QUALITY LIFE CYCLE OF OBJECT ORIENTED SOFTWARE DEVELOPMENT IN EXTREME PROGRAMMING

A Thesis Submitted to
The Graduate School of Engineering and Sciences of
İzmir Institute of Technology
In Partial Fulfillment of the Requirements for the Degree of

MASTER OF SCIENCE

in Computer Software

by
Gökçe MUTLU

September 2008
İZMİR

We approve the thesis of **Gökçe MUTLU**

_____
Assoc. Prof. Dr. Ahmet Hasan KOLTUKSUZ
Supervisior

_____
Prof. Dr. Şaban EREN
Committee Member

_____
Dr. Serap ATAY
Committee Member

03 September 2008

_____
Prof. Dr. Sıtkı AYTAÇ
Head of the Computer Engineering Department

_____
Prof. Dr. Hasan BÖKE
Dean of the Graduate School of Engineering and Sciences

ABSTRACT

QUALITY LIFE CYCLE OF OBJECT ORIENTED SOFTWARE DEVELOPMENT IN EXTREME PROGRAMMING

Although there are many teams using Extreme Programming, many people still think that applying its values, principles and practices will cause catastrophic results. However extreme programming is not only compatible with today's software standards, technologies and most importantly with the changes at every phase of software development but also improves the quality of software. In my thesis I analyze its values, principles, and practices and how they increase the quality comparing to old software development methodologies.

ÖZET

UÇ PROGRAMLAMADA NESNEYE YÖNELİK YAZILIM GELİŞTİRMENİN KALİTE YAŞAM DÖNGÜSÜ

Uçdeğer yazılım geliştirmeyi uygulayan bir çok takım olmasının yanı sıra getirdiği değerleri, ilkeleri ve pratikleri yetersiz bulan ve yazılımları felaketle sonuçlandıracağına inan da az değildir. Ancak uçdeğer yazılım geliştirme günümüz yazılım standartlarına, teknolojilerine ve en önemlisi yazılımın her aşamasında olan değişime ayak uydurmakla kalmayıp eski yazılım süreçlerine oranla ortaya çıkan yazılımın kalitesini de arttırmaktadır. Tezimde uçdeğer yazılım geliştirme degerlerini, ilkelerini ve pratiklerini inceleyip kaliteyi nasıl arttırdığına dair bulgularımı aktarıyorum.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1. Background

Extreme programming software development values, principles, and practices have claimed to improve the quality of the software product since their inception. The extreme programming practitioners have also claimed that use of the extreme programming approach has greatly improved the quality of their products. However, software quality is a rather complex concept. In fact some have defined the entire discipline of software engineering as the production of quality software.

1.2. Motivation

In the existing extreme programming literature there has not been a comprehensive definition of which characteristics of software quality are improved by the use of extreme programming practices in developing object oriented software.

1.3. Research Problem

In this thesis, quality life cycle of object oriented software development in extreme programming (XP) is explored. An innovative technique is introduced for evaluating XP practices and object oriented practices in order to determine which properties of software quality they improve. The technique uses a set of adapted software quality factors as defined by McCall. However these factors are reconstructed according to XP.

The whole software quality life cycle is introduced and there are two important parts for explaining it. One of them is XP practices which affect software quality and the other is object oriented practices in order to measure and as a result improve it. In

this thesis, I answer which practices to use, how these practices are combined and the responsibilities of the roles in the life cycle of software.

1.4. Structure and Outline of the Thesis

Chapter 2 is about object oriented programming. Elements of object oriented approach and its terminology are briefly explained.

Chapter 3 is a comprehensive introduction to extreme programming to understand its values, principles, and practices and to understand the technique which is introduced in this thesis. We also look at the general life cycle of a software development in extreme programming.

In chapter 4 software quality definitions of both classical and extreme programming perspective are mentioned. We also look at several software quality models in order to understand the technique introduced in this thesis.

In chapter 5 the technique is introduced comprehensively. Extreme programming and object oriented practices are discussed and also bad smells of extreme programming projects are introduced in order to show unproductive practices.

Chapter 6 is about roles in extreme programming and their involvement in quality life cycle.

Thesis ends with a conclusion giving ideas about possible future studies.

CHAPTER 2

OBJECT ORIENTED PROGRAMMING

2.1. Elements of Object Oriented Approach

Class, object, method, message, instance variable, and inheritance are the basic concepts of the Object Oriented (OO) programming. OO metrics measure how these concepts are used in the design and development process. Therefore, a short review of definitions is in order.

The basic element in an object-oriented system is an object. An object is an encapsulation of both data and functionality with the added support of message passing and inheritance. The data in an object is its attributes, while the functionality of the object is provided by its methods. Attributes and methods form a single logical entity which is called an object.

Objects themselves are created through an instantiation process that uses a general template called a class. The template contains the characteristics of the class, without containing the specific data that needs to be inserted into the template to form the object. This lack of specification is analogous to the well-known concept of referencing a stack without specifying what is in the stack. That is, certain stack features are well known and understood, although we do not yet know the type of elements in the stack.

Classes are either super classes (root classes) which created with a set of basic attributes and methods, or subclasses which inherit the characteristics of the parent class and have the ability to add (or remove) functionality when needed. An abstract class is a class that has no instances, created to facilitate sharing of state data and services among similar, more specialized subclasses. A concrete class is a class that has instances.

From the perspective of the class that inherits the characteristics of another class, the inheritance forms an IS-A relationship. This type of relationship forms a class hierarchy lattice.

Aggregate classes interact through messages, which are directed requests for services from one class which is called a client, to another class which is called a server. The class that makes the request depends upon the collaborating server class; the client is said to be coupled to the server. The serving class may have no dependence on the class using the requested material, so clearly this relationship is not commutative. The relationship in which two or more different classes form a component, consequently developing a HAS-A relationship.

2.2. Terminology

The term object is a primitive term. Objects have attributes, methods, and identity (a name). The following terminology is a partial adaptation of Booch's set of terms shown in (Archer and Stinson 1995).

Abstraction: The essential characteristics of an object that distinguish it from all other kinds of objects, and thus provide the process of focusing upon the essential characteristics of an object.

Aggregate object (aggregation): An object composed of two or more other objects.

Attribute: A variable or parameter that is encapsulated into an object.

Class: A set of objects that share a common structure and behavior manifested by a set of methods; the set serves as a template from which objects can be created.

Class structure: A graph whose vertices represent classes and whose arcs represent relationships among the classes.

Cohesion: The degree to which the methods within a class are related to one another.

Collaborating classes: If a class sends a message to another class, the classes are said to be collaborating.

Coupling: Object X is coupled to object Y if and only if X sends a message to Y.

Encapsulation: The technique of hiding the internal implementation details of an object from its external view.

Information hiding: The process of hiding the structure of an object and the implementation details of its methods. An object has a public interface and a private representation; these two elements are kept distinct.

Inheritance: A relationship among classes, wherein one class shares the structure or methods defined in one other class (for single inheritance) or in more than one other class (for multiple inheritance).

Instance: An object with specific structure, specific methods, and an identity.

Instantiation: The process of filling in the template of a class to produce a class from which one can create instances.

Message: A request made of one object to another, to perform an operation.

Method: An operation upon an object, defined as part of the declaration of a class.

Polymorphism: The ability of two or more objects to interpret a message differently at execution, depending upon the superclass of the calling object.

Superclass: The class from which another subclass inherits its attributes and methods.

Uses: If object X is coupled to object Y and object Y is coupled to object Z, then object X uses object Z.

CHAPTER 3

EXTREME PROGRAMMING

3.1. Extreme Programming Values, Principles, and Practices

3.1.1. Extreme Programming Values

Values which are defined in (Beck and Andres 2004) are the roots of the things we like and do not like in a situation. Extreme programming has five values to guide software development. These are communication, simplicity, feedback, courage, and respect.

Communication

Communication is important to be a team and it creates an effective teamwork. Problems occur and there is no escape from them. If team members communicate they will either find out that someone in the team already knows the solution or learn about it, if the problem is new, to prevent it in the future.

