasize on a strong customer and developer relationship. With the expanding Internet technology and development of integrated solutions, the business demands faster delivery of projects to its clients and it can be best achieved with continuous involvement of the stakeholders who have a stake in the project completion. In order to facilitate dynamic changes in requirements even late in the implementation phase requires active participation from customers.

Most of the organizations practicing agile methodologies have their business analysts associated with clients. The business analysts keep an active contact with clients, ensuring faster delivery of releases to the end users and improving productivity. According to the Agile Manifesto, which was discussed earlier, the most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

6.2.3 Self-Organized Teams

Teams are organized to handle complexity and pressures of deadlines during project development, and to bring the project to completion. They take decisions on their own and adapt accordingly with changing situations. Self-organized teams do a much better job of utilizing the talents of the team because more minds are involved in any activity. Self-organizing teams are better than command and control teams because they it provides are chance for personal development, having responsibility placed on the shoulders of individuals working in a team solely for the successful completion of the project.

6.2.4 Reduced Documentation

Agile development methodologies emphasize more towards completion and delivery of the project to the customer in short time span than emphasizing on documentation. The main objective of the development teams is to deliver a working release when needed complying with quality standards. New releases are produced at frequent intervals, in some approaches even hourly or daily. The developers are urged to keep the code simple, straightforward, and technically as advanced as possible, thus lessening the documentation burden to an appropriate level.

6.2.5 Response to Change

Agile advocates that changes are welcome even late in the implementation phase, and this is one of the major reasons of increase in productivity. Changes are managed, analyzed and implemented by the development teams even on short notices. It is only possible if the organization is using agile development methodology and it is mature enough to welcome dynamic changes.

Researches show that more than 90 percents organizations practicing agile welcome dynamic changes in requirements even late in the implementation phase for their projects. (A. Ahmed, et al., 2010)

6.2.6 Trainings

To face the challenging environment offered by fast changing technological scenarios, it is necessary for an organization to have its people up to date with the latest tools and technologies. Especially for organization practicing agile method of development training of resources is essential for keeping the development process up to the mark, satisfying customers and business expectations. Training of professionals always has a positive and significant impact on productivity.

6.2.7 Refactoring

Refactoring is a programming technique used to improve quality, including source code quality. When done by changing the code in smaller chunks in a disciplined manner, improves the quality of the code without affecting the external functionality. Code refactoring and database refactoring is done in order to attain make the source code maintainable and database design more flexible.

6.3 An Individual Survey Respond: Autodesk

Autodesk as a leading software company in computer-aided design software production response is worthy. The survey was on comparison of their development projects before and after employing agile methodologies. The following is the result of using agile in Autodesk Ges.M.B.H. located in Wels, Austria.

Benefit derived from going agile is described as huge for being able to engage development much earlier in the cycle using the Agile Process in comparison with the former waterfall approach, with less accumulated technical and design debt.

Overall quality is improved prior to the system test phase as a result of addressing critical bugs during the sprint instead of allowing them to accumulate to the end game.

Provided earlier and longer focus of development team on core business projects as opposed to other side projects. The side projects like extra bug fixing and development of anticipated long lead work are not of value anymore and the focus to what the business objectives are of more value.

Amplified communication and collaboration across teams with different functions like product design, development and quality assurance, which were not in contact of such close level and frequency before, leads to improvement in quality and productivity.

The process changes unlock focusing on higher priority customer requests. This focus also provided less surprise at the end as iterations with customers inject feedback throughout the cycle.

6.4 Agile Development in action: Results from practitioners

This chapter is dedicated to agile development within today's software organizations, why Agile is being implemented and the value realized by doing so. The results are based on the literature published upon working with thousands of agile teams around the world.

6.4.1 Members Size Growth

Surveys show that organizations practicing agile development methodologies are now larger and more distributed than ever before: 39 percents of organizations surveyed deploying agile development practices have 101 or more total employees responsible for the delivery of software with 27 percents even having over 250 employees.

