Impact of Software Comprehension in Software Maintenance and Evolution

Usman Akhlaq, Muhammad Usman Yousaf
This thesis is submitted to the School of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Computer Science. The thesis is equivalent to 20 weeks of full time studies.

Contact Information:
Author(s):
Usman, Akhlaq
Address: Peder holmsg 4A, 372 35 Ronneby
E-mail: mani_xtreme@hotmail.com

Muhammad Usman, Yousaf
Address: Peder holmsg 4A, 372 35 Ronneby
E-mail: usman_comsian@yahoo.com

University advisor:
Jeff Winter

School of Computing
Blekinge Institute of Technology
Box 520
SE – 372 25 Ronneby
Sweden

Internet : www.bth.se/tek
Phone : +46 457 38 50 00
Fax : +46 457 102 45
ACKNOWLEDGEMENTS

In the name of Allah, the Most Beneficent, the Most Merciful.

First and foremost we would like to thank Allah the Almighty for blessing us with the ability and opportunity to reach our goal. It is a pleasure for us to express our gratitude wholeheartedly to Mr. Jeff winter, our thesis supervisor, whose guidance, encouragement, and support throughout the study enabled us to complete this study. His crucial contribution and feedback made him a backbone of this thesis. He was really a source of inspiration for us during this study. Where we would be without our family? We would like to owe our deepest gratitude to our parents for their inseparable support and prayers, whose involvement triggered and nourished our intellectual maturity. Last but not least we would like to thank all our friends for coloring our lives, without whom the life would be quite boring.
ABSTRACT

The need of change is essential for a software system to reside longer in the market. Change implementation is only done through the maintenance and successful software maintenance gives birth to a new software release that is a refined form of the previous one. This phenomenon is known as the evolution of the software. To transfer software from lower to upper or better form, maintainers have to get familiar with the particular aspects of software i.e. source code and documentation. Due to the poor quality of documentation maintainers often have to rely on source code. So, thorough understanding of source code is necessary for effective change implementation.

This study explores the code comprehension problems discussed in the literature and prioritizes them according to their severity level given by maintenance personnel in the industry. Along with prioritizing the problems, study also presents the maintenance personnel suggested methodologies for improving code comprehension. Consideration of these suggestions in development might help in shortening the maintenance and evolution time.

Keywords: Program comprehension, software maintenance, software evolution.
Abstract ..................................................................................................................................... 4

1 Introduction ................................................................................................................................. 10
  1.1 INTRODUCTION .................................................................................................................. 10
  1.2 BACKGROUND .................................................................................................................... 10
  1.3 PURPOSE ............................................................................................................................... 10
  1.4 AIMS AND OBJECTIVE ......................................................................................................... 10
  1.5 RESEARCH APPROACH ......................................................................................................... 10
    1.5.1 Research question ............................................................................................................ 11
    1.5.2 Research Methodology .................................................................................................. 11
    1.5.3 Inclusion/Exclusion criteria ............................................................................................. 12
  1.6 VALIDITY THREATS .......................................................................................................... 12
    1.6.1 Internal validity threats .................................................................................................. 12
    1.6.2 External validity threats ................................................................................................. 12
    1.6.3 Statistical conclusion validity threats ............................................................................. 13
    1.6.4 Construct validity threats ............................................................................................... 13
  1.7 THESIS OUTLINE ............................................................................................................... 13

2 Software Maintenance ............................................................................................................... 14
  2.1 INTRODUCTION ................................................................................................................. 14
  2.2 SOFTWARE SYSTEMS .......................................................................................................... 14
  2.3 SOFTWARE MAINTENANCE ............................................................................................... 14
  2.4 MAINTAINABILITY ............................................................................................................... 15
  2.5 IMPORTANCE OF MAINTENANCE ..................................................................................... 15
  2.6 PURPOSE OF MAINTENANCE ............................................................................................. 16
  2.7 TYPES OF MAINTENANCE .................................................................................................. 16
    2.7.1 Corrective maintenance ................................................................................................. 16
    2.7.2 Adaptive maintenance ................................................................................................. 16
    2.7.3 Perfective maintenance ............................................................................................... 16
    2.7.4 Preventive maintenance ............................................................................................... 17
    2.7.5 Emergency maintenance ............................................................................................. 17
  2.8 MAINTENANCE COST AND TIME ...................................................................................... 17
  2.9 KEY CONCEPTS .................................................................................................................. 18
    2.9.1 Software Maintenance Process ..................................................................................... 18
    2.9.2 Software Maintenance Lifecycle (SMLC) ..................................................................... 18
    2.9.3 Software Maintenance vs. Development Model ............................................................ 18
  2.10 MAINTENANCE MODELS ................................................................................................. 19
    2.10.1 Quick-fix model ........................................................................................................... 19
    2.10.2 Boehm’s Model ........................................................................................................... 19
    2.10.3 Osborne’s Model .......................................................................................................... 19
  2.11 MAINTENANCE CONSCIOUS LIFECYCLE MODEL ...................................................... 19
  2.12 PROBLEMS IN MAINTENANCE ...................................................................................... 19
  2.13 MEASURE OF MAINTENANCE ....................................................................................... 20
    2.13.1 Size measures ................................................................................................................ 21
    2.13.2 Complexity measures .................................................................................................. 21
    2.13.3 Understandability measures ....................................................................................... 21
    2.13.4 Maintainability measures ............................................................................................. 21
  2.14 SUMMARY ......................................................................................................................... 21

3 Software Evolution .................................................................................................................. 22
  3.1 INTRODUCTION .................................................................................................................. 22
  3.2 EVOLUTION ........................................................................................................................ 22
    3.2.1 Lehman’s Law of Evolution .......................................................................................... 23
  3.3 SOFTWARE QUALITY ATTRIBUTES ............................................................................... 24
  3.4 MAINTENANCE VS EVOLUTION ....................................................................................... 24
3.5 EVOLVABILITY ........................................................................................................24
3.5.1 Software Evolvability test Factors ................................................................25
3.5.2 Structural Measures for Evolvability .................................................................26
3.6 EVOLUTION CHALLENGES ............................................................................26
3.6.1 Upholding and amending software quality ..................................................................26
3.6.2 Need of common evolution platform ................................................................26
3.6.3 Requirement of tools for higher order artifacts ....................................................27
3.6.4 Co-evolution between higher and lower artifacts .................................................27
3.6.5 Formal support for evolution................................................................................27
3.6.6 Evolution-oriented Languages .............................................................................27
3.6.7 Multi-language Support ......................................................................................27
3.6.8 Evolution as fundamental part in Software life cycle ...........................................27
3.6.9 Collection and manipulation of evolution records ...............................................27
3.6.10 Generalized law of software evolution ................................................................27
3.6.11 Other Evolution Challenges ..............................................................................28
3.7 SOFTWARE ARCHITECTURAL SELECTION AND SOUNDNESS ......28
3.7.1 A framework for architecture selection .................................................................28
3.8 OTHER ACTIVITIES TO SUPPORT EVOLVABILITY ..........................29
3.8.1 System Reengineering .......................................................................................29
3.8.2 Program Refactoring ..........................................................................................29
3.9 SUMMARY ............................................................................................................29
4 Software Comprehension .......................................................................................30
4.1 INTRODUCTION ...................................................................................................30
4.2 SOFTWARE COMPREHENSION ............................................................................30
4.3 PURPOSE OF SOFTWARE COMPREHENSION .................................................30
4.4 ANATOMY OF SOFTWARE COMPREHENSION ................................................30
4.5 INFORMATION REQUIRED FOR COMPREHENSION SUPPORT ...............31
4.5.1 Problem domain ..............................................................................................31
4.5.2 Execution effect ..............................................................................................31
4.5.3 Cause-effect relationship ...............................................................................31
4.5.4 Product-environment relationship ................................................................31
4.5.5 Decision support features ................................................................................31
4.6 COMPREHENSION CHALLENGES ..................................................................31
4.6.1 Stupid use of high intelligence (Tricky Code) ......................................................31
4.6.2 Different programming Styles ..........................................................................31
4.6.3 Poor Naming convention ..................................................................................32
4.6.4 Lack of Domain words in Software Vocabulary ...............................................32
4.6.5 Conflict on Software Vocabulary .....................................................................32
4.6.6 Program representation ....................................................................................32
4.6.7 Insufficient comments ....................................................................................32
4.6.8 Dangerously deep-nesting/Deep-Inheritance tree (DIT) .....................................32
4.6.9 Concept location ..............................................................................................33
4.6.10 Code duplication .............................................................................................33
4.6.11 Identification of dead code .............................................................................33
4.7 FACTOR THAT CAN IMPROVE CODE UNDERSTANDABILITY ..........33
4.7.1 Expertise ..........................................................................................................34
4.7.2 Syntax highlighting ..........................................................................................34
4.7.3 Cross referencing............................................................................................34
4.7.4 Call graphs ........................................................................................................34
4.7.5 Comments and Annotations ...........................................................................34
4.7.6 Program Slicing ................................................................................................34
4.7.7 Ripple Analysis ................................................................................................34
4.7.8 Program Decomposition ..................................................................................35
4.8 SUMMARY ............................................................................................................35
5 Survey design .........................................................................................................36
5.1 INTRODUCTION ..................................................................................................36
5.2 PURPOSE OF THE SURVEY ............................................................................36
5.3 WHY USED THIS STRATEGY OF INQUIRY? ....................................................36
# 5.4 Form of Data Collection .......................................................... 36
# 5.5 The Population and Sample ................................................... 36
# 5.6 Ethical Consideration ............................................................. 37
# 5.7 Questionnaire Building ........................................................... 37
# 5.8 Survey Instrument ............................................................... 37
# 5.9 Survey Pilot .......................................................................... 40
# 5.10 Survey Execution ................................................................. 41