Simplicity

Making the solution as simple as possible so that it works is another value of extreme programming. The solution we found may be either simple or complex in the future. When we need change to make our solution simple again, we should know where we were and find a way to where we want to be.

Simplicity depends on a team's expertise and experience. The same problem can be solved in different ways by different teams.

Feedback

Extreme programming teams have comprehension of the fact that the sooner they know the sooner they adapt. Creating the perfect system at once is not possible. The most important thing that makes it impossible is change. Change is unavoidable in software development and it creates the need of feedback. Extreme programming teams use feedback to achieve their goals easily and quickly.

Courage

People in software development feel fear and courage helps them face their fears. Courage appears differently and requires different actions. If somebody knows the problem, doing something about it is courage. However if s/he feels that there is a problem but does not know about it, waiting for its emergence is also courage. When it is used alone, it can be dangerous. Doing something without being aware of the results creates problems for whole team and this does result an ineffective teamwork.

These values balance and support each other. Communication discards unneeded or deferrable requirements and helps achieve simplicity. When simplicity is achieved there is less need of communication. Feedback is a part of communication and feedbacks are useful to create simple systems. The courage to speak truths encourages communication, to remove failing solutions fosters simplicity, and to seek answers creates feedback.

Respect

The previous four values are important when team members respect each other. Otherwise extreme programming does not work and failures are inevitable.

3.1.2. Extreme Programming Principles

Values and practices are two distinct points. It is not possible to guide practices by only following values because values are too abstract. Principles connect these points.

Humanity

There is an inescapable fact in software development – People develop software. Software development does not meet human needs all the time. In (Beck and Andres 2004), it is mentioned that there are four main topics to describe what people need to be good developers.

- Basic Safety: freedom from hunger, physical harm, and treats to loved ones. Fear of job loss threatens this need.
- Accomplishment: the opportunity and ability to contribute to their society.
- Belonging: the ability to identify themselves within a group.
- Growth: the opportunity to expand their skills and perspective.
- Intimacy: the ability to understand and be understood deeply.

People who have responsibilities in software development can be successful if their needs are satisfied. Otherwise there is no escape from the costs and disruption of high turnover.

Economics

Software development is successful if two aspects are successful. One is technical success and the other one is business success. Projects have to have business values, meet business goals, and serve business needs. If these are not satisfied then projects are not successful even they are technically great.

Mutual Benefit

Every activity should benefit all the people in software development teams. Mutual benefit in extreme programming searches practices that benefit team members and their customers now and in the future.

Self Similarity

When nature finds a shape that works it uses it everywhere it can. The same principle applies to software development. However there can be problems which need unique solutions.

Improvement

In (Beck and Andres 2004), perfect is classified as a verb not an adjective because there is nothing which is perfect. However teams can perfect their tasks. Waiting for perfection is a waste of time and improvement principle aims at finding a start place, getting started, and improving from there.

Diversity

Teams need a variety of skills and perspectives in today's competitive environment. You can see that big companies often hire people from all around the world. Teams which have alike people have less conflicts however they have less skills, attitudes and perspectives to see problems, to think of multiple ways to solve problems, and to implement solutions. Diversity is required to overcome this situation.

Reflection

Reflection comes after action. For example; learning is action reflected. Good teams do not only do their works but also they think about how and why they work. They analyze why they succeeded or failed. They do not try to hide their mistakes, but expose them and learn from them.

Flow

The practices of extreme programming are biased towards a continuous flow of activities rather than discrete phases. Flow in software development is delivering a steady of valuable software by engaging in all the activities of development simultaneously. Deploying software less frequently, integrating software less often, less feedback, responding to feedbacks less often and like activities interrupt the flow of software development which creates big problems.

Opportunity

Software developments have problems but it is important to see them as opportunity for change. To perfect software development, problems need to turn into opportunities for learning and improving. This way maximizes strengths and minimizes weakness.

Redundancy

Defects are a critical problem. They decay trust which is a great waste eliminator. Defects are addressed in many practices of extreme programming such as pair programming, continuous integration, sitting together, real customer involvement, and daily deployment. Although some of the practices seem to be redundant, there is a high chance to catch defects and increase trust within team and with customer. While redundancy can be wasteful, be careful not to remove redundancy that serves a valid purpose.

Failure

Failure is not a waste if it improves knowledge. However this is not intended to excuse failure when you know something. If you know the best way to implement a story then implement it that way. However if you know three ways and you are not sure which one is the solution then try it all three ways. Even if they all fail, you will learn something valuable.

Quality

Projects do not go faster by accepting lower quality and they do not go slower demanding higher quality. Pushing quality higher often results in faster delivery while lowering quality standards often results in later, less predictable delivery.

Baby Steps

Big changes done at once are dangerous. Continuous small changes can be done rapidly that projects seem to be leaping. Baby steps are expressed in practices like test-first programming, which proceeds one test at a time, and continuous integration, which integrates and tests a few hours' worth of changes at a time.

Accepted Responsibility

Responsibility cannot be assigned. It can only be accepted. Extreme programming suggests that whoever signs up to do work also estimates it. Similarly a person who is responsible for implementing a story is eventually is responsible for the design, implementation, and testing of the story.

3.1.3. Extreme Programming Practices

3.1.3.1. Primary Practices

Sit Together

Sit together practice aims at more face to face time so the project is more productive. Providing an open space for the whole team meeting the need for privacy by having small spaces nearby is the best action for this practice. However if a team located in different places, it is important to arrange more face to face time.

Whole Team

To succeed in a project teams should embrace people with all the required skills. These people also should have the sense of being a team. As mentioned in (Beck and Andres 2004), if people think that they belong, they are in this together, and they support each others' work, growth and learning, then they constitute a team.

As change in software development is inevitable, teams should be dynamic. If new skills are required a person should be brought to the team and when he is no longer required s/he should not be in the team.

Informative Work Space

Workspaces must have information about project going on. It can be achieved by putting story cards on a wall. Therefore a new team member is able to get a general idea of how the project is going on in a short time and can get more information by looking at more closely to the wall. Another implementation of the informative workspace is visible charts. Workspaces should be used for important and active information.

Energized Work

People can work affectively when they are healthy and have free and fresh mind. For example when a person is sick, s/he should not come to the work. In order to support energized work practice, work hours are limited to eight hours per week in extreme programming. Figure 3.1 shows a map of how to balance energized work.

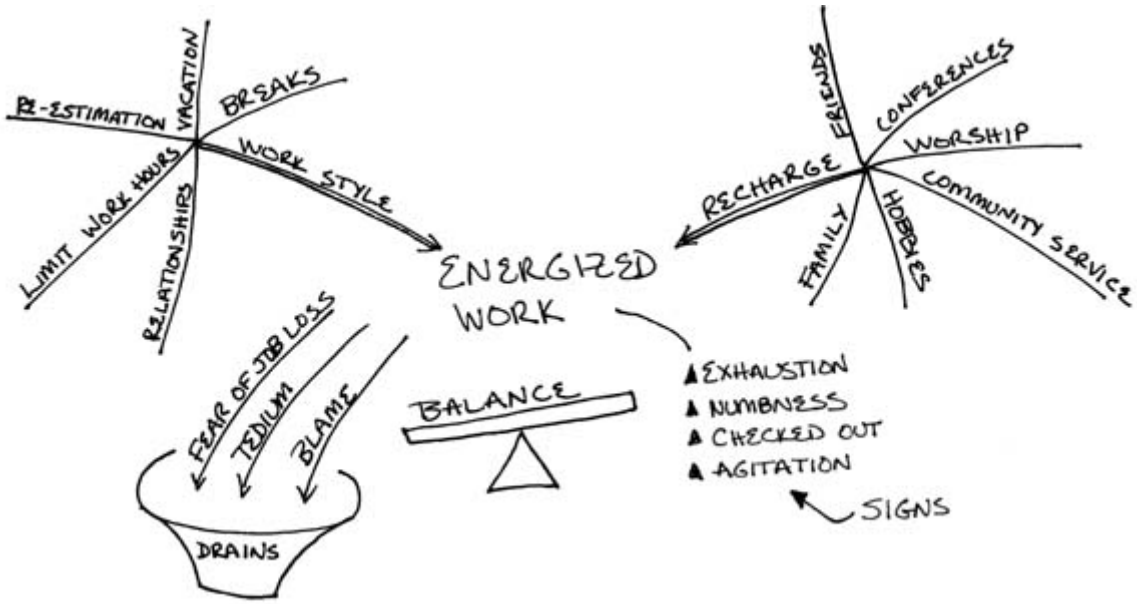

Figure 3.1. A map of energized work from

Pair Programming

Writing all code with two people sitting at one machine is tiring but satisfying. Because pair programmers support each other to concentrate on their tasks, brainstorm and clarify ideas. When one person in the pair is stuck, the other one can take an initiative. This practice requires rotating pairs frequently. However personal space must be respected for both parties to work well.