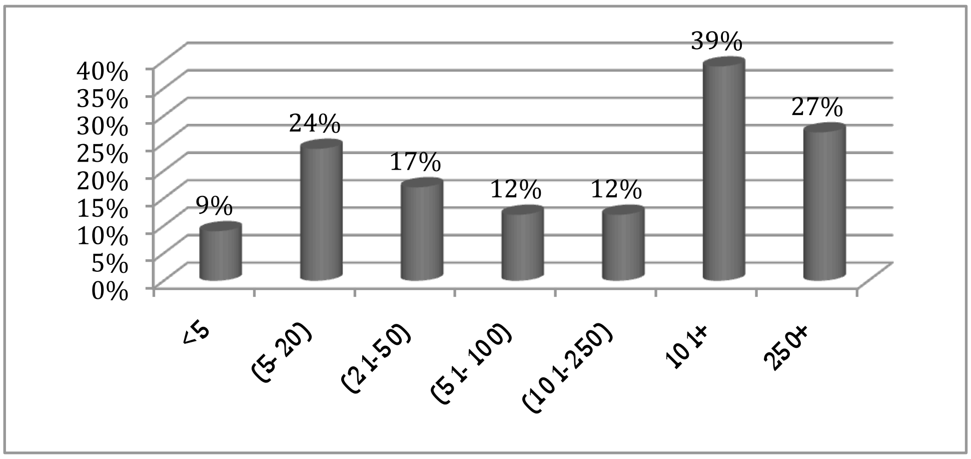


Figure 6.1 - Size of Software Organization Adopting Agile Process.

6.4.2 Purposes of Employing Agile and the Results in them

Business environment is simply faster paced and more distributed and fluid than ever before. This competitive landscape requires software development organizations to be more flexible and adaptive to create and maintain a competitive advantage. Surveys have found that the four most critical needs for software delivery organizations are:

- Need to manage rapidly changing priorities.
- Need to accelerate time-to-market.
- Need to increase productivity.
- Need to improve quality.
- Need to decrease costs or to prevent its growth
- Need to improve business satisfaction

These factors are the foundations for companies looking to implement agile practices, as depicted in figure 6.2.

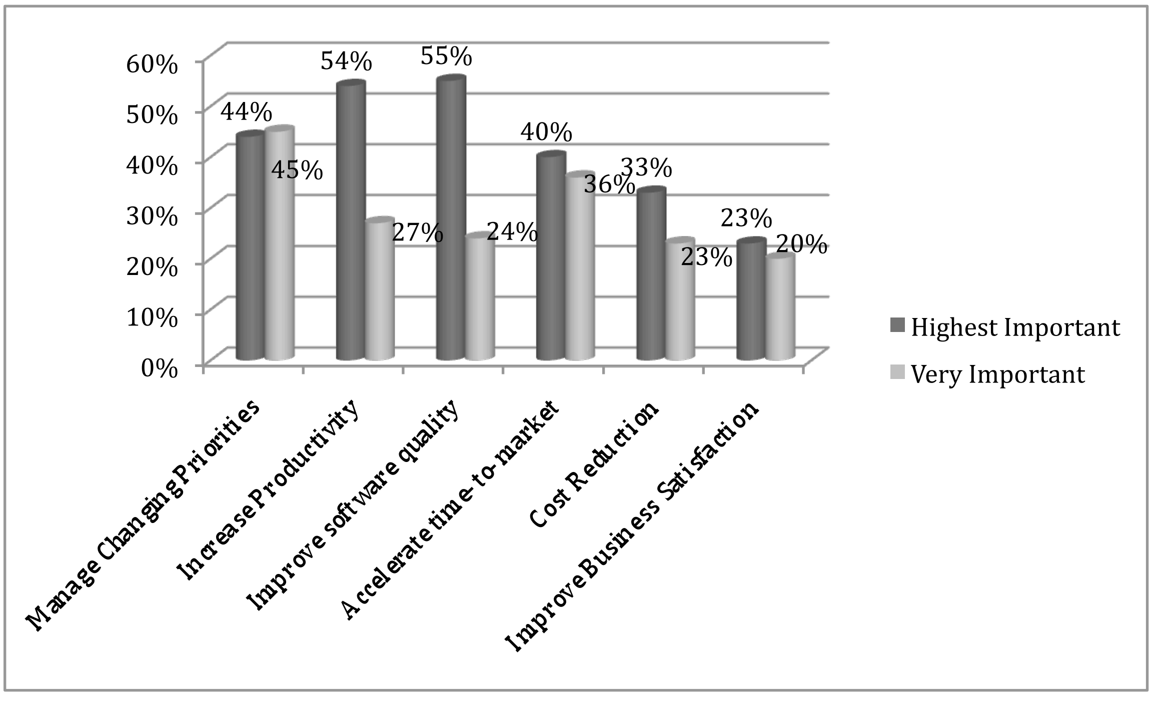


Figure 6.2 Reasons for going Agile.

While managing changing priorities was the most important reason cited for implementing Agile and was recognized as the most improved, all four critical needs were identified as having been either improved or significantly improved in majority of cases.

6.4.2.1 Ability To Manage Rapidly Changing Priorities

Agile development inherently welcomes change throughout the development lifecycle. Agile teams plan and prioritize requirements at each iteration. This iterative planning process gives them the ability to change, adapt or remove requirements throughout the project timeline as priorities change. The majority of traditional processes restrict flexibility to a serial planning phase prior to coding, testing and deployment.