## 6 Results and Analysis ............................................................... 42
  6.1 Introduction ........................................................................... 42
  6.2 Response Rate ...................................................................... 42
  6.3 Experiences in Maintenance .................................................. 43
  6.4 Maintenance Team Management ......................................... 43
  6.5 Maintenance Vs Development ............................................. 44
  6.6 Problems in Software Maintenance ...................................... 45
  6.7 Problems in Code Comprehension ....................................... 46
  6.8 Suggestions to Improve Code Comprehension ..................... 47
  6.9 Discussion on Survey .......................................................... 48
  6.10 Discussion and Answers of Research Question .................... 48
  6.11 Study Limitations .............................................................. 49
  6.12 Conclusion and Future Work ................................................ 50

## 7 References ............................................................................ 51

## 8 Appendixes ............................................................................ 54
  8.1 Appendix A: Survey Cover Letter .......................................... 54
  8.2 Appendix B: Survey Questionnaire ........................................ 55
  8.3 Appendix C: Sasadekleva’s Classification of Maintenance Problems [22] ... 59
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Osborne Software quality attributes</td>
<td>15</td>
</tr>
<tr>
<td>2.2</td>
<td>Effort needed in different stages of maintenance and development lifecycle</td>
<td>18</td>
</tr>
<tr>
<td>2.3</td>
<td>Maintenance conscious lifecycle</td>
<td>19</td>
</tr>
<tr>
<td>3.1</td>
<td>Species Evolution</td>
<td>22</td>
</tr>
<tr>
<td>6.1</td>
<td>Survey response rate</td>
<td>42</td>
</tr>
<tr>
<td>6.2</td>
<td>Respondent’s maintenance experience</td>
<td>43</td>
</tr>
<tr>
<td>6.3</td>
<td>Maintenance team management</td>
<td>43</td>
</tr>
<tr>
<td>6.4</td>
<td>Maintenance vs. development</td>
<td>44</td>
</tr>
<tr>
<td>6.5</td>
<td>Problems in software maintenance</td>
<td>45</td>
</tr>
<tr>
<td>6.6</td>
<td>Problems in code comprehension</td>
<td>46</td>
</tr>
</tbody>
</table>
LIST OF TABLES

Table 5.1 Mapping of survey questionnaire and research questions ....................................... 40
Table 6.1 Suggestions to improve code comprehension................................................................. 47
1 INTRODUCTION

1.1 Introduction
This chapter introduces the reader about the background of our thesis, the motivation behind the study, purpose, aims and objectives of the study. It also describes the research questions and the adopted methodology during the research study.

1.2 Background
Change is a basic inevitable feature of Software systems. Software maintenance and evolution is a continuous process in the software development life cycle (SDLC) to repair existing faults, eradicate deficiencies, make the software compatible and adaptable with new hardware and software requirements, overcome complexity and increase user satisfaction[1]. Change management and the ability to cope with these changes in the environment has been a challenging task for the software engineers. Software that doesn’t have the ability to face these changes in environmental behavior has to undergo early demise. Although the dimension and extent of change is not 100% predicted, but introducing evolvability and changeability from the very beginning (in software architecture & design process) improves the quality of software and reduces the maintenance cost.
Software comprehension is a set of activities done in maintenance process to get a better understanding of software functionality, architecture and overall impact on executing the change. The major hindrance in maintenance and evolution process is the understandability of the software system (Program, documentation & operating procedures) and existing problem. Software Comprehension is an important phase of maintenance process as problem understanding is the major part of problem solving [2]. There has always been a need to investigate the barriers in software comprehension to introduce best practices in development process. These efforts make the maintenance more efficient in order to face the rapidly changing demands of innovative software industry.

1.3 Purpose
The purpose of this thesis is to explore the conceptual and practical issues in software comprehension during the software maintenance and evolution process and to discuss possible measurement in improving software comprehension (from coding perspective). This involves the study of how software evolves with the change in its working environment and how the use of well formulated comprehension strategies can be helpful in software maintenance and evolution process. For grounding the results in practice, an online survey will be conducted. In our research context, the survey corresponds to survey software maintenance team members. Using the academic’s perspective and the industrial technical expert’s perspective, survey results will be used to develop an understanding of different software comprehension problems. Moreover, there is a possibility to conclude our findings in a model for improving the software comprehension.

1.4 Aims and Objective
Based on the purpose statement the ultimate goal of this study is to investigate the major issues regarding to program comprehension. As aforementioned software system is composed of program/source code and documentation, it is much seen that documentation is either insufficient or not available so maintenance personnel often have to rely on source code [2]. Hence the scope of this project is restricted to the investigation of problems in software comprehension process particular to source code.
Following objectives are set to meet the goal:
- Analyzing the key phases in the software maintenance and evolution life cycle.
• Understanding the different maintenance types discussed in literature.
• Analyzing the role of software comprehension activities on the software maintenance and evolution process.
• Understanding the software comprehension problems faced by the software maintenance team in industry.
• Analyzing the importance of software maintenance measures on the software maintenance.

1.5 Research Approach
This section outlines the research questions and the methodology used to reach the answers.

1.5.1 Research question
Research questions are interrogative statements that are tried to be answered while carrying out the study. Research questions are used to set the track on which the research is going to be done and also act as signposts to drive the reader towards the aim of the study [3]. In this study we will try to answer the following research questions:

RQ-1: What are the problems that a maintenance team has to face in code comprehension and their effects on overall maintenance and evolution process?

RQ-2: How does the good understanding of source code contribute in speeding up and improving the software maintenance and evolution process?

1.5.2 Research Methodology
Three types of research approaches are discussed in the literature i.e. qualitative, quantitative and mixed methods [3]. The main concern of quantitative research approach is to examine the causes and effects relationship using two strategies of inquiry i.e. experiments and surveys. Statistical operations are performed on collected data and the results are expressed in terms of numbers. The qualitative research approach is based on an understanding of a certain behavior and the reasons behind that particular behavior. This type of research is emergent by nature and is done naturally in the real environment. There are five types of strategies associated with qualitative research approach: the biography/narrative research, phenomenology, grounded theory, ethnography and case study [3]. Mixed method approach contains both the properties of qualitative and quantitative approaches [3].

To answer the above mentioned research questions quantitative research approach is used in this study.

As software comprehension is a wide spectrum activity. There is a broad range of related activities such as understanding of problem domain, understanding of software structure; understanding of source code etc [2]. As technical skill of the people who are doing the research is one of the responsible factors for the threats to internal validity, so in the first part of our research study a comprehensive and detailed literature review is performed. Purpose of the literature review was to find the related information about software comprehension, maintenance and evolution. For literature review we used the reliable sources like IEEE, ACM digital library and books available on software comprehension, maintenance and evolution processes. This review enabled us to get familiar with the different software engineering concepts and comprehension activities performed by the maintenance team in the process of software maintenance and evolution. Another aim of the literature review is that, it made it possible for us to design the survey for software industry to know the major problems in software maintenance process and the impact of good program comprehension on the maintenance and evolution process.

The survey is a widely used quantitative research strategy of inquiry to collect relevant data about the trends, attitudes and opinions of a selected population [3]. On the basis of viewpoints of experienced software developers and maintainers, results will be compiled and
analyzed i.e. whether our results complement the existing findings or contradict with them. The outcome of the survey will be a qualitative and quantitative analysis. At the end a report will be prepared on the basis of our findings.