Stories

Plans are made using units of customer-visible functionality. These units represented as story cards. They should have short names, short descriptions or graphical description.

Weekly Cycle

Plan work a week at a time. Have a meeting at the beginning of every week. During this meeting:

- Review progress to date, including how actual progress for the previous week matched expected progress.
- Have the customers pick a week's worth of stories to implement this week.
- Break the stories into tasks. Team members sign up for tasks and estimate them.

Start the week by writing automated tests that will run when the stories are completed. Then spend the rest of the week completing the stories and getting the tests to pass.

Quarterly Cycle

Plan work a quarter at a time. Once a quarter reflect on the team, the project, its progress, and its alignment with larger goals.

During quarterly planning:

- Identify bottlenecks, especially those controlled outside the team.
- Initiate repairs.
- Plan the theme or themes for the quarter.
- Pick a quarter's worth of stories to address those themes.
- Focus on the big picture, where the project fits within the organization.

A season is another natural, widely shared timescale to use in organizing time for a project. Using a quarter as a planning horizon synchronizes nicely with other business activities that occur quarterly. Quarters are also a comfortable interval for interaction with external suppliers and customers.

Slack

In any plan, include some minor tasks that can be dropped if you get behind. You can always add more stories later and deliver more than you promised. It is important in an atmosphere of distrust and broken promises to meet your commitments. A few met commitments go a long way toward rebuilding relationships.

Ten-Minute Build

Automatically build the whole system and run all of the tests in ten minutes. A build that takes longer than ten minutes will be used much less often, missing the opportunity for feedback. A shorter build does not give you time to drink your coffee.

Continuous Integration

Integrate and test changes after no more than a couple of hours. Team programming is not a divide and conquer problem. It is a divide, conquer, and integrate problem. The integration step is unpredictable, but can easily take more time than the original programming. The longer you wait to integrate, the more it costs and the more unpredictable the cost becomes.

Test-First Programming

In extreme programming writing a failing automated test before starting programming or changing any code is another important practice. Test-first programming addresses many problems at once:

- Scope does not creep. By stating explicitly and objectively what the program is supposed to do, you give yourself a focus for your coding. If you really want to put that other code in, write another test after you've made this one work.
- If it is hard to write a test, it is a signal that you have a design problem, not a testing problem. Loosely coupled, highly cohesive code is easy to test.
- It is hard to trust the author of code that does not work. By writing clean code that works and demonstrating your intentions with automated tests, you give your teammates a reason to trust you.
- It is easy to get lost for hours when you are coding. When programming test-first, it is clearer what to do next: either write another test or make the broken test work. Soon this develops into a natural and efficient rhythm test-code-refactor, test-code-refactor...

Incremental Design

The question is not whether or not to design, the question is when to design. Incremental design suggests that the most effective time to design is in the light of experience. If small, safe steps are how to design, the next question is where in the system to improve the design. Eliminate duplication is the starting point. If there is the same logic in two places, it is an improvement to make one copy. Designs without duplication tend to be easy to change. You do not find yourself in the situation where you have to change the code in several places to add one feature. As a direction for improvement, incremental design does not say that designing in advance of experience is horrible. It says that design done close to when it is used is more efficient. As more teams invest in daily design, they notice that the changes they are making are similar regardless of the purpose of the system. Refactoring is a discipline of design that codifies these recurring patterns of changes.

3.1.3.2. Corollary Practices

Real Customer Involvement

The point of customer involvement is to reduce wasted effort by putting the people with the needs in direct contact with the people who can fill those needs.

Incremental Deployment

When replacing a legacy system, gradually take over its workload beginning very early in the project. After finding a little piece of functionality or a limited data set you can handle right away is the time to deploy the system. In order to make big deployment work you spend months not adding any new functionality just getting ready for the deployment day.

Team Continuity

This practice means keeping effective teams together. There is a tendency in large organizations to abstract people to things, plug-compatible programming units. Value in software is created not just by what people know and do but also by their relationships and what they accomplish together. Keeping teams together does not mean that teams are entirely static. New members begin contributing to established extreme programming teams quickly.

Shrinking Teams

As a team grows in capability, keep its workload constant but gradually reduce its size. This frees people to form more teams. When the team has too few members, merge it with another too-small team.

Root-Cause Analysis

Every time a defect is found after development, eliminate the defect and its cause. The goal is not just that this one defect will not ever recur, but that the team will never make the same kind of mistake again.

Shared Code

When extreme programming teams develop a sense of collective responsibility it is time to have a shared code. Anyone on the team can improve any part of the system at any time. If something is wrong with the system and fixing it is not out of scope for what I am doing right now, I should go ahead and fix it.

Code and Test

Customers pay for what the system does today and what the team can make the system do tomorrow. Any artifacts contributing to these two sources of value are themselves valuable. Everything else is waste. Code and test practice advice to maintain

only the code and the tests as permanent artifacts, generate other documents from the code and tests, and rely on social mechanisms to keep alive important history of the project.

Single Code Base

Multiple code streams are an enormous source of waste in software development. I fix a defect in the currently deployed software. Then I have to retrofit the fix to all the other deployed versions and the active development branch. Then you find that my fix broke something you were working on and you interrupt me to fix my fix and on and on.

There are legitimate reasons for having multiple versions of the source code active at one time. Sometimes, though, all that is at work is simple expedience, a micro-optimization taken without a view to the macro-consequences. If you have multiple code bases, put a plan in place for reducing them gradually.

Daily Deployment

Put new software into production every night. Any gap between what is on a programmer's desk and what is in production is a risk. If a programmer who is not synchronized with the deployed software makes decisions without getting accurate feedback about those decisions, his / her decisions are risky. Daily deployment is a corollary practice because it has so many prerequisites. The defect rate must be at most a handful per year. The build environment must be smoothly automated. The deployment tools must be automated, including the ability to roll out incrementally and roll back in case of failure. Most importantly, the trust in the team and with customers must be highly developed.

Negotiated Scope Contract

You can move in the direction of negotiated scope. Big, long contracts can be split. This practice advices to write contracts for software development that has fix time,

cost, and quality but calls for an ongoing negotiation of the precise scope of system, and reduces the risk by signing a sequence of short contracts instead of a long one.

Pay-Per-Use

With pay-per-use systems, you charge for every time the system is used. Money is the ultimate feedback. Connecting money flow directly to software development provides accurate, timely information with which to drive improvement.

3.2. Extreme Programming Life Cycle

Life cycle of an extreme programming (XP) project highly depends on projects. Every XP team may follow a different life cycle according to their experience and project types. However Figure 3.2 forms the base life cycle of any XP project.