Iterations within an agile process contain each element of a traditional process, from planning and coding to testing and deployment. The team's focus remains on prioritized work items and they limit re-prioritizing and planning of additional functionality until the next iteration.

By better management of changing priorities, agile methods allow software teams and their projects to closely align with business need and value. As shown in Figure-6.3, 91 percents of respondents indicated that the implementation of agile development either improved or significantly improved their ability to manage changing priorities.

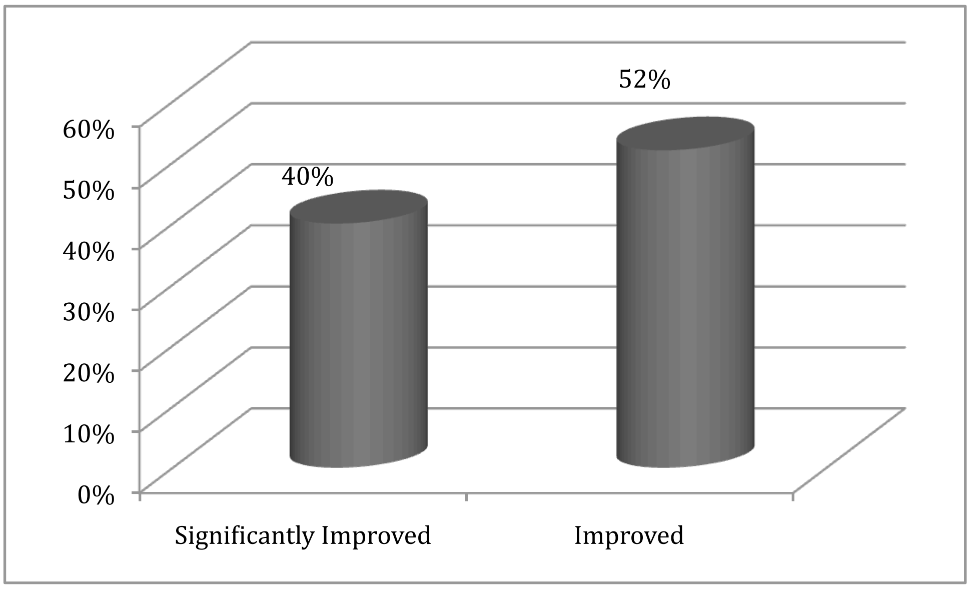


Figure 6.3 – Managed Changing Priorities.

4.4.2.2 *<u>Accelerate Time-To-Market</u>*

The most notable factor in decreasing time-to-market is the ability to deliver working software prior to the end of a project. Traditional development's big bang method is characterized by development running through silos of individual work groups: planning leads to coding, coding leads to testing, testing leads to deployment. In contrast, agile methods work within frequent iterations that result in working software. These iterations are time-boxes of typically two to four weeks and constrain the team to a fixed period of time for delivery. The highest priority items with the most complexity and risk are developed first, giving the team the ability to deploy the product to the customer, whether internally or externally, prior to low value items being completed. As expected the results are shown in Figure 6.4.

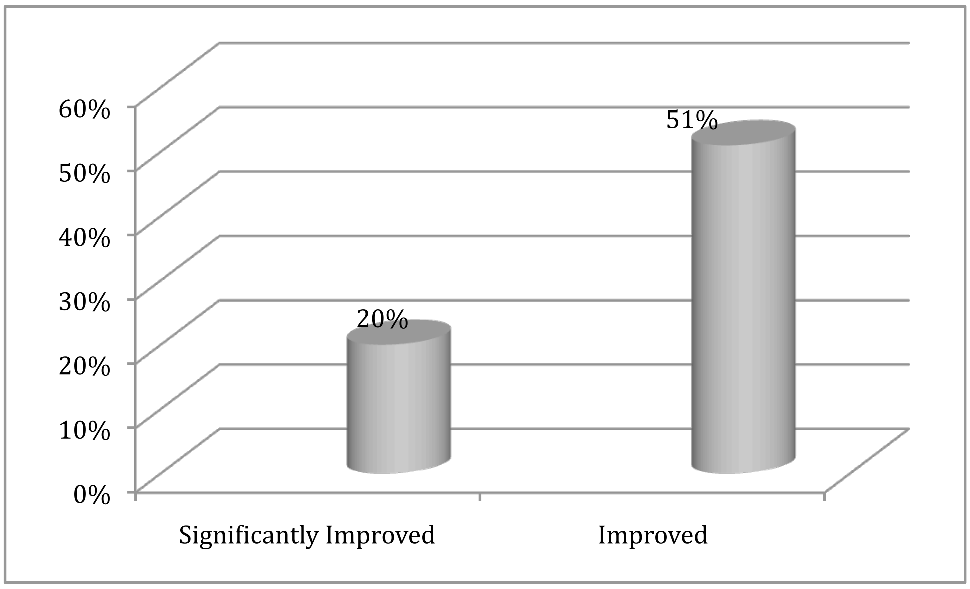


Figure 6.4 Accelerated Time-to-Market.