1.5.3 Inclusion/Exclusion criteria

Before starting the literature review we developed a search strategy. In search strategy we first defined and formulated our research questions. Then we compiled a list of keywords to locate the useful, relevant and updated information.

For compilation of keywords first we broke down our research questions into search phrases and on evaluating the results we obtained, we modified our keywords. We tried possible alternative ways to express the concerning concepts and ideas. We also used the IEEE standard glossary of software engineering terminology to know the common terminology about the subject area.

After that we identified the reliable resources. For this study we used the IEEE Xplore and ACM digital library. To extract the relevant material the abstract, introduction and the conclusion sections of each article were studied.

We also consulted the books written by the well known authors in the field of software comprehension, maintenance and evolution. The overall inclusion /exclusion criterion for information selection is to make sure that it is very latest and relevant to the subject area.

1.6 Validity threats

There are many potential issues that may affect the soundness of the research. There are four types of validity threats mentioned in the literature [3] are as follows:

- Internal Validity threats
- External Validity threats
- Statistical Conclusion Validity threats
- Construct Validity Threats

1.6.1 Internal validity threats

The factors that cause interference in the investigator’s ability to draw correct inference from the gathered data are known as internal validity threats [3]. These factors might be inadequate procedures, technical skill of the people who are doing the research, wrong participation of the respondents due to incomprehensive and ambiguous survey questionnaire [4]. To equip us with the technical skills required in this study to overcome the internal validity, we got familiar with the core concepts of survey design and recent ongoing research about software comprehension, software maintenance and software evolution. To avoid the wrong participation of respondents, due to ambiguity in the designed questionnaire, we conducted a pilot survey which helped us in real means to find out the ambiguities in the questionnaire.

1.6.2 External validity threats

External validity threats are the factors that may drive the researcher in making wrong generalization. Wrong generalization means mapping the result of one setting to a totally different setting [3]. The responsible factors for external validity threats may be faulty sampling, sending non standardized questions to all participants and bias behavior of the respondents [4]. In our study we extenuated the external validity threats by requesting the participants to reply their own experience, sending same standardized questions to all participants and considering the results of only those participants who have some software maintenance experience in their professional career. But there may be a risk for external validity threat as people really don’t report the personal reality as they want to see them in a good light [4].
1.6.3 Statistical conclusion validity threats

Another type of validity threat is known as statistical conclusion validity threat. These threats are arisen when researchers draw wrong results from the data due to the inadequate usage of statistical power i.e. sampling [3]. In our case number of respondents was not too much and we performed the basic operations (average and percentage) on the data so the chances of statistical conclusion validity threats are very low.

1.6.4 Construct validity threats

Another type of validity threat is the construct validity threat; which is caused by the wrong usage of definitions and measures of variable [3]. Usage of correct definitions and measures of variables provided by authentic sources like IEEE and ACM gave us a great deal to mitigate this sort of threat.

1.7 Thesis outline

This section describes the overall structure of the thesis and chapter distribution for improving the readability of the thesis. This section also describes the contribution of each chapter to find the answers of defined research questions.

Chapter 2: (Software Maintenance) gives a brief overview about the software maintenance process. Detailed discussion about the importance of maintenance activities, classification of maintenance activities, time spent, maintenance problems and other related core concepts are given. Based on this literature study, questions for survey have been formulated. To some extent this chapter also contributes to answer the research question (1) and identifying more questions to answer research questions (2).

Chapter 3: (Software Evolution) explains evolution in biology in general and evolution in software engineering in particular. This chapter details out the Lehman’s laws of evolution that are considered as pioneer work in software evolution. A detailed discussion about software quality attributes, factors to test the evolvability of a software system, evolution challenges, misconception between software maintenance and evolution is given in this chapter. This chapter provides the ground knowledge to formulate the questions for the survey. To some extent this chapter contributes to answer the research question (1) and (2).

Chapter 4: (Software comprehension) details out the software comprehension process, importance of comprehension process, information required for better software comprehension, comprehension activities, different challenges and the effect of software comprehension on overall maintenance and evolution process. This chapter contributes to answer the research question (1) and (2).

Chapter 5: (Survey Design) details out the procedure of survey design. This chapter explains the purpose of the survey design, reasons for choosing survey as a preferred type of data collection, specification of the form of data collection, selection of target population, procedure to analyze the responses and to validate the accuracy.

Chapter 6: (Discussion) Actual data analysis and is done and research questions are answered in this chapter. This chapter rounds up and concludes the thesis and presents the directions for future work.
2 SOFTWARE MAINTENANCE

2.1 Introduction
In this chapter we illustrate an overview about software maintenance process. Further we discuss the importance of software maintenance with an example of software failure in the airbag system of Volvo cars. Moreover, we give the classification of maintenance activities by different authors. Maintenance problems and other related core concepts are also given in the later part of this chapter.

2.2 Software Systems
From last few decades a rapid increase has been seen in computer users. The real power of computer hardware is really equipped with software. Software systems are all around us from military control systems to normal mobile handsets, from healthcare systems to educational systems [5].

Software systems are used to analyze, visualize and simulate processes or data which facilitate both computing communities i.e. general and scientific [6]. The importance of software systems is different in different situation depending upon the nature of use but it is never neglectable. Due to the growing demand of software based computerized solutions the software systems are becoming more and more complex.

2.3 Software Maintenance
Change is an essential characteristic of software development. According to Lehman; change is built-in in software and it is a patent fact of software lifecycle there is no point to make a distinction between maintenance and development [7, 8]. User experience is the main acts to arouse these changes [9], so software systems must be able to cope with the evolving requirements, platforms, and other environmental pressures [10].

According to IEEE glossary definitions software maintenance is defined as [11]:

“Software maintenance is the process of modifying a software system or component after delivery to correct faults, improve performances or other attributes, or adapt to a changed environment.”

According to the above definition the process of maintenance seems only be done after the final release of software. But there are other definitions available that contradict the above point of view. According to Pigoski the processes of maintenance and development of software are not apart. Maintenance starts with the beginning of development of software. Pigoski gave the definition of software maintenance as follows [12]:

“Software maintenance is the totality of activities required to provide cost-effective support to a software system. Activities are performed during the pre-delivery stage as well as the post-delivery stage. Pre-delivery activities include planning for post delivery operations, supportability, and logistics determination. Post-delivery activities include software modification, training, and operating a help desk.”

Software systems should be capable of bearing the overhead of variations during the software lifecycle which might make the maintenance process effective in terms of cost and effort as more than 80% of the software resources are required to improve the existing system [7]. There are no doubts that software maintenance is the most expensive part in the SDLC, in this way the cost of final product is always influenced by that cost of maintenance as well. There has been lot of researches done to increase the easiness of maintenance but there is no high-flying enhancement noticeable yet.
McDermid clarified [13] that the software systems are not only made up of source and object code but there is another part that should be given full consideration which is documentation i.e. requirement analysis, design, specifications, user manuals and most important are the procedures that are used to run the system. Therefore maintenance is not matter of changing the code only but also the documentation in parallel. Maintenance is also known as a change management process.

2.4 Maintainability

Maintainability is one of the important software quality attributes. There are many definitions of maintainability in the literature but the main idea behind all of them is same. J. Martin and C. McClure, defined maintainability as follows:

“The ease with which a software system can be corrected when errors or deficiencies occur and can be expanded or contracted to satisfy new requirements” [14].

In other words maintainability is simply the software’s ability to support foreseen future changes. In literature the term maintainability is also referred as variability or architectural flexibility [15]. The architecture of the system should be designed in a way maintenance could be done easily on it later on.

No doubt the quality of software can be improved by introducing more quality attributes. But excessive or imbalanced use of quality attributes may severely compromise the maintainability of the software. Osborne [12] highlighted those attributes whose good use can improve the maintainability; which are reliability, understandability, testability, modularity and expandability. It can be explained by the figure below:

![S/W Quality Attributes Diagram](image)

**Figure 2.1 Osborne Software quality attributes**

2.5 Importance of maintenance

Software maintenance has become one of the most important concerns of the software industry. The maintenance of software has a significant importance to keep software operational and fulfills the needs of its user [2]. We can consider the importance of software maintenance by the following example; Airbags are used in auto vehicles as a safety instrument which works under highly sensitive software which put it into action. But with very minor software negligence in detecting the input parameters can cause severe loss of precious lives. Basically the system should be deployed within well time as the vehicle bump with any external object to save the rider being injured.
Swedish famous auto manufacturing company Volvo had to recall 65,000 cars due to airbag’s software problem. Euro NCAP is a European automotive testing authority which found that problem in Volvo products and then Volvo had to re-examine the robustness of the airbag’s software and release the cars after maintenance \[16\]. From the above example we can realize the importance of software maintenance; it would be a massive loss for Volvo if there were no software maintenance.