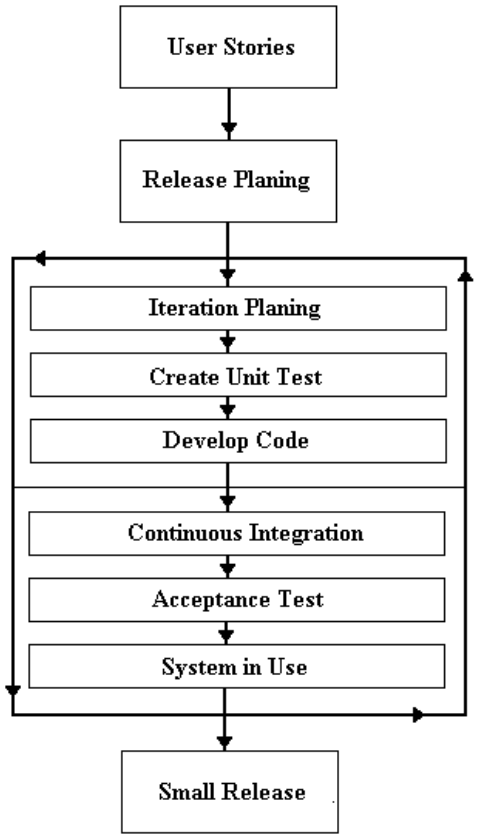

Figure 3.2. Extreme Programming Life Cycle

CHAPTER 4

SOFTWARE QUALITY

4.1. Definition

Quality is a rather abstract concept that is difficult to define but where it exists it can be recognized. However some definitions of software quality exist for both classical software development and extreme programming.

4.1.1. IEEE Definition

The definition suggested by IEEE (1991) is as below:

1. The degree to which a system, component, or process meets specified requirements.

2. The degree to which a system, component, or process meets customer or user needs or expectations.

4.1.2. Pressman's Definition

Additional aspects of software quality are included in the definition suggested by Pressman (2000).

Conformance to explicitly stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software.

4.2. Definition from XP Perspective

Classical definitions require requirements and standards documentation of software. However, in extreme programming documentation is not used and this brings out different software quality definitions.

4.2.1. McBreen's Definition

Response to changes as the customer requires. This implies that the frequent delivery of working software according to the customer's needs at the end of each iteration.

4.2.2. Ambler's Definition

Results of practices such as effective collaborative work, incremental development, and iterative development as implemented through techniques such as refactoring, test-driven development, modeling, and effective communication techniques.

4.3. Models of Software Quality Properties

Quality properties are attributes of software development and maintenance issues. The classic model of software quality properties, suggested by McCall (1977), consists of 11 properties. Subsequent models, consisting 12 to 15 properties, were suggested by Deutsch and Willis (1988) and by Evans and Marciniak (1987). The alternative models do not differ substantially from McCall's model.

4.3.1. McCall's Model

There are 11 properties and these properties are grouped into three categories as follows:

- Product operation properties: Correctness, Reliability, Efficiency, Integrity, Usability.
- Product revision properties: Maintainability, Flexibility, Testability.
- Product transition properties: Portability, Reusability, Interoperability.

4.3.1.1. Product Operation Properties

Correctness is the ability of a system to perform according to defined specification.

Reliability is the ability of a system that deals with failures to provide service.

Efficiency is the ability of a system to place as few demands as possible to hardware resources, such as memory, bandwidth used in communication and processor time.

Integrity is how well the software protects its programs and data against unauthorized access.

Usability is the ability to deal with the scope of staff resources needed to train a new employee and to operate the software system.

4.3.1.2. Product Revision Properties

Maintainability is determining the efforts that will be needed by users and maintenance personnel to identify the reasons for software failures, to correct the failures, and to verify the success of corrections.

Flexibility is the capability and effort of a system to support adaptive maintenance activities.

Testability is the ability of a system to deal with testing of an information system as well as with its operation.

4.3.1.3. Product Transition Properties

Portability is the ease of installing the software product on different hardware and software platforms.

Reusability is the ability of a system to deal with the use of software modules originally designed for one project in a new software project.

Interoperability is the ability to create interfaces with other software systems or with other equipment firmware.

4.3.2. Alternative Models of Software Quality Properties

Two models, appearing during the late 1980s, are considered to be alternatives to the McCall classic model.

- The Evans and Marciniak model
- The Deutsch and Willis model

A formal comparison of the models reveals:

- Both alternative models exclude one of the McCall's 11 properties which is the testability property.
- The Evans and Marciniak model consists of 12 properties that are classified into three categories.
- The Deutsch and Willis model consists of 15 properties that are classified into four categories.

Taken together, five new properties suggested by the two alternative models:

- Verifiability (by both models)

Verifiability requirements define design and programming features that enable efficient verification of the design and programming.

- Expandability (by both models)

Expandability requirements refer to future efforts that will be needed to serve larger populations, improve services, or add new applications in order to improve usability. The majority of these requirements are covered by McCall's flexibility property.

- Safety (by Deutsch and Willis)

Safety requirements are meant to eliminate conditions hazardous to operations of equipment as a result of errors in process control software.

- Manageability (by Deutsch and Willis)

Manageability requirements refer to the administrative tools that support software modification during the software development and maintenance periods, such as configuration management, software change procedures, and the like.

- Survivability (by Deutsch and Willis)

Survivability requirements refer to the continuity of service.

4.3.3. Comparison of Property Models

After comparing the contents of the property models, two of the five additional properties, expandability and survivability, are similar to McCall's model, though under different names, flexibility and reliability. In addition, McCall's testability property can be considered as one element in his own maintainability property. This implies that the differences between the three factor models are much smaller than initially perceived.

4.3.4. A Property Model from XP Perspective

This model has properties which define extreme programming quality. These properties are the required properties after eliminating some properties which require heavy documentation that is prescribed in plan-driven processes as a requirement for quality.

Table 1. The Description of Software Quality Properties

Property	Description
Correctness	The ability of a system to perform according to defined specification.
Robustness	Appropriate performance of a system under cases not covered by the specification. This is complementary to correctness.
Extendibility	A system that is easy to adapt to new specification.
Reusability	The ability of a system to deal with the use of software modules originally designed for one project in a new software project.
Compatibility	Software that is composed of elements that can easily combine with other elements.
Efficiency	The ability of a system to place as few demands as possible to hardware resources, such as memory, bandwidth used in communication and processor time.
Portability	The ease of installing the software product on different hardware and software platforms.
Timeliness	Releasing the software before or exactly when it is needed by the users.
Integrity	How well the software protects its programs and data against unauthorized access.

(cont. on the next page)

Table 1. (cont.) The Description of Software Quality Properties

Property	Description
Verifiability and Validation	How easy it is to test the system.
Ease of Use	The ease with which people of various backgrounds can learn and use the software.
Maintainability	The ease of changing the software to correct defects or meet new requirements.
Cost Effectiveness	The ability of a system to be completed within a given budget.

These are going to be discussed in chapter 5 in detail.

CHAPTER 5

QUALITY ACTIVITIES IN EXTREME PROGRAMMING

The technique proposed here basically breaks extreme programming down into practices. Then for each practice of extreme programming, an evaluation of what software quality properties are met is done. This action is repeated until all the quality factors are covered.

5.1. Map of Activities of OO Software Development in XP

Each of the factors defined in Table 1 is evaluated in relation to the corresponding extreme programming practices that implement the properties.

The process starts by selecting a quality assurance parameter and analyzing the meaning of the parameter. For example correctness means "The ability of a system to perform according to defined specification". The analysis should then lead to the identification of features of the development process that ensure performance of the intended system to suit the defined specification.

For example when using XP user stories ensures that the requirements are represented in a simple language that can be easily understood by customers. When user stories are combined with the practice of test first programming then each implementation of the user stories is tested as the system is developed. Continuous testing ensures correctness.

Pseudo Code

DEFINE X as an integer, Matrix as a diagonal matrix
SET X equal to 1.
FOR each Property of Quality Properties

```
BEGIN
    ASSIGN the reference of Property to 1st column in Xth row of Matrix
    DEFINE Property List as a linked list
    FOR each Practice of Extreme Programming Practices
    BEGIN
        IF Property meets the definition of Practice
        BEGIN
            ADD Property to Property List
        END
    END
    ASSIGN the reference of Property List to 2nd column in Xth row of Matrix
    INCREMENT X by 1.
END
RETURN Practice Matrix
```

This approach is followed for each software quality assurance parameter.