As shown in Figure 6.5, 60 percents practitioners estimated a 25% or greater improvement in time-to-market.

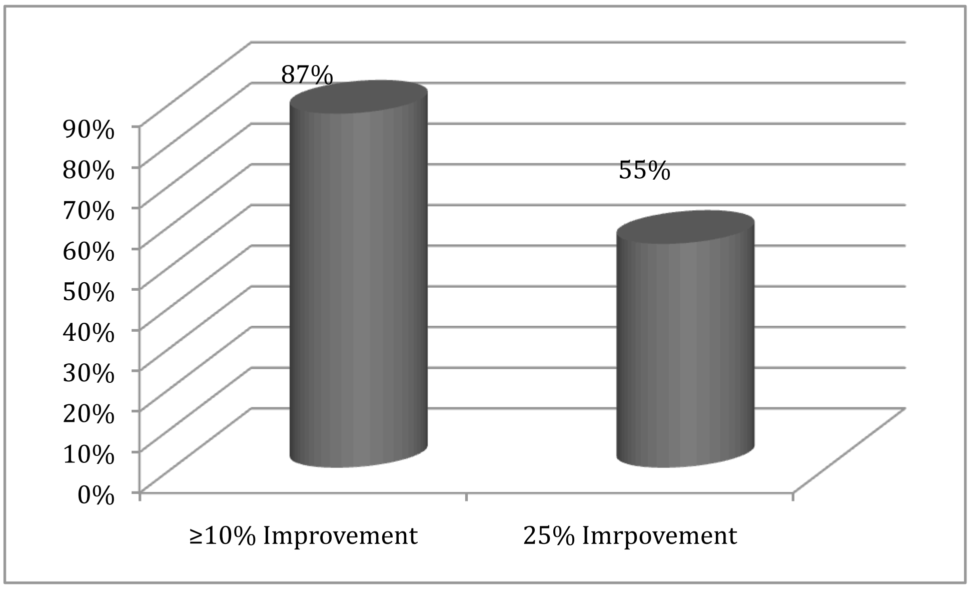


Figure 6.5 Time-to-Market Improvement.

4.4.2.3 <u>Increase Productivity</u>

The action of breaking work into manageable chunks also allows the development team to keep the finish line in sight. A greater sense of urgency and purpose is created when there is a definitive time-box of a reasonably short duration. This frequent delivery of working software provides a consistent reward and reminder of accomplishment throughout the entire project timeline. When a delivery date is six or twelve months out it is easy to lose this sense of urgency and can provide more risks for teams to be sidetracked by lower value priorities. The constant reprioritization of work items at each iteration also acts as a control mechanism limiting the amount of unnecessary or low value scope creep. The inability to let workload accumulate contributes to agile teams needing less overtime. Figure 6.6, depicts this result.

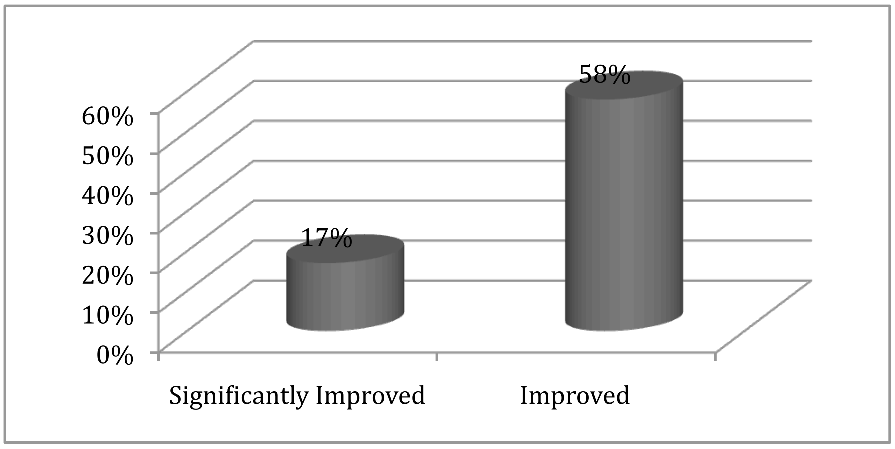


Figure 6.6 Productivity Improvement.