### 2.6 Purpose of Maintenance

In order to avoid early death software system should have the ability to evolve. Software can be considered as a moving target as the environmental requirements are continuously changing so “maintenance” should be performed continuously \[9\].

The impulse behind software maintenance has numerous reasons which are necessary to keep a system in a working condition and updated according to the user requirements. The first major aim of it is to endow with stability of services which can only be carried out with the help of maintenance. Maintenance might be done to fix the bugs, to prevent it from the risk of failure and to make changes in it if hardware is changed.

Maintenance is required whenever a change is needed to update the software for government rules and to accommodate competitor products. Sometimes the requirements of user changes and there is a need to update the software to accommodate the software accordingly. There could be several reasons behind these requirements i.e. improvement in the functionality, improved performance and to make it according to operational model. Updation in the documentation, change in the code and databases is needed sometime so maintenance should be done to handle these problems \[2\].

### 2.7 Types of Maintenance

Maintenance is classified into four major categories, which are given below. According to Pigoski the purpose of categorizing maintenance is to answer why is the maintenance must required \[12\]. As the different diseases have different cures and without a proper diagnosis of a disease the doctor is not able to write the proper prescription. Similarly before heading towards the process of maintenance proper understanding of each type make the maintenance easier and appropriate. Hence, we should have to know in which criteria the system lies. Then we can follow what kind of maintenance should be performed on the system.

There are different classifications of the maintenance activities in the literature. According to IEEE standards, maintenance activities can be categorised into four major classes (corrective, adaptive, perfective and emergency maintenance) depending upon the nature of purpose \[17\] and according to Takang and Grubb four types of maintenance are(corrective, adaptive, perfective and preventive maintenance) \[2\].

#### 2.7.1 Corrective maintenance

It is a correction of the faults and errors in the software after the delivery that has not previously been exposed. According to Takang and Grubb \[2\], the reason of the defects might be design errors, logic errors and coding errors.

#### 2.7.2 Adaptive maintenance

Adaptive maintenance is done when the requirements or the environment of a program has changed i.e. operating system or hardware. The term environment refers to all the circumstances which act on the system from outside.

#### 2.7.3 Perfective maintenance

This kind of maintenance deals with rapidly changing user requirements. As the name shows it makes system more perfect both from functional (performance) as well as well as non functional requirements (user interface). Success of a software system is not based on the success of its first release but the users just for experiment try to extend the system’s
functionality along the direction for which it was not initially designed so in result they identify the new requirements for the system. The maintenance activities that are used for this purpose are known as perfective maintenance [2].

2.7.4 Preventive maintenance
This type of maintenance activities are performed to mitigate the future potential threat of malfunction and to enhance the maintainability of the system by updating documentation, adding comments, and improving the modular structure of the system. Studies show that corrective, adaptive and perfective maintenance activities increase the system’s complexity. As the understandability and complexity are inversely proportional to each other so there should be some way to sort it out, so this work is known as preventive maintenance. It should be noted that for the three types of maintenance to be done correctly, both the documentation (e.g. requirements specifications, design specifications, test plans) and the code must be changed so that the code and documentation remain consistent with one another.

2.7.5 Emergency maintenance
In IEEE standard there is another type of maintenance known as emergency maintenance. These are unscheduled activities of corrective nature performed to keep a system operational. According to Chapin et.al [18], among the first four types of maintenance, only one of them is ‘traditional’ maintenance and rest of all are considered as evolution.

2.8 Maintenance cost and time
Software has become more expensive as compared to the hardware due to the growing demand of the former as compared to the latter [19]. Software maintenance is a time consuming act i.e. there are many steps involve to perform reliable maintenance as understandability of the structure, comprehension of code, behaviour and functionality of existing software are very time taking actions. Time and cost are highly related to each other because; as much time software takes to be operational the level of cost will obviously go up.

By considering that many researches has been done and many well-known method of software engineering has been explored, one might anticipate that the cost of maintenance would be decreasing but unfortunately they could not bring an efficient decrease in the maintenance cost yet [2]. Software maintenance was considered to be ‘iceberg’ as there are large numbers of potential problem and cost dwells beneath the plane [20]. Many researches has been done to find out maintenance cost though the results show a discrepancy but almost 60 to 80% of the total SDLC’s cost is consume for maintenance [20].

As maintenance is a time consuming act and employment of time for different types of maintenance is different. According to Lientz and Swanson [2], 20% of the total time is used to do corrective maintenance, adaptive maintenance takes almost same time as corrective does i.e. 25% and the most time consuming type of maintenance is perfective one which takes 50% of the time. The ratio of timing given above differ with different software depending upon some factors i.e. proficiency of maintenance workforce, environment and nature of associated documentation.

According to Sommerville [1] there are number of reasons that are always there to make the process of software maintenance more expensive. They are as follows:

- Software Development team stability
- Contractual responsibility
- Maintenance Staff Skills
- Program age and Structure

According to him the long term solution is to accept the universal truth that software has rarely defined lifetime. Schneidewind also highlighted that myopic view that maintenance is a post delivery activity is also a source to make maintenance harder [9].
2.9 Key Concepts
In the literature there are three key concepts related to software maintenance i.e. maintenance process, maintenance lifecycle and maintenance models [2, 12].

2.9.1 Software Maintenance Process
The software maintenance process is defined as a set of activities that are carried out to effect changes in a software system to get intended results [2]. Software maintenance process always starts with change request [12]. According to the IEEE standards the Software maintenance process is divided into the following activities [17].

- Problem identification, classification and prioritization
- Analysis
- Design
- Implementation
- Regression/System testing
- Acceptance testing and
- Delivery

2.9.2 Software Maintenance Lifecycle (SMLC)
The SMLC defines the order in which maintenance activities should be performed [2]. SMLC may start from a new idea, a new user requirement, software or hardware innovation or a problem in previous release of software and then goes through all activities mentioned in software maintenance process. Any of the driving factors may start the lifecycle again.

2.9.3 Software Maintenance vs. Development Model
Software maintenance model is an abstract representation of each activity of SMLC to get the better understanding. As software maintenance is a novel field as compared to the software development so there are a few number of process and lifecycle models that are designed keeping maintenance issues into consideration [2]. The initial phases in maintenance model require much more effort as compared to later which is totally inverse to the software development lifecycle models. Because in traditional development models the future software evolution is not much concentrated but in maintenance model there is high consideration of prediction [2].

![Figure 2.2 Effort needed in different stages of maintenance and development lifecycle](image)

The lack of consideration of evolutionary nature of software in traditional software development models make them inappropriate for the maintenance. This deficiency of development model leads us to find proper maintenance conscious models.
2.10 Maintenance Models

There are many models but we are going to have a look on very general models of software maintenance [2].

2.10.1 Quick-fix model

It is an unreliable model; it is ad-hoc in fashion. This model waits for the problem to occur and then its implementation is applied to fix the problem as quickly as possible. Although it is not an efficient model but when there are deadline and resource constraints this model is used. This model don’t have good concept of maintenance, the documentation is given very less importance in it. The fixes are done on the basis of quick fix without detailed analysis so there are more chances of ripple effects.

2.10.2 Boehm’s Model

It takes maintenance process as a closed loop cycle. According to Boehm, management decision act as a driving force for a maintenance process. Management determine the changes that should be implemented on the basis of cost benefit evaluation. Managerial decisions are very important in balancing the constraints and objectives. From economic point of view it is an effective model but managerial decisions might put the system into risk because of wrong judgement, biasness and unfair prediction etc.

2.10.3 Osborne’s Model

This model has a difference form all other models that it deals directly with the real maintenance environment. This model allows the maintenance features to be added in the system at each stage if they don’t already exist. According to Osborne, lack of managerial communication and control might cause technical problem in maintenance.

2.11 Maintenance conscious lifecycle Model

Maintenance conscious model is the process model that recognizes the need of “design for maintenance” to instrument the maintainability in the software system from very beginning. A maintenance conscious life cycle is shown in the figure below [2].
2.12 Problems in maintenance

Maintenance is considered to be the hardest part of SDLC and also considered a thankless job. Usually the maintainers are the unsung heroes of the software industry [21]. Different researchers have found different problems in software maintenance. Sasa Dekleva [22] classified the maintenance problems into four categories; maintenance management, organizational environment, personnel factors and system characteristics. According to him the major maintenance problems from the above four categories are as follows

- Changing priority
- Inadequate testing factors
- Performance measurement difficulties
- Insufficient documentation
- Adapting to a rapidly changing business environment
- Large backlog

According to Schneidewind [9]

- It is not an easy job to find out whether the alterations in the program are going to make effective change in the system or not.
- It is hard to find out which part of program’s code is related to specific programming action.
- Another major problem in maintenance is that, if a system is not designed for maintenance then maintenance could not be performed on it.