An Iteration of the Algorithm

Correctness means "The ability of a system to perform according to defined specification".

User stories ensure that the requirements are represented in a simple language that can be easily understood by customers.

When user stories are combined with the practice of test first programming then each implementation of the user stories is tested as the system is developed.

Continuous testing ensures correctness.

This approach is followed for each software quality property.

Compatibility

Extreme programming practices that ensure correctness of a system include the following: A general feature of all Object-Oriented (OO) software development. Possible improvement on the extreme programming approach includes design and architectural considerations that aim for platform independency.

Cost Effectiveness

Extreme programming practices that ensure cost effectiveness of a system include the following: controlling the scope, for example iterations in XP are used to prevent sudden requirement changes. Each iteration has its stories and stories are implemented according to their priorities. New stories can be introduced to iterations but some stories can be left to following iterations.

Possible improvements include avoiding scope creep without locking requirement changes. It is generally difficulty to convince a customer to sign a contract for a project whose cost is based on the cost of each iteration. The advantage of costing based on iterations however is that since iterations are short (one to four weeks) the customer gets frequent feedback on the project costs.

Correctness

Extreme programming practices that ensure correctness of a system include the following as obtained from the generic principles that guide XP development: writing code from minimal requirements, specification, which is obtained by direct communication with the customer, allowing the customer to change requirements, user stories, and test-first development. Since all the development in XP is done iteratively these practices ensure the correctness at iteration level before making the decision to continue or cancel the project.

These extreme programming practices can be improved by implementing the following: Consider the possibility of using formal specification in XP development

(which some developers are already using), possible use of general scenarios to define requirements.

Ease of Use

Extreme programming practices that ensure ease of use of a system include the following: since the customer is part of the team, and customers give feedback frequently, they will likely recommend a system that is easy to use. The frequent visual feedback that customers get during the delivery of an iteration allows them to provide useful feedback to improve the usability of the system. These can be improved upon by designing for the least qualified user in the organization.

Efficiency

Extreme programming practices that ensure efficiency of a system include the following: application of good coding standards. The most efficient algorithms are encouraged.

Extendibility

Extendibility of a system is a general feature of all Object-Oriented software development however emphasis should be on technical excellence and good design. The improvement on these practices includes the usage of modeling techniques for Object Oriented software architecture.

Integrity

Integrity of a system is ensured at operating system level and also at the development platform level. Improving the integrity of the techniques that define the product would improve system integrity.

Maintainability

The application of Object-Oriented design principles leads to maintainable systems. Development technologies that improve the interfaces between different object modules can have a positive impact on maintainability.

Portability

Originally defined as a major part of Object-Oriented design and now further enhanced by the concepts of distributed computing and web services, this quality factor is generally implemented through the concepts of Object-Oriented design.

Reusability

This quality factor is generally implemented through the concepts of Object-Oriented technology. More work on patterns can improve the reusability.

Robustness

Extreme programming practices that ensure robustness of a system originally defined as a major part of Object-Oriented design which XP development follows. This is case dependent however XP development ensures robustness in the general sense through the development standards that are inherent to particular development platform in use.

Timeliness

Extreme programming practices that ensure timeliness of a system include the following: iterative development, quick delivery, and short cycles. This can be improved upon by reducing the time for the deployment process.

Verifiability and Validation

Extreme programming practices that ensure verification and validation of a system include the following: test-driven-development, unit tests and frequent integration. The improvement on these practices can be automated testing approach.

Table 2 lists the identified practices for Extreme Programming (XP) and Object-Oriented (OO) software development.

Table 2. Quality Activities in XP and OO Software Development

Software Quality Parameters	XP Quality Activities
Correctness	User stories, Unit tests, Customer feedback, Informative workspace, Acceptance testing
Robustness	Generic OO design practices
Extendibility	Simple design, Continuous improvement, Refactoring, Shared code
Reusability	Generic OO design practices
Compatibility	Generic OO design practices
Efficiency	Simplicity, Coding standard, Pair programming, Shared code
Portability	Generic OO design practices
Timeliness	Iterative incremental development
Integrity	Generic OO design practices
Verifiability and Validation	Unit testing, Continuous integration, Acceptance testing
Ease of Use	Simple design, On-site customer
Maintainability	Iterative development
Cost Effectiveness	Iterative development, quick delivery

5.2. Extreme Programming Practices Affecting Software Quality

“Informative Work Space” embraces visible charts of the whole system for each iteration. These charts are instead of a formal architecture. They present simple shared stories of how the system works. The core flow of the system being built can bee seen by looking at the charts. The main purpose for this is communication. It bridges the gap between developers and users to ensure an easier time in discussion and in providing examples.

Having a “Real Customer Involvement” is an important practice in extreme programming because customers help developers refine and correct requirements. The customer should support the development team throughout the whole development process.

“Pair Programming” means two programmers continuously working on the same code. Pair programming can improve design quality and reduce defects. This shoulder-to-shoulder technique serves as a frequent design and code review process, and as a result defect rates are reduced. This action has been widely recognized as continuous code inspection.

Refactoring "is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior. Its heart is a series of small behavior preserving transformations. Each transformation (called a 'refactoring') does little, but a sequence of transformations can produce a significant restructuring." Refactoring is the heart of “Incremental Design” practice. Because each refactoring is small, the possibility of going wrong is also small and the system is also kept fully functional after each small refactoring. Refactoring can reduce the chances that a system can get seriously broken during the restructuring. During refactoring developers reconstruct the code and this action provides code inspection functionality. This activity reduces the probability of generating errors during development.

“Continuous Integration” means the team does not integrate the code once or twice. Instead the team needs to keep the system fully integrated at all times. Integration may occur several times a day. "The key point is that continuous integration catches

enough bugs to be worth the cost". Continuous integration reduces time that people spend on searching for bugs and allows detection of compatibility problems early. This practice is an example of a dynamic QA technique.

Acceptance testing is carried out after all unit test cases have passed. This activity is a dynamic QA technique. A classical software development methodologies include acceptance testing but the difference between extreme programming acceptance testing and traditional acceptance testing is that acceptance testing occurs much earlier and more frequently in an XP development. It is not only done once.

Early feedback is one of the most valuable characteristics of extreme programming practices. The short release and moving quickly to a development phase enables a team to get customer feedback as early as possible, which provides very valuable information for the development team.

5.3. Object Oriented Programming Practices Affecting Software Quality

In order to understand object oriented metrics we should understand traditional metrics. Three of these metrics are explained in section 5.3.1. After this section two popular object oriented metric suits are explored. These are Chidamber and Kemerer (CK) metrics model and metrics for object oriented design (MOOD) metrics model.

5.3.1. Traditional Metrics

McCabe Cyclomatic Complexity (CC)

The measurement of CC by McCabe (1976) was designed to indicate a program's testability and understandability (maintainability). Cyclomatic complexity (McCabe) is used to evaluate the complexity of an algorithm in a method. This metric is based on graph theory. The general formula to compute CC is:

$$M = V(G) = e - n + 2p \quad (5.1)$$

where

V(G) = Cyclomatic number of G
e = Number of edges
n = Number of nodes
p = Number of unconnected parts of the graph

CC cannot be used to measure the complexity of a class because of inheritance, but the CC of individual methods can be combined with other measures to evaluate the complexity of the class.

To have good testability and maintainability, McCabe recommends that no program module should exceed a CC of 10.

Source Lines of Code (SLOC)

The SLOC metric measures the number of physical lines of active code which does not include blank or commented lines. The functionality is not interconnected with SLOC however methods of large size always pose a higher risk in the attributes of Understandability, Reusability, and Maintainability. SLOC can also be very effective in estimating effort to develop methods.

Comment Percentage (CP)

The comment percentage is calculated by the total number of comments divided by the total lines of code less the number of blank lines. A comment percentage of about 30% is the most effective percentage. Since comments assist developers and maintainers, this metric is used to evaluate the attributes of Understandability, Reusability, and Maintainability.