As illustrated in Figure 6.7, 55 percents of respondents reported a 25% or greater improvement in productivity.

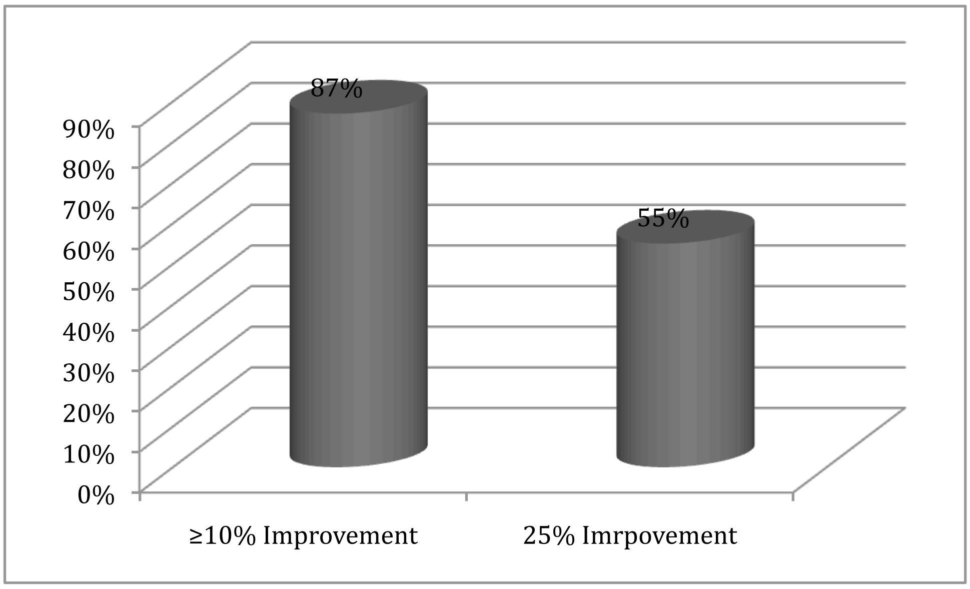


Figure 6.7 Productivity Improvement, detailed.

4.4.2.4 Quality Improvement

Each Agile team includes a cross-section of a traditional development project. This includes quality assurance or testing personnel who are moved up-front as the development is actually taking place. This allows the testing environment to gain the same advantage of constraining overall complexity. As Agile teams work together, testing is done alongside or prior to development, therefore test-driven development goes hand-in-hand with agile methods. Continuous testing and integration allows teams to catch, report and fix defects early in the delivery process rather than allowing them to compound when the cost and complexity is much higher. Quality improves significantly, also shown in Figure 6.8

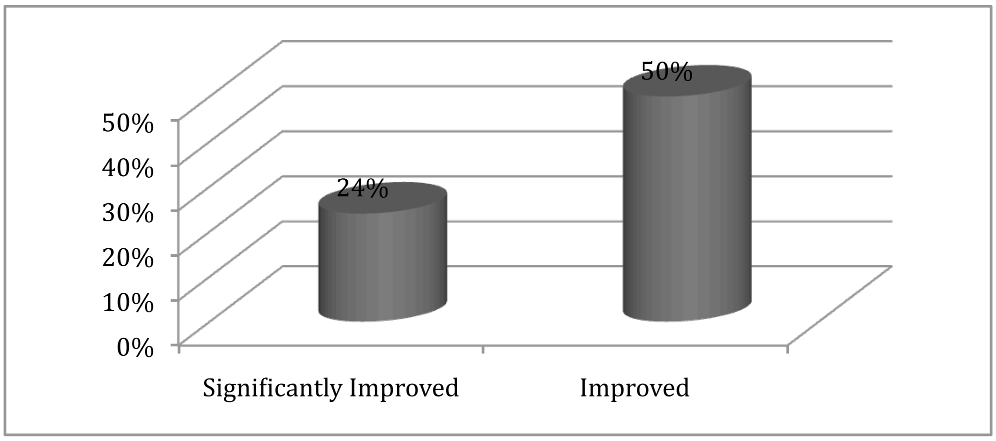


Figure 6.8 Quality Improvement.

Illustration in Figure 6.9 shows 55 percents of parishioners enjoyed a 25% or greater reduction in software defects.