Negative image associated with maintenance is also a problem, according to the Higgins [23].

“Programmers ...tend to think of program development as a form of puzzle solving and it is reassuring to their ego when they manage to successfully complete a difficult section of code. Software Maintenance on the other hand entails very little new creation and is therefore categorised as dull, unexciting detective work”.

Aforementioned first two problems are related to software comprehension. There are number of other problems in maintenance i.e. maintenance personnel turnover, code duplication, code complexity, ripple effect, insufficient documentation, software aging, lack of tools and graphical user interface available for maintenance etc [24].

2.13 Measure of maintenance

Measurement is an important aspect in general but if we look into maintenance it has great importance. The things should be measured before starting work on them to get best results, if we talk about software maintenance we should have to do measures to see the effective change in system. Software measurement can be defined as

“Software measurement is a process of objectively and empirically quantifying an attribute of a software system and the process connected with its development, use, maintenance and evolution” [2].

The above definition is not only for maintenance but also for development process as well. Measurement can also be seen as a process of mapping an entity to an attribute. For example maintenance team is an entity and their expertise is an attribute. Measures are used to evaluate methods, tools, programmes in order to assure quality. Measures may also be used to instrument a specific level of maintainability, understandability, evolvability and other quality attributes in the software system. There could be several reasons for software measurements that are; evaluation, control, assessment, improvement and prediction [2].

Some code based measures such as size, complexity, understandability and maintainability are as follows.
2.13.1 Size measures
Program size is measured by counting the number of lines of code (LOC) in the program but the blanks and comments are not counted as LOC [2]. There are no restrictions of no. of lines written in a program for functions, classes, modules but still size measures is used as size measure. In maintenance LOC measure is used only to count the number of altered or newly added lines. The LOC measure can be used to estimate the maintenance efforts in terms of staff estimation [12].

2.13.2 Complexity measures
Complexity is defined as a difficulty level in performing an activity. The overall complexity of a program depends upon the individual complexity level of each of: semantic contents, program structure, control flow, data flow, and algorithmic complexity. Dependence on the complexity measure of a single attribute might cause misjudgment which in results affects overall maintenance. There are varieties of complexity measure available like, McCabe’s cyclomatic complexity, Halstead’s difficulty measure, Basili-Hutchens measure etc. The maintainability of a program depends on the complexity value of program, lower the complexity value higher the maintainability and vice-versa [2].

2.13.3 Understandability measures
Ease of program comprehension can be calculated by understandability measures. It includes lines of comments, domain words in software vocabulary etc [2]. Frequent usage of domain words, proper and updated comments can make the understandability easier.

2.13.4 Maintainability measures
The maintainability measures are used to measure the extent up to which the system is maintainable. Software system’s modularity level and mean time to repair (MTTR), degree of coupling and cohesion are commonly used as maintainability measures [2, 12].

2.14 Summary
This chapter gives a detailed account of various aspects regarding to software maintenance. The beginning part of this chapter illustrates the software maintenance process and its types. Five types of software maintenance (corrective, adaptive, perfective, preventive and emergency maintenance) are discussed. Different software system need different kind of maintenance depending upon the nature of the problems or shortcomings they have. Software maintenance is highly time and cost consuming activity which takes 60-80% of total SDLC’s cost. Initial phases of Software maintenance lifecycle are more effort demanding as compared to the later stages while on the other hand in software development lifecycle it is inverse. Selection of appropriate maintenance model has the paramount importance for successful maintenance. Three different kinds of maintenance models (quick-fix model, Boehm’s model, Osborne’s model) are discussed. Use of complexity, size, understandability and maintainability measures and their importance to build maintainable software is also given. The last part of the chapter shows the problems in software maintenance discussed by different authors. Beside holding and pulling the reader towards the aim of the study this chapter also contributes in highlighting the Pigoski’s [12] found conflicted research area “maintainer’s involvement”.
3 Software Evolution

3.1 Introduction
Software maintenance and Software evolution are two different but partially coinciding concepts. Previous chapter has presented a detailed overview about software maintenance. This chapter is about software evolution. It acquaints the readers with evolution in biology and then leads them to software evolution. After that a detailed description of Lehman’s laws of evolution is posed. The importance of software quality attributes, different evolvability test factors, misconception between maintenance and evolution is described in the midsection of this chapter. The last part of the chapter describes the potential challenges in software evolution.

3.2 Evolution
The Merriam-Webster dictionary defines the term evolution as “a process of continuous change from a lower, simpler, or worse to a higher, more complex, or better state” [25]. Before leaping into the pool of software evolution it is important to understand the biological evolution first. Biological evolution is defined as the genetic change in a population from one generation to another while the speed and direction of change is variable. Continuous changes come up with new species and varieties. The species that don’t have the ability to evolve with environmental changes have to undergo extinction [26]. As shown in the figure 3.1 new species [26] are being generated from their ancestor and with the passage of time those who adopt themselves to environmental changes are still there till end and those having lack of the ability to evolve are being disappeared.

Figure 3.1 Species Evolution

The above example also implies to software systems. After the deployment of a software system it must have the ability to undergo change in order to remain useful [1]. This phenomenon of the software system to shape it with change is known as Software Evolution. It can also be defined as:

“The dynamic behavior of programming systems as they are maintained and enhanced over their lifetimes” [27].

Software evolution deals with examining the dynamism in software behavior and the way how they react to change over time [28]. Due to the million dollars investment, software systems have become vital assets for software industry. They want their products to have the ability to change according to the innovative demands of rapidly growing software users and versatile working environment [1].

As the working environment of software is not stable so the real success of a software system is not based on the successes of its one or two releases but it based on the ability of software system to evolve gracefully against the changing requirements [15].
3.2.1 Lehman’s Law of Evolution
Lehman and Belady set the ground for software evolution in 70’s and 80’s; they proposed set of laws for software evolution. There are eight different laws of evolution given by Lehman as follows [1, 15, 29].

- **Continuing change**
The software which is running in a real environment should have the ability to evolve as it is a very basic requirement for software to reside longer in the market. When software is evolved on behalf of environmental changes, it causes new changes in the environment in return. This puts software in a close loop cycle.

- **Increasing complexity**
When a software system experiences change due to several factors, its architecture undergoes erosion and become more complex for future change. Additional evolvability efforts in preventive maintenance are required to avoid or at least prolong the structural decay.

- **Large program evolution**
According to this law enormous changes in a software system might have severe side effects. When a software system goes through large change there are more possibilities of introduction of new problems. Large change in software system might produce new faults that could affect the release time of the software. Another aspect of this law is that the overall evolution is independent of the management decision while rate of change of a system depend upon the managerial/organization decision making process. It is an accepted fact that large organizations are the ones who turn out large systems. Financial planning for each proposed change is done prior to implementing the change. As large changes might uncover new threats and risks which cause delay in release, in meanwhile if change request in a high priority system arises then current system has to be halted.

- **Organizational stability**
This law is opposite to conventional thinking, that rate of development of software is directly proportional to the resource dedicated for system development. According to Lehman’s fourth law; when maintaining an evolving system, large development teams might not be more productive due to some factors i.e. communication overhead, lack of good control and management. So the rate of development or maintenance of software remains constant.

- **Conservation of familiarity**
According to this law, the rate of change in each release of software remains alike. As large changes are always more error prone. So introduction of new functionality in a release might cause more problems and that should be fixed before next release. Major part of effort goes in repairing the faults introduced by the large change in previous release so the overall growth in each new release is not going to exceed a constant limit and remain same.

- **Continuing growth**
This law states that continuous growth in functionality of a system is as essential for user satisfaction as change is necessary for system to exist longer in market.

- **Declining quality**
The quality of a system depends upon its ability to adopt changes offered by operational environment. According to this law, if adoptability of new features does not exist in system then its quality will appear to be decreasing.

- **Feedback systems**
The evolution of a system relies on feedback given by different entities in a continuous way.
3.3 Software Quality Attributes

Quality attributes provide an outline to evaluate the software quality [30]. The quality attributes are used to characterize anticipated quality level and they often define the non-functional aspects of the software system. The overall quality of system depends upon the tight coupling of different attributes [31]. There are different classifications for the quality attributes discussed in the literature. The ISO/IEC 9126 defines six software quality attributes which are functionality, reliability, usability, efficiency, maintainability and portability [30] while McCall described the following quality attributes and named them as quality factors [27].