5.3.2. Chidamber and Kemerer Metrics Model

Chidamber and Kemerer (CK) metrics model is the most popular suite in object oriented measurement suits. They claim that using their metrics it can be understood if software is being developed with object oriented practices.

Weighted Method per Class (WMC)

WMC measures the complexity of a class. Complexity of a class can for example be calculated by the cyclomatic complexities of its methods. High value of WMC indicates the class is more complex than that of low values. So class with less WMC is better. As WMC is complexity measurement metric, we can get an idea of required effort to maintain a particular class.

Depth of Inheritance Tree (DIT)

DIT metric is the length of the maximum path from the node to the root of the tree. So this metric calculates how far down a class is declared in the inheritance hierarchy. Figure 5.1 shows the value of DIT for a simple class hierarchy. This metric also measures how many ancestor classes can potentially affect this class. DIT represents the complexity of the behavior of a class, the complexity of design of a class and potential reuse.

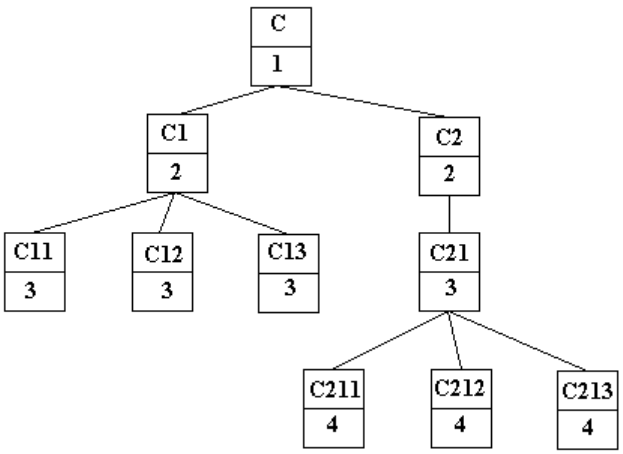

Figure 5.1. The Value of DIT for the class hierarchy

If DIT increases, it means that more methods are to be expected to be inherited, which makes it more difficult to calculate a class's behavior. Thus it can be hard to understand a system with many inheritance layers. On the other hand, a large DIT value indicates that many methods might be reused.

Number of Children (NOC)

This metric measures how many sub-classes are going to inherit the methods of the parent class. As shown in Figure 5.1, class C1 has three children, subclasses C11, C12, and C13. The size of NOC approximately indicates the level of reuse in an application. If NOC grows it means reuse increases. On the other hand, as NOC increases, the amount of testing will also increase because more children in a class indicate more responsibility. So, NOC represents the effort required to test the class and reuse.

Coupling Between Objects (CBO)

An object is coupled to another object if two object act upon each other. A class is coupled with another if the methods of one class use the methods or attributes of the other class. An increase of CBO indicates the reusability of a class will decrease. Thus, the CBO values for each class should be kept as low as possible.

Response for a Class (RFC)

RFC is the number of methods that can be invoked in response to a message in a class. Pressman States, since RFC increases, the effort required for testing also increases because the test sequence grows. If RFC increases, the overall design complexity of the class increases and becomes hard to understand. On the other hand lower values indicate greater polymorphism.

Lack of Cohesion in Methods (LCOM)

This metric uses the notion of degree of similarity of methods. LCOM measures the amount of cohesiveness present, how well a system has been designed and how complex a class is. LCOM is a count of the number of method pairs whose similarity is zero, minus the count of method pairs whose similarity is not zero.

Example:

C is a class with three methods M1, M2, and M3. Let I1 = {a, b, c, d, e}, I2 = {a, b, e}, and I3 = {x, y, x} where I1 is the set of instance variables used by the method M1. Two disjoint set can be found: I1 ∩ I2 (= {a, b, e}) and I3. M1 and M2 share at least one instance variable. Therefore; LCOM = 2-1 =1.

If LCOM is high, methods may be coupled to one another via attributes and then class design will be complex. So, designers should keep cohesion high, that is, keep LCOM low.

5.3.3. Metrics for Object Oriented Design Metrics Model

Metrics for object oriented design (MOOD) refers to a basic structural mechanism of the object-oriented paradigm as encapsulation (MHF, AHF), inheritance (MIF, AIF), polymorphism (POF), and message passing (COF). Each metrics is expressed as a measure where the numerator represents the actual use of one of those feature for a given design.

In MOOD metrics model, two main features are used in every metrics; these are methods and attributes. Methods are used to perform operations of several kinds such as obtaining or modifying the status of objects. Attributes are used to represent the status of each object in the system.

MOOD metrics are discussed in the context of encapsulation, inheritance, polymorphism, and coupling.

Encapsulation

The Method Hiding Factor (MHF) and Attribute Hiding Factor (AHF) were proposed together as measure of encapsulation. MHF and AHF represent the average amount of hiding between all classes in the system.

Method Hiding Factor (MHF)

The MHF metric states the sum of the invisibilities of all methods in all classes. The invisibility of a method is the percentage of the total class from which the method is hidden. The MHF denominator is the total number of methods defined in the system under consideration. The MHF metric is defined as follows:

$$MHF = \frac{\sum_{i=1}^{TC} \sum_{m=1}^{M_d(C_i)} (1 - V(M_{mi}))}{\sum_{i=1}^{TC} M_d(C_i)} \quad (5.2)$$

where

$$V(M_i) = \frac{\sum_{j=1}^{TC} is\_visible(M_{mi}, C_j)}{TC} \quad (5.3)$$

$$is\_visible(M_{mi}, C_j) = \begin{cases} 1 \text{ iff } C_j \text{ can call } M_{mi} \\ 0 \text{ otherwise} \end{cases}$$

TC: total number of classes

M_{mi}: methods

$M_d(C_i)$: methods defined (not inherited)

$V(M_{mi})$: visibility – % of the total classes from which the method M_{mi} is visible

If the value of MHF is high (100%), it means all methods are private which indicates very little functionality. Thus it is not possible to reuse methods with high

MHF. MHF with low (0%) value indicates all methods are public that means most of the methods are unprotected.

Attribute Hiding Factor (AHF)

The AHF metric shows the sum of the invisibilities of all attributes in all classes. The invisibility of an attribute is the percentage of the total classes from which this attribute is hidden. MHF and AHF represent the average amount of hiding among all classes in the system. The AHF metric is defined as follows:

$$AHF = \frac{\sum_{i=1}^{TC} \sum_{m=1}^{A_d(C_i)} (1 - V(A_{mi}))}{\sum_{i=1}^{TC} A_d(C_i)} \tag{5.4}$$

where

$$V(A_i) = \frac{\sum_{j=1}^{TC} is\_visible(A_{mi}, C_j)}{TC} \tag{5.5}$$

$$is\_visible(A_{mi}, C_j) = \begin{cases} 1 \text{ iff } C_j \text{ can reference } M_{mi} \\ 0 \text{ otherwise} \end{cases}$$

TC: total number of classes

A_{mi}: attributes

$A_d(C_i)$: attributes defined (not inherited)

$V(A_{mi})$: visibility – % of the total classes from which the attribute A_{mi} is visible

If the value of AHF is high (100%), it means all attributes are private. AHF with low (0%) value indicates all attributes are public.

Inheritance

Inherited features in a class are those which are inherited and not overridden in that class. Method Inheritance Factor (MIF) and Attribute Inheritance Factor (AIF) are proposed to measure inheritance.

Method Inheritance Factor (MIF)

The MIF metric states the sum of inherited methods in all classes of the system under consideration. The degree to which the class architecture of an object oriented system makes use of inheritance for both methods and attributes. MIF is defined as the ratio of the sum of the inherited methods in all classes of the system as follows:

$$MIF = \frac{\sum_{i=1}^{TC} M_i(C_i)}{\sum_{i=1}^{TC} M_a(C_i)} \tag{5.6}$$

$M_a(C_i) = M_d(C_i) + M_i(C_i)$

$M_a(C_i)$ = available methods
$M_d(C_i)$ = methods defined
$M_i(C_i)$ = inherited methods
TC: total number of classes

If the value of MIF is low (0%), it means that there is no methods exists in the class as well as the class lacking an inheritance statement.