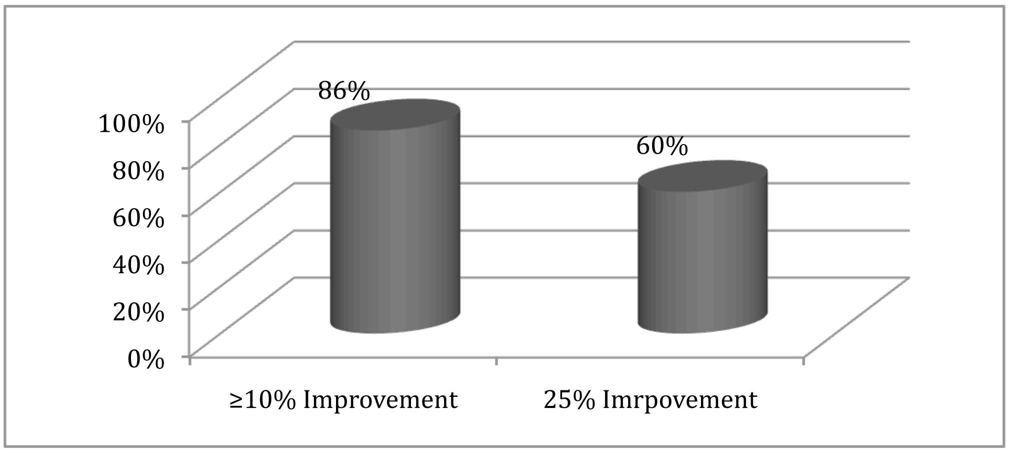


Figure 6.9 Reduced Software Defects.

4.4.2.5 Cost Reduction

Across respondents, more than 48 percents believed that development costs were reduced. Including the responses that indicated that costs were unchanged, a whopping 95 percents believe agile processes have either no effect or a cost reduction effect.

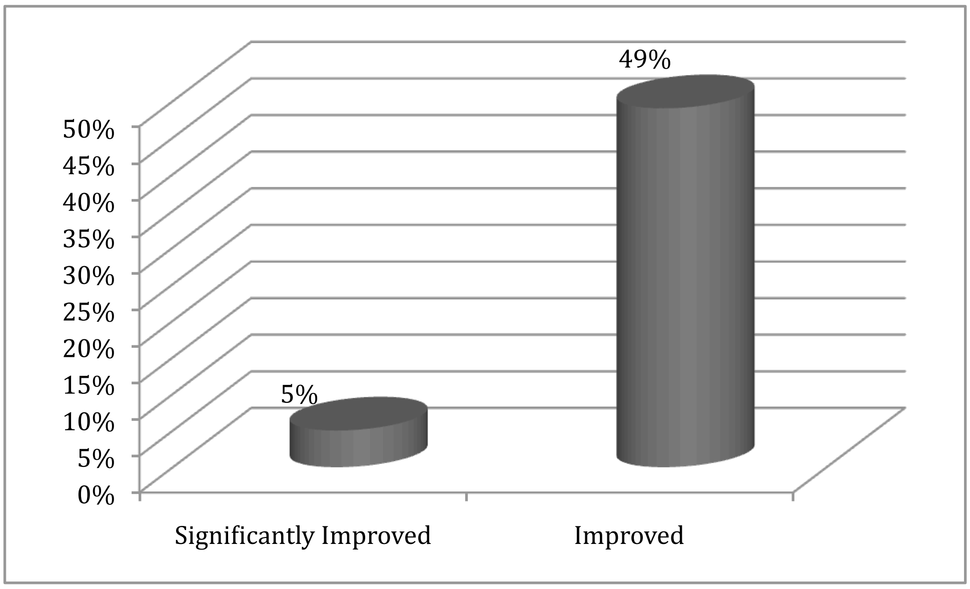


Figure 6.10 Cost Reduction

4.4.2.6 Business Satisfaction

An interesting result of analyzing the surveys reveals business satisfaction of better or significantly better was a phenomenal 83 percents for respondents. Only 1 percent believe it has had a negative effect, as shown in Figure 6.11.