- **Correctness**
  This attribute defines the degree to which software system meets its intended goals.

- **Reliability**
  This attribute defines the degree to which software system expected to meets its intended goals.

- **Efficiency**
  This attribute defines the degree to which software system proficiently use the available resources to meet its intended goals.

- **Integrity**
  It is the extent to which software system is secure in term of authorization.

- **Usability**
  This attribute defines the degree to which software system is easy to use, interact and operate.

- **Maintainability**
  This attribute defines how easy to locate and fix the error.

- **Testability**
  This attribute defines degree of easiness in ensuring the correctness of software system.

- **Flexibility**
  This attribute defines how easy to modify or evolve a software system.

- **Portability**
  This attribute defines how easy to shift a software system from one hardware/software platform to another.

- **Reusability**
  This attribute defines the extent to which a software system or the components of software system can be used for other applications.

- **Interoperability**
  This attribute defines the capability of a software system to work with other systems without unusual effort.

3.4 Maintenance Vs Evolution

In software engineering maintenance and evolution are two overlapping concepts and often used interchangeably [10, 32], it is relatively difficult to differentiate between these two terms. Some researchers consider maintenance as ‘fine-grained’ and evolution as ‘coarse-grained’ activities as in maintenance we deal with localized changes (corrective, adaptive, perfective) with narrow scope while in evolution we deal with structural changes with broader view (preventive maintenance)[33, 34].

3.5 Evolvability

All the attributes discussed above have their own importance but overall quality depends upon the attributes as whole. We are going to expand evolvability as it is one of the important quality attributes and it might be considered as combination of flexibility and
maintainability. Evolvability may be defined as the software potential to provide easiness to introduce enhancement in it.

According to Liguo Yu et al. evolvability is still not well understood neither in biology nor in software engineering [35]. Similarly it is very difficult to quantify or measure the evolvability of a software system as there is not a standard or direct way to do this [32]. Nasir and Rizwan discussed two ways to calculate the evolvability i.e. structural measures and expert assessments. But both of these techniques are also not completely reliable, each has its own limitations [32].

3.5.1 Software Evolvability test Factors

The evolvability test factors are the characteristics which are used to measure software evolvability; these characteristics are used to make software more usable, maintainable and evolvable. These factors are helpful in instrumenting the evolvability in the system. Much research has been done to investigate these factors. According to Sommerville [1] the evolvability of software system can be assessed by following factors

- Number of requests for corrective maintenance
- Time required to study change cause-effect relationship
- Time required to implement a change

According to David E. Peercy the evolvability of software system depends upon a strong relationship between source code and documentation. The six test factors that are used to measure the evolvability of a software system are listed below [36].

- **Modularity**
  According to this factor a software system should have good logical partitioning with low coupling. Because a software system composed of independent parts (sections, modules) is easy to evolve as the change in one module will have no effect on the rest of the system.

- **Descriptiveness**
  If there exist useful information about the objective, design, logic and implementation of the system then evolvability is going to be easier.

- **Consistency**
  Consistency is also an important factor in measuring the evolvability of a system because consistent use of standards in documentation, flowcharts construction, naming of modules or variables helps in software comprehension which in return improve the software ability to evolve.

- **Simplicity**
  Software complexity is the major hindrance in the comprehension of software system. As higher order languages are easier to understand so the use of low order language i.e. machine, assembly languages may reduce the simplicity. There are many other factors that can affect the simplicity like; dynamic allocation of resource, recursive code, increased number of operators and operands, depth of nested control structures, number of executable statements, statement labels, dead code, code duplication.

- **Expandability**
  Expandability is another evaluation factor which is used to examine the level of expandability of software system. Lack of consideration of some factors might reduce the ability of software to expend properly or easily; the prediction of directions that software can be evolved, inefficient use of data structures, inadequate memory constraints. Detailed discussion about the selection of data structures and memory allocation may help in understanding the developer’s philosophy behind that particular usage, which help in better expendability of software system without extra efforts.
• **Instrumentation**
  Another test factor is instrumentation which refers to an ability to examine or gauge the extent of performance of a product. As in general the quality of documentation is evaluated by the level how well the program is designed to add test aids that can be added to the system later on. The development of the system should be done in such a way that designed instrumentation can be implemented easily [36].

3.5.2 **Structural Measures for Evolvability**
A large amount of research work is done on finding the structural measures for software maintainability and evolvability. Chidamber & Kemerer ’s [37] CK-metrics is very helpful for structural measures but the limitation with CK-metrics is that it only offers services to class level and not to the complete software system level. CK-metrics uses following class level measure to assess the level of evolvability of a class:

- Number of methods in class (WMC)
- Coupling between objects (CBO)
- Number of children (NOC)
- Depth of inheritance tree (DIT)
- Cohesion of methods (LCOM)
- Response set for class (RFC)

Bente Anda [38] described these test factors and they are as follows:

- Choice of Classes
- Design
- Architecture
- Components
- Encapsulation
- Inheritance
- Class Libraries
- Simplicity
- Naming
- Comments
- Technical Platform

3.6 **Evolution Challenges**
There are number of evolution challenges discussed in the literature by different researchers and authors but a brief explanation is given by Tom Mens [39] which is as follows:

3.6.1 **Upholding and amending software quality**
By the passage of time the quality of software decreases due to adaptation of new requirements and it makes the structure of the system more complex. This phenomenon was named as ‘software aging’ by David Parnas [40]. According to Manny Lehman [29] if the software is being evolve, there should be some strategies, tools and techniques that improve the quality of the system if not then it should atleast maintain the quality of software.

3.6.2 **Need of common evolution platform**
Addressing the previous challenge; with respect to long-lasting huge software systems demands highly intelligent tools and techniques. Building of such kind of tool is not an easy task to be done by a single individual or research group. So there is a need of a common platform on which group of researches and individual form all over the world can work and make some tools and techniques.
3.6.3 Requirement of tools for higher order artifacts
According to the literature, majority of the evolution supporting tools are available for low
level software artifacts like source code. So the tool support for higher level artifacts i.e.
software requirement specification, analysis, designs and architecture is another challenge.

3.6.4 Co-evolution between higher and lower artifacts
As there is a greater relationship between higher and lower artifacts, so evolution in any of
the artifact will have impact on other. This give rise to another problem which is a need of
coevolution between lower and higher artifacts, so that they can be evolved in parallel.

3.6.5 Formal support for evolution
Formal methods are the methods and techniques that use the mathematical power for
software specification, verification, validation and development activities [41]. As the user
requirements continuously change so such sort of formal methods are required which may
handle evolving user demands.

3.6.6 Evolution-oriented Languages
The research and development of evolution-oriented languages has also been a challenge in
order to treat the concept of change as essential fact of life. It is similar to object oriented
languages that consider the reusability as first-class feature. Tom Mens et al. [39] discussed
that it is easy to integrate the evolutionary aspects in dynamically typed languages as
compared to statically typed languages. Classical modular development approach is also
very helpful in improving the evolvability of a program but it has some limitation i.e. the
ability of adding or replacing a method in a class that is not defined in that particular
module. Object oriented languages like C++ and Java although support this feature but to
very low extent. On the other hand the languages i.e. Smalltalk and Flavors that support this
type of method addition and replacement philosophy have the impact scope problem. To
solve this problem a concept of classboxes is presented by Alexandre Bergel, St´ephane
Ducasse, and Roel Wuyts [42] that fully support class extensions. These extensions are only
visible to the classbox that defined them but the rest of the system remains unchanged.

3.6.7 Multi-language Support
As the number of languages for different sort of software engineering activities are
increasing day by day i.e. programming languages, modeling languages, specification
languages. So the innovation of set of standard in order to improve the interoperability and
coordination among them is also a challenge for software evolution.

3.6.8 Evolution as fundamental part in Software life cycle
Conventional software life cycle models have less focus on the notion of evolution as a fact
of life. The development of change integrated lifecycle model should be considered to
improve evolvability.

3.6.9 Collection and manipulation of evolution records
Finding evolution records i.e. bug reports, change requests, source code, configuration
information, versioning repositories, error logs, documentation etc and the ways how the
evolution is recorded, can help in improving evolution. The tools and techniques required to
manipulate huge amount of data in well time is another evolution challenge.