Attribute Inheritance Factor (AIF)

AIF is defined as the ratio of the sum of inherited attributes in all classes of the system. AIF denominator is the total number of available attributes for all classes. It is defined in an analogous manner and provides an indication of the impact of inheritance in the object oriented software. AIF is defined as follows:

$$AIF = \frac{\sum_{i=1}^{TC} A_i(C_i)}{\sum_{i=1}^{TC} A_a(C_i)} \quad (5.7)$$

$A_a(C_i) = A_d(C_i) + A_i(C_i)$

$A_a(C_i)$ = available methods
$A_d(C_i)$ = methods defined
$A_i(C_i)$ = inherited methods
TC: total number of classes

If the value of AIF is low (0%), it means that there is no attribute exists in the class as well as the class lacking an inheritance statement.

Polymorphism

Polymorphism is an important characteristic in object oriented paradigm. Polymorphism measure the degree of overriding in the class inheritance tree.

Polymorphism Factor (POF)

The POF represents the actual number of possible different polymorphic situation. It also represents the maximum number of possible distinct polymorphic situation for the class C_i. The POF is defined as follows:

$$POF = \frac{\sum_{i=1}^{TC} M_o(C_i)}{\sum_{i=1}^{TC} [M_n(C_i) \times DC(C_i)]} \quad (5.8)$$

$M_d(C_i) = M_n(C_i) + M_o(C_i)$
$DC(C_i)$ = descendant count

$M_n(C_i)$ = new methods

$M_o(C_i)$ = overriding methods

TC: total number of classes

The numerator represents the actual number of possible different polymorphic situation. The denominator represents the maximum number of possible distinct polymorphic situation for the class C_i.

POF is only really a valuable metric if the organization using it has strict guidelines regarding the use of polymorphism, e.g. an overriding method must either extend a template method or invoke the superclass method from within its body. Without clear guidelines the value produced by POF will have little meaning in terms of the quality of a system's design.

Coupling

It is a measure of dependency. Coupling is the degree to which one class relies on another. In a perfect system, coupling should be low (loose), which means that objects are highly self-contained and do not have to depend on other classes to do work.

Coupling Factor (COF)

$$COF = \frac{\sum_{i=1}^{TC}[\sum_{j=1}^{TC} is\_client\ (C_i, C_j)]}{TC^2 - TC} \tag{5.9}$$

where

$$is\_client\ (C_i, C_j) = \begin{cases} 1 \text{ iff } C_c \Rightarrow C_s \wedge C_c \neq C_s \\ 0 \text{ otherwise} \end{cases}$$

TC: total number of classes

TC^2 – TC: maximum number of couplings in a system with TC classes

The client – server relation ($C_c \Rightarrow C_s$) means that C_c (client class) contains at least one non-inheritance reference to a feature (method or attribute) of class C_s (server class).

The numerator represents the actual number of couplings not imputable to inheritance. The denominator stands for the maximum number of couplings in a system with TC classes.

5.3.4. Summary of Metrics for Extreme Programming

Chidamber and Kemerer Metrics or Metrics for Object Oriented Design Metrics Model shows if code is developed according to object oriented practices. However these suites must be automated. Tools can be used for this purpose.

One of the suites should be selected according to needs. The selected suite should be applied every iteration and there are two phases. In the first phase developers have responsibilities applying the suite to their code. The good time is after their entire unit tests pass. The second phase is after integration and before acceptance test.

5.4. Bad Smells in Extreme Programming

Bad smells are the identification of early warning signals. To improve the quality of projects, some parts of them should be rewritten, refactored. In this chapter this definition is extended to the whole software development process in extreme programming.

Amr Elssamadisy and Gregory Schalliol explained the bad smells in big projects according to their experience in (Elssamadisy and Schalliol 2001).

Quartering the Chicken

Story cards are the fundamental units of Extreme Programming (XP). In each development cycle new functionalities are introduced and these functionalities are divided into stories. If one activity is similar to the previous activities, stories are

divided as previously done. However procedures used in previous iterations may not be appropriate for the new iterations. Because of this in each new iteration, the requirements should be reconsidered and story card division should be done at a more granular level.

When Should the Customer Be Happy?

Real customer involvement is an important practice in XP because customers have also tasks. They should provide honest and substantial feedback in each iteration. If they do not say anything in the early iterations but they start complaining about many things for all iterations, XP teams may have to pay for this. In (Elssamadisy and Schalliol 2001), it is associated with the relation of a tailor and his / her customer. If the customer does not return for new measurements to tailor's shop, then the suit will not fit the customer. In XP customers should provide useful feedback to XP teams from early iterations.

Functions Work but just not Together

For complex applications if there is no complete overview about the overall functionality, in the end of iterations when the stories are joined, interconnections may not be established easily. There should be a picture that remind a XP team of the all interconnections in a system that rapidly become complex. Story cards by themselves are not enough to understand the whole application when it is complex. In XP there is not up front design however for big projects there should be overview of applications. Informative workspace practice also suggests this kind of pictures, diagrams or graphics.

Finishing vs. "Finishing"

Estimation is an important activity in XP. XP empowers each member of a team to estimate their own tasks. However estimation takes time to learn. Junior team members may estimate incorrectly and this may lead them to finish their stories with full of bugs. XP delivers high quality products therefore these bugs should be resolved before delivery. It means that even a story card is told to be finished it is actually not

finished yet. If all the story cards are finished but it is still required to have more time before delivery, XP teams should create a precise list of tasks that must be completed before a story is considered finished.

Factory vs. Instances and Look-Ahead Design

In XP everybody should do the simplest thing that could possibly work. When a team needs to develop a single object in the early iteration they only create it. In the proceeding iterations they may need to develop a similar object with different functionalities. After some iterations turning back and changing the design is difficult and costly. XP teams should create a factory instead of creating different instances. In (Elssamadisy and Schalliol 2001), it is advised to look ahead and use the common sense. Even if teams do not need extra flexibility in the further iterations, the cost of design is negligible in this case.

Large Refactorings Stink

If XP teams end up large refactoring they were lazy in early iterations and they did not do small refactorings. It is important to refactor continuously and not to put band-aids on the code.

Automated Functional Tests

All the unit tests may pass but the system may still be broken. It is important to have automated functional test as well as unit tests. After a bug is fixed, functional tests should be carried out as well.

Object Mother and the Special Instance of a Factory for Test Fixtures

The smell is extensive setup and teardown functions in unit tests and difficulty in setting up complex objects in different parts of their lifetime. In order to test a scenario developers need a business object or group of business objects in their different states. Not to write large setup and teardown codes every time, developers should prepare fixtures that return objects in different states.