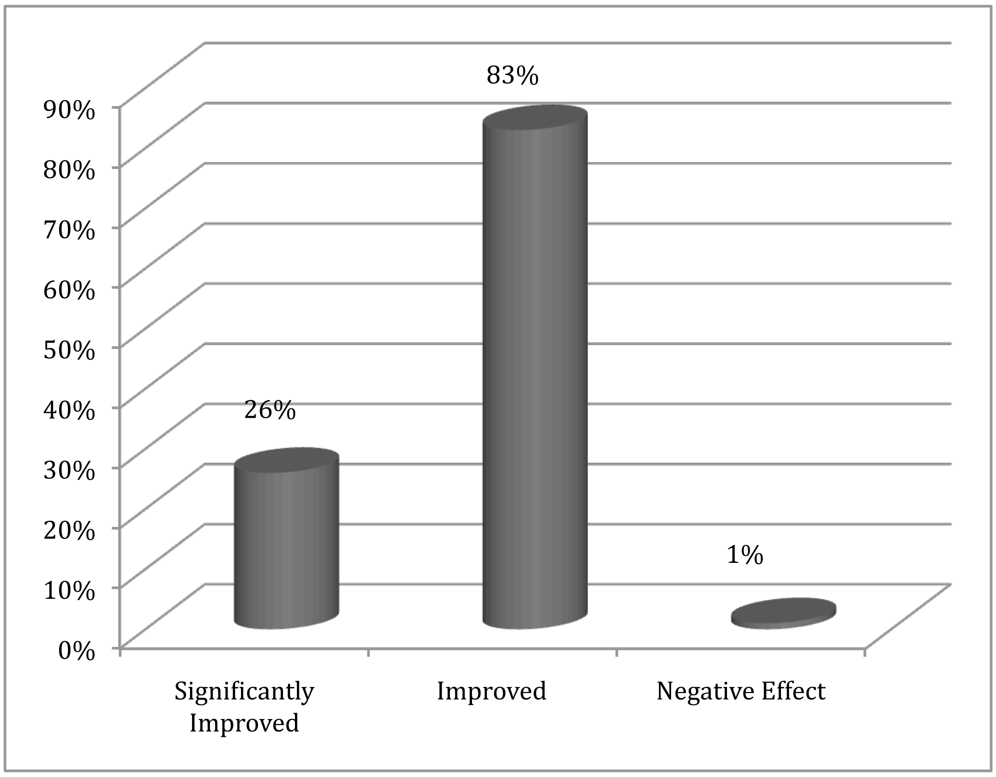


Figure 6.11 Business Satisfaction Improved.

A survey made by Shine Technologies Pty. Ltd. (2003) on agile methodologies resulted in impressive answer by majority of the respondents. Survey asked the practitioners if they intend to use or adopt agile methodologies in the upcoming year. An overwhelming 94.7 percents of all respondents would continue to use or would adopt agile methodologies. This rises to 96.4 percents when limited to knowledgeable respondents.

7 SUMMARY AND FUTURE WORK

7.1 Conclusion

There is no software development available with guaranteed success for any type of software project. Since the very first introduced development models, one after another came to existence with improvements and advantages based on the experience gained from their predecessors, focusing on their weakness and strength.

After publication of the Agile Manifesto in the early days of the current decade, agile software development methodologies came into center of attention, like never before, as they had already existed with specific names.

The agile development methodologies are believed to be a better choice for majority of software projects. The followings support the very last statement:

- They do not tend to fail as size of the developing team grows as their predecessors acted other way, in significant portion of the experiences.
- They respond quickly to the changes in such a ever changing environment of needs and requirements.
- As they respond timely to the changes, they defend the project from delays and failure in release time. In other words they improve the Time-to-Market.
- They improve the overall productivity.
- They reduce the number of defects and improve quality.
- They do not impose additional costs to the projects, even they cause in cost reduction in majority of projects.
- They improve business satisfaction.

The stated reasons are based on the result derived from employing agile development methodologies based on what practitioners reported from all across the planet, involved in projects with wide range of number of team members

7.2 Caveats

The following restrictions are necessary to be noted:

- This thesis does not focus on the tools and means used in the development projects
- The type of development projects and their usage is not considered.
- The results and statistics are based on the experience of those practitioners responded to the surveys.
- The study focuses on quality of software in general not detailed issues.
- Respondents to the surveys, from which data is used, are not categorized based on their knowledge level of agile methodologies.

7.3 Future Work

The potential and opportunities for further studies in agile development methodologies are huge.

Specifically focused research and study on agile is recommendable for the following reasons:

- Study on agile development on specific software product fields.
- Comparison between tools and methods used in projects of specific size of members as it is absent in the literature.
- Study concerning the knowledge level of agile methodologies of respondents.
- Scrutiny on cost reductability of methods, tools and means and research for amplification of reduction.

Also further work on quality improvement driving factors, such testing tools and techniques with combination of working field and programming frameworks is highly recommendable.

REFERENCES

Afjehi-Sadat A., M.N. Durakbasa, P.H. Osanna, J.M. Bauer (2004). Quality Management Systems in European Industry and the Importance of Modern Technology and Metrology.

Ahmed, A., et al. (2010). Agile Software Development: Impact on Productivity and Quality, Proceedings of the 2010 IEEE ICMIT.

Baker, Simon (2005). INTRODUCTION TO AGILE AND SCRUM, Think Pad.