3.6.10 Generalized law of software evolution
Michael W. Godfrey [10] raised another issue; which is to generalize laws of software
evolution for open source as well as proprietary software. As it is not considered a good
practice to do experiments on proprietary software due to reason that proprietors of the
software usually don’t want their information to be exposed anyway i.e. through their
software. So it is another evolvability challenge to perform experiments on non proprietary
software and develop generalized laws for both kind of software.
3.6.11 Other Evolution Challenges

Some other evolution challenges that are discussed in literature are as follows:

- Good managerial awareness to foresee, plan, control and manage.
- Development of state-of-the-art tools for software configuration management i.e. recording evolution (version control).
- Conduction of careful empirical study to make evolution better [10, 28] in order to know how process model, tools, languages, people response to evolution and how they can be used appropriately.
- Development of mature predictive model to foresee the potential direction along which system can evolve.
- Consensus on evolution benchmarks and standards to validate the quality of new research in this domain. Michael W. Godfrey termed the same concept as unification of disparate development artifacts to understand systems evolution [10].
- Less importance of software evolution in college/university level software engineering curricula.
- Understanding of biological evolution’s concepts, different type of symbiotic relationship i.e. mutualism, commensalism, parasitism, amensalism, competition and neutralism and investigation to find these type of interdependencies between software families in order to map both biological and software evolution[35].
- Understanding of social, technical and economic environmental pressures.
- Integration of economics and risk into the theories of software evolution [10].

3.7 Software Architectural Selection and Soundness

The importance of software architecture could not be neglected in software development, maintenance and evolution. Software architecture presents overall layout of a software systems, their modules and the way they interact with each other [15]. Architecture of a system deals with both sort of requirements i.e. functional and non functional. Quality of a system comes under non functional requirement and functionality of a system is on the other hand [43].

To come up with a better output, it is important to classify the stakeholder’s expectations into functional and non functional requirements as early as possible before heading towards thorough design. Stakeholder’s can be end users, developers, maintenance engineers, quality assurance engineers, architects, business people etc [43]. As quality attributes are highly dependent to each other and the quality requirement of different stakeholders are also different so it is very difficult to select an architecture which fulfill quality requirements of all stakeholder simultaneously [15, 43]. So selection of suitable software architecture is also a concerning issue to improve the evolvability. This may be helpful in reducing the later efforts. A lot of research studies have been done to find the techniques and framework for software architecture because it is not appropriate to classify architectures in to good or bad. The main problem is the choice of right architecture according to the requirements.

3.7.1 A framework for architecture selection

G. Zayaraz and P. Thambidurai [44] gave a framework to select appropriate architecture and steps involved in right selection are as follows.

- Classification of stakeholders.
- Recognition of quality requirements for each stakeholder.
- Set acceptable scope of each quality attributes for each stakeholder.
- Normalization.
- Identification of the strength of preference.
- Determination of the weight of preference.
- Conversion of the values of quality attribute for the given candidate architectures.
• Calculation of cumulative scores.
• Selection of the architecture.

By reviewing the literature we found the above framework as best for architecture selection and we didn’t discuss it because it is out of scope of our thesis. With the passage of time, as human being get aged or we can say in other sense becomes useless. Likewise when software keep on adopting changes according to user needs the strength of software becomes weak and time comes when it become dead [40]. This phenomenon is known as architecture erosion. So, continuous evaluation is necessary to retain the crispness of architecture.

3.8 Other Activities to Support Evolvability

3.8.1 System Reengineering
Software reengineering deals with the re-documenting, re-implementing, re-structuring or translating the legacy systems into modern programming language to make them more maintainable and evolvable, while the functionality and the architecture remain the same [1]. Chikofsky and Cross named the conventional software development as “forward engineering” and reengineering as reverse engineering. The main difference between the forward and reverse engineering is that forward engineering starts with written specification while reverse engineering or reengineering starts with existing system.

3.8.2 Program Refactoring
Program refactoring, introduced by John Opdyke in the early 1990s as a way to improve the structure of object-oriented programs without affecting their desired external behavior [45].

3.9 Summary
This chapter acquaints the readers with intricate concept of software evolution. Evolution is biologically defined as the genetic change in a population from one generation to another. The emergence of a new software release from the previous one is same as genetic change in a population from one generation to another. Lehman’s laws which are considered as a pioneer work in software evolution are explicated. Software quality attributes are considered to be of paramount importance to build maintainable and evolvable software. Maintenance and evolution are overlapping concepts but the difference is that one is fine-grained while the other is coarse-grained. Evolvability of a software system can be measured by modularity, descriptiveness, consistency, expandability and instrumentation. A detailed description of the challenges in software evolution is also given.

As software comprehension, maintenance and evolution are the three joints of a chain. The need of software comprehension is meaningless without software maintenance and similarly a system cannot evolve without successful maintenance. So this chapter also acts as a milestone to reassure the reader that the right path is being followed. This chapter also sets the ground for the next chapter which is Software comprehension.
4 SOFTWARE COMPREHENSION

4.1 Introduction
Software maintenance process starts with change request and evolution of system rely on success of maintenance process [46]. The first and foremost issue for successful maintenance and evolution is complete understanding of the software system which is going to be maintained. This chapter describes the software comprehension process as a whole to illustrate different comprehension aspects that are helpful for software understandability. But the main focus of the chapter is on code comprehension, problems in code comprehension their solutions and some methods that can be helpful in making it more understandable.

4.2 Software Comprehension
Software comprehension is an elementary and expensive activity in the software maintenance. A large part of software maintenance is software comprehension which utilizes massive amount of time and effort [47]. It is essential to have good understanding of the software system as a whole to make required changes effectively. Software comprehension is a process to know: What a software system really does how it is modularized and how the modules are unified with each other. Which modules may be affected by implementing a change and how this impact can be localized to only that particular module which undergoes change [2]. Studies shows that about half of the total maintenance effort is expended on software comprehension and this expenditure increases if the software under maintenance is developed by some other organization [1, 12].

4.3 Purpose of Software Comprehension
The basic purpose of software comprehension is to get the quick understanding of the system to implement the requested change in such a way that does not disturb the architecture of the software system and does not make hurdle in future evolution [2]. When doing maintenance of a software system it is not possible to make changes without having a complete understanding of system and the interactions within that system [46]. As software has become one of the major business capitals so software companies are striving to expand their revenues and market reach by using the best software engineering methodologies. According to Kristian Raue [48], business life contains both sprint and marathons. But if we look around in software industry there is no concept of sprints. Both software development and maintenance are huge processes and can be considered as marathon. As speed of innovation in market demands is indeterminable and software companies also firmly believe in the fact that “The most costly outlay is time”, so the importance of quick understanding of software system is desirable for effective outcome when maintaining a software system [46].

4.4 Anatomy of Software Comprehension
Comprehension is not only a matter of getting familiar with source code but there are some other important artifacts that should be understood to completely comprehend the software. These artifacts are formal specifications, context diagrams, data flow diagrams, flow charts, source code listing etc. These artifacts are very helpful to understand software architecture and design [2]. No doubt these artifacts are presented and well explained in software documentation but it is seen that documentation is less often co-evolved with the source code. In many cases either the documentation is not available or insufficient or outdated. So in this chapter our main focus is on source code and on those entities that are helpful in understanding the source code.
4.5 Information Required for Comprehension Support
For better software comprehension it is essential to have detailed information about certain software aspects such as problem domain, execution effect, cause-effect relationship, product-environment relationship and decision support features [2].

4.5.1 Problem domain
Problem is defined as “A state of difficulty that needs to be resolved” while the domain is that particular area which is facing the problem. Examples of problem domain are health care, finance, education etc. The information about problem domain is as necessary as the problem description itself is. Good knowledge of problem domain is very important in the process of software maintenance and evolution. It helps the management team in selecting appropriate resources i.e. expertise skills, algorithms, tools, techniques.

4.5.2 Execution effect
Execution effect is an abstract level representation of the system’s behavior for a specific set of inputs. It is categorized into two levels of abstractions i.e. high level and low level. In high level abstraction, the overall results of the systems are considered while in low level abstraction the contribution of individual module is observed for specific behavior. The purpose of execution effect is to verify whether or not the requested change is implemented and expected outcomes are achieved. For the fulfillment of this activity, control and data flow has to be considered.

4.5.3 Cause-effect relationship
The purpose of cause-effect relationship is to find out the effects of intended change on other parts of system, it also helps in predicting potential ripple effects. Another major contribution of cause-effect relationship is to find out the nature of dependencies in a system, because whether the change is small or large the dependencies always produce many difficulties and errors.

4.5.4 Product-environment relationship
The purpose of product-environment relationship is to find out all the external factors that could have affect on a system. The factors those might influence a system could be business rules, government regulations, software/hardware operating platform.