5.5. Comparison: Waterfall vs. XP

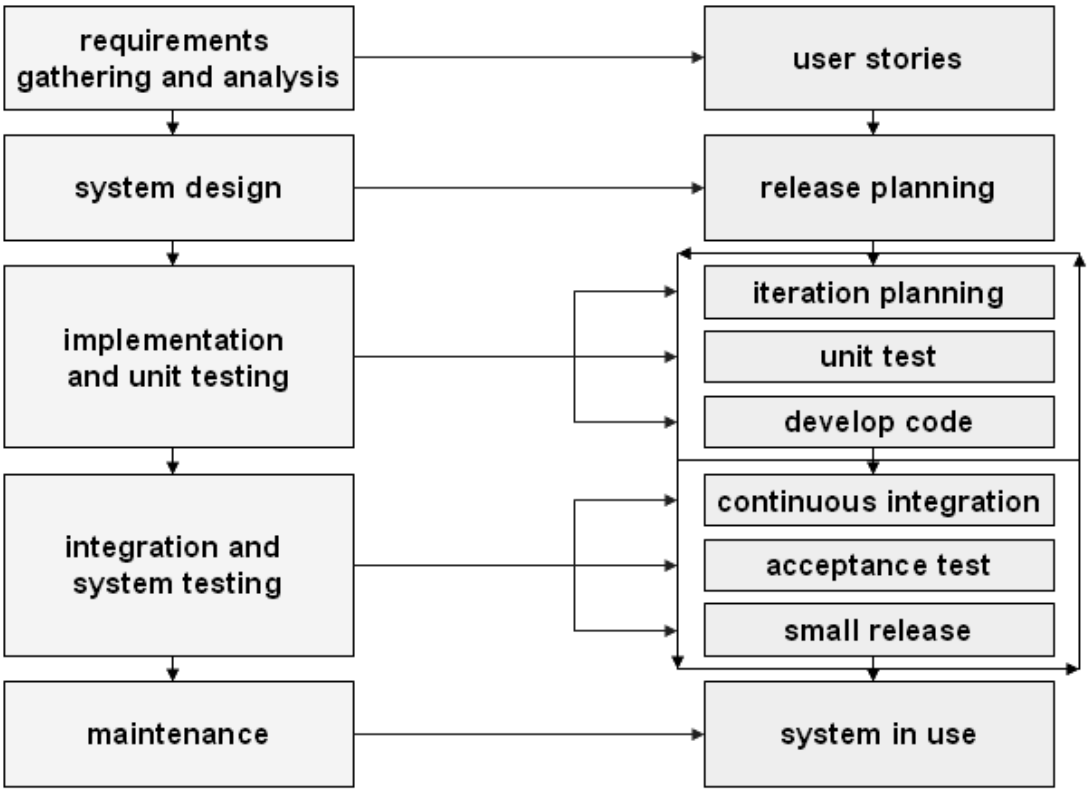

Figure 5.2. Life cycles of Waterfall and XP methodologies

Figure 5.3. shows the QA activities in XP. Many of the extreme programming quality activities such as customer feedback, unit testing, acceptance testing occur much earlier than they do in waterfall model.

These activities are done more frequently in extreme programming than in waterfall model and in each iteration, these activities will be included.

Extreme programming has more dynamic verification and this means it has more test then analysis during the life cycle.

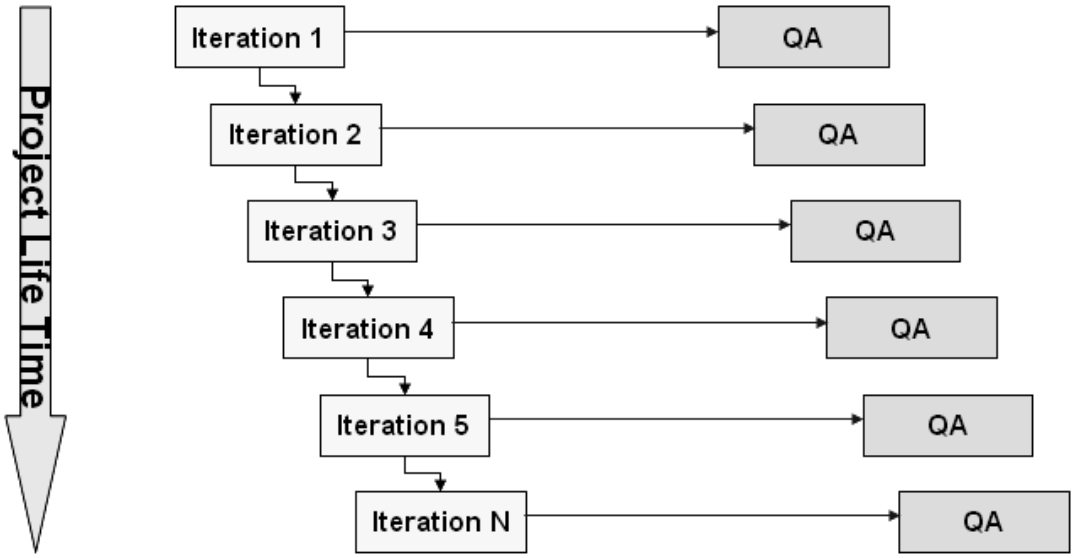

Figure 5.3. QA Activities in XP

CHAPTER 6

THE WHOLE TEAM

Extreme programming team includes testers, interaction designers, architects, project managers, product managers, executives, technical writers, users, programmers, human resources.

Roles on a mature extreme programming (XP) team are not fixed and rigid. The goal is to have everyone contribute the best he has to offer to the team's success. At first, fixed roles can help in learning new habits, like having technical people make technical decisions and business people make business decisions. After new, mutually respectful relationships are established among the team members, fixed roles interfere with the goal of having everyone do his best. Programmers can write a story if they are in the best position to write the story. Project managers can suggest architectural improvements if they are in the best position to suggest architectural improvements.

Testers on an XP team help customers choose and write automated system-level tests in advance of implementation and coach programmers on testing techniques. On XP teams much of the responsibility for catching trivial mistakes is accepted by the developers. Test-first programming results in a suite of tests that help keep the project stable. Testers' role in development is to help define and specify what will constitute acceptable functioning of the system before the functionality has been implemented.

Architects on an XP team look for and execute large-scale refactorings, write system-level tests that stress the architecture, and implement stories.

The role of technical publications on an XP team is to provide early feedback about features and to create closer relationships with users.

Programmers on an XP team estimate stories and tasks, break stories into tasks, write tests, and write code to implement features, automate tedious development process, and gradually improve the design of the system. Programmers work in close

technical collaboration with each other, pairing on production code, so they need to develop good social and relationship skills.

CHAPTER 7

CONCLUSION

Even though some agile practices are not new, agile methods themselves are recent and have become very popular in industry. Extreme programming (XP) introduces a paradigm shift in project management in the sense that every part of the software development process is reviewed with the aim of reducing the activities and number of deliverables to the minimum needed in any given situation. Such an approach appears to take control away from a traditional project manager. The move is in fact from a command oriented management structure to a facilitator oriented management system. As seen from the way software quality factors are defined in XP processes, the central players in the development process are the customer and developer and not the manager. There is an important need for developers to know more about the quality of the software produced. Developers also need to know how to revise or tailor their XP methods in order to attain the level of quality they require.

In this thesis I have analyzed XP practices' quality assurance abilities and their frequency. XP methods do have practices that have QA abilities, some of them are inside the development phase and some others can be separated out as supporting practices. The frequency with which these XP QA practices occur is higher than in other traditional development processes development. XP QA practices are available in very early process stages due to the XP process characteristics.

REFERENCES

Abreu, F.B. and W. Melo. 1996. Evaluating the impact of object-oriented design on software quality.

Archer, C. and M. Stinson. 1995. Object-oriented software measures. *Carnegie Mellon University Software Engineering Institute Technical Report*.

Beck, K. and C. Andres. 2004. *Extreme programming explained, embrace change*. Boston: Pearson.

Chidamber, S.R. and C.F. Kemerer. 1993. A metrics suite for object-oriented design. *M.I.T. Sloan School of Management* 53-315.

Elssamadisy, A. and G. Schalliol. 2002. Recognizing and responding to bad smells in extreme programming. *Proceedings of the 24th international Conference on Software Engineering* 617-622.

Galin, D. 2004. *Software quality assurance, from theory to implementation*. London: Pearson.

Harrison, R. and S.J. Counsell and R.V. Nithi. 1998. An Evaluation of the MOOD set of object-oriented software metrics. *IEEE Transactions Software Engineering* 24(6):491-496.

Huo, M. and J. Verner and L. Zhu and M.A. Babar. 2004. Software Quality and Agile Methods. *In Proceedings of the 28th Annual international Computer Software and Applications Conference* 520-525.

Lorenz, M. and K. Jeff. 1993. *Object-oriented software metrics*. New York: Prentice Hall

Mnkandla, E. and B. Dwolatzky. 2006. Defining agile software quality assurance, *proceedings of the international conference on software engineering advances*. Washington: IEEE Computer Society.

Rosenberg, L.H. and L.E. Hyatt. 1997. Software quality metrics for object-oriented environments.