Beck, Kent (2000). Extreme Programming Explained: Embrace Change, Addison-Wesley.

Boehm, B. (1988). A Spiral Model of Software Development and Enhancement, ACM SIGSOFT Software Engineering Notes.

Branson, M. J., and E. N. Herness (1992). Process for Building Object-Oriented Systems from Essential and Constrained System Models: Overview, Proceedings of the Fourth Worldwide MDQ Productivity and Process Tools Symposium.

Cockburn A. and Laurie Williams (2002). "The Costs and Benefits of Pair Programming," http://collaboration.csc.ncsu.edu/laurie/Papers/XPSardinia.PDF, Visited on Ocotober 2010

Crosby, Philip (1979). Quality is Free. New York: McGraw-Hill.

DeGrace, Peter; Stahl, Leslie Hulet (1990). Wicked problems, righteous solutions. Prentice Hall.

Drucker, Peter (1985). Innovation and entrepreneurship. Harper & Row.

Garvin, David A. (1984). What Does "Product Quality" Really Mean? , Harvard Business Review.

Gomaa, H., and D. Scott, (1981). Prototyping as a Tool in the Specification of User Requirements" Proceedings 5th IEEE International Conference on Software Engineering

Henderson-Sellers, B., and Y. R. Pant, (1998). Adopting the Reuse Mindset Throughout the Lifecycle: When Should We Generalize Classes to Make Them Reusable?, Object Magazine, Vol. 3, No. 4.

Jacobson, I., G. Booch, and J. Rumbaugh (1998). The Unified Software Development Process, Reading, Mass.: Addison-Wesley.

Juran, J. M., and F. M. Gryna, Jr. (1970). Quality Planning and Analysis: From Product Development Through Use, New York: McGraw-Hill.

Kan, Stephan H. (2002). Metrics and Models in Software Quality Engineering, Second Edition, Addison Wesley.

Kano, Noriaki (1984). "Attractive quality and must-be quality". The Journal of the Japanese Society for Quality Control.

Kitchenham B. and Pfleeger Sh. L. (1993). "Software Quality: The Elusive Target," IEEE Software, vol. 13.

Leveson, Nancy (1993). Medical Devices: The Therac-25, University of Washington.

Luckey, P. H., R. M. Pittman, and A. Q. LeVan (1992). "Iterative Development Process with Proposed Applications," Technical Report, IBM Owego.

McCall, J. A., Richards, P. K. and Walters, G. F. (1977). Factors in Software Quality,. Volumes I, II, andIII, US. Rome Air Development Center Reports.

McConnell, Steve (1993). Code Complete: A Practical Handbook of Software Construction. Redmond, Wa.: Microsoft Press.

Miller, Ade (2008). Distributed Agile Development at Microsoft patterns & practices, Microsoft Press.

Nienaber, R. C. and A. Barnard (2005). Software Quality Management Supported by Software Agent Technology, Issues in Informing Science and Information Technology.

Paschall, E. (2009). Guidelines of the Scrum Development Process, White Paper – scrumtime.org.

Robert C. Martin posting to the newsgroup comp.object, subject: “Pair Programming—Yuck!” October 28, 2001. Visited Ocotober 2010.

Royce, Winston (1970). Managing the Development of Large Software Systems, Proceedings of IEEE WESCON.

Rusk, John (2009). Earned Value for Agile Development, Optimation Ltd.

Schwaber, Ken (2004). Agile Project Management with Scrum. Microsoft Press.

Schwaber, Ken (2004). Agile Project Management with Scrum. Microsoft Press.

Shine Technologies Pty Ltd. (2003). Agile Methodologies: Survey Results.

Stephens M. and Doug Rosenberg (2000). Extreme Programming Refactored: The Case Against XP.

Stephens, Matt and Doug Rosenberg (2003). Extreme Programming Refactored: The Case Against XP.

Sutherland, Jeff (1995). "Agile Development: Lessons learned from the first Scrum.

Szalvay, V. (2004). An Introduction to Agile Software Development, Danube Technologies, Inc.

Taguchi, G. (1992). Taguchi on Robust Technology Development. ASME Press.

Takeuchi, Hirotaka; Nonaka, Ikujiro (1986). The New New Product Development Game. Harvard Business Review.

TC 176/SC (2005). ISO 9000:2005, Quality management systems -- Fundamentals and vocabulary. International Organization for Standardization.

Tian, Jeff (2007). Software Quality Engineering, IEEE Computer Society.

VersionOne (2006). Agile development: Results Delivered.

VersionOne (2008). 3rd Annual Survey: 2008 “The State of Agile Development”.

8 LIST OF FIGURES