4.5.5 Decision support features
These are the features that invigorate the technical and managerial decision power of maintenance team while making decisions about option analysis, budgeting and resource allocations. These features involve complexity, maintainability.

4.6 Comprehension Challenges

4.6.1 Stupid use of high intelligence (Tricky Code)
It is observed that programmers write tricky code which is very hard to understand by other person, which later on causes problems in comprehension. There could be many intentions behind writing tricky code that are; to show intelligence or smartness, job security etc [49].

4.6.2 Different programming Styles
Programming styles vary from company to company and programmer to programmer. Growing dependency on software systems has made them larger in size. Sometimes it is not possible for a programmer or a small group of programmers to write a large program. Due to higher development cost of the local service providers, software development companies also use offshore software development where programmers globally participate in software development [50]. So a large software program is co-written by a group of programmers having different programming skill and experience. In result the developed product have a combination of programming styles followed. These sorts of programs are very difficult to understand [51].
4.6.3 **Poor Naming convention**

Though the program comprehension is not only limited to the naming of identifier but it is also not possible to refuse their importance. The use of meaningless, inconsistent and poor naming conventions may be a heavy burden or source of lack of comprehension [52]. Similarly meaningful, descriptive and self-documenting names are also insufficient but the main concern is that a name should also have the ability to define the concept clearly.

4.6.4 **Lack of Domain words in Software Vocabulary**

In software maintenance, problem domain information is very important to have abstract level understanding [2]. Use of domain words in source code, documentation and user manuals is considered a good practice [53]. Sonia Haiduc and Andrian Marcus [53] highlighted the fact that the good developer usually try to embed this domain information in identifiers and comments because in case of unavailability of sufficient documentation it is very helpful in improving the code understandability. Similarly in case of system systems that has been developed by some other organization, the maintenance team also start the comprehension process by keyword search to locate the concerning concept. So the lack of domain words in software vocabulary is another problem in comprehension.

4.6.5 **Conflict on Software Vocabulary**

Through empirical studies it is proven that word choice is another conflicting issue leading to misconception [52]. There is a rare chance that two people use the same word for a certain concept. Moreover, there are different words that can describe the certain concept. Negligence of this issue by developers may also severely compromise the comprehension. The comprehension of program identifier depends upon their ability to describe the domain concept. So the wrong selection of descriptive terms corresponding to domain concept is another hurdle in comprehension [53].

4.6.6 **Program representation**

Complex kind of control flow and unordered block of code make the comprehension more complex. Unobvious or unclear dependencies make it more difficult for programmer to understand code [54]. Grouping related statements and modules might smoother the process of comprehension. Good visual layout makes the code more understandable and modifiable in the absence of author. Different layout tools can be used like whitespace, blank lines, alignment, indentation etc.

4.6.7 **Insufficient comments**

It is a known fact that the lack of comments and the outdated-comments as well, are the major causes to increase maintenance cost and reduce the comprehension. According to Ryan Campbell [55]

> “Commenting your code is like cleaning your bathroom—you never want to do it, but it really does create a more pleasant experience for you and your guests”.

The aforementioned statement is true as the importance of comments is well accepted but it is often neglected. There are some problems regarding to commenting code which are hard to cope. A study showed that [56], it is often seen that comments and source code ratio does not remain stable in source code evolution. So, there should be a process to make co-evolvement of source code and comments to make comments updated. It will maximize the benefit for understanding.

4.6.8 **Dangerously deep-nesting/Deep-Inheritance tree (DIT)**

Undue usage of deep-nesting is one of the major causes in making code comprehension unmanageable. According to the author [54], there are just a few people who are able to understand deep-nesting which goes more than three levels. Similarly, depth of inheritance tree could also affect understandability [57]. For example, if classes are deeply inherited in a
program they keep more methods to inherit, which give rise to complexity hence it is another reason of making the understandably harder.

4.6.9 Concept location
A change request may consist of a problem statement or an innovative idea by the user. After the change request understanding the next phase is to locate the specific part of code which is responsible for the problem or where the change should be implemented to get the required result. Identification of that particular part of code is known as concept/concern location [58]. Concept location is another challenging task in software comprehension as in case of a large complex program it is neither possible nor cost effective way to read all the code from the scratch. In ideal case the developer would be well aware of code or the traceability link for concept location might be provided by the documentation. But worst case is very common where the maintainer mostly have to use the related queries to find the desired part of the code.

The difference between natural language description of change request and program vocabulary for its implementation is another hindrance in concept location. As the mapping of natural language with the program vocabulary requires the detailed knowledge of problem domain, programming techniques and idioms etc [59].

4.6.10 Code duplication
Code duplication is another common problem that severely complicates the software comprehension process [60]. No doubt writing the code from the beginning is very hard as compared to code duplication but software having that problem is often considered bad [61]. There could be several reasons for code duplication like programmers feel it easier and more reliable to use a pretested code fragment. Some time working environment where line of code is used as a measure for programmer’s performance might also motivate him/her to copy a code segment [60]. Code duplication pushes the entire burden towards the activities that may be performed later on with the software i.e. maintenance and evolution. Code duplication makes the maintenance and evolution more difficult and expensive. Fowler and Beck [62] referred it to “bad smells in code” and Richard Wettel, Radu Marinescu [61] are also agree with them.

Although, several tools have been designed to detect code duplication but the code after being copied it seldom seems similar to the original code. Reid Kerr highlighted [63] another noteworthy issue context-sensitiveness in code duplication. According to him as a segment of text has a strong relation with its prior and later blocks of text and copying that particular segment to another location might break the flow or cause error due to the difference in context. Just like that copying a block of code to another location may cause failure to correctly adapt to a new context [63].

4.6.11 Identification of dead code
The Legacy systems that has been maintained and evolved by different programmers for several times mostly contain noteworthy amount of dead code [64]. The dead code might contain some methods and classes that are no more usable, but identification of such classes and methods is not an easy task. Moreover, when a legacy system undergoes maintenance the dead code become one of the major problems in understanding code.

4.7 Factor that can improve code understandability
There are a number of factors that can be helpful in improving the accuracy, correctness, completeness and the ease with which the program can be understood. These factors are as follows:
4.7.1 Expertise
It is proven from many empirical studies that expertise matters a lot in quick and accurate understandability building. According to Andrew Jensen Ko and Bob Uttl [65] individual difference is a major factor in selecting the inappropriate program comprehension strategies in unfamiliar programming systems. Well grounded knowledge of problem domain, problem solving and programming languages make the program comprehension easier [2].

4.7.2 Syntax highlighting
Syntax highlighting is another supporting factor that provides help in improving the code familiarity. It is commonly used to highlight different code constructs by using different colors and fonts for different identifiers [46]. Syntax highlighting support is often provided by the compilers or the integrated development environments. Although it is a very basic approach but no doubt it contributes a lot in improving understandability. Syntax highlighting may be used with combination of other strategies to improve the concept/concern location techniques.

4.7.3 Cross referencing
Cross referencing is used as a tool for general understandability building about the source code. During the software comprehension activities the maintainer often has to jump to and fro to the definition and use of different program identifiers. Cross reference support is used to provide the linkage between identifier’s definition and its use at different places in the source code [46].

The information gathered through cross referencing can be used to draw different code overview diagrams, which to some extent can help to determine the dependencies between different source files [46]. As nowadays many of the software systems are developed in different languages, so cross referencing support between code segments written in different languages can make the understandability of code easier [66].

4.7.4 Call graphs
The Call graphs are used to represent inter procedural communication in a source code. It also describes the information about the data used within a procedure and global data shared by different procedure [67]. Two commonly used types of call graphs are static and dynamic call graphs. Static call graphs show the potential calls while the dynamic call graphs show the calls that can occur on a particular execution of a program [46]. Static and dynamic call graphs may be used in static and dynamic program analysis.

4.7.5 Comments and Annotations
Traditional comments in the source code are usually helpful for the code comprehension. But often the comments are outdated or incomplete to show the proper meaning of the source code. Annotation is another way to help the maintainer to comprehend the code easily. They are virtually added comments that originally don’t exist in the code but often more helpful. Annotations are different from traditional comments in the sense that they don’t require write access to add them in source code. Another advantage of annotation is that programmers can add as many comments as they want without flooding the code [46].

4.7.6 Program Slicing
Understanding a large program consisting of thousands of code is not an easy job. To make it a bit easier is program slicing is used. Program slicing was originally introduced by Weiser M., is an effective technique for narrowing down the concentration on the responsible part of the program [46, 68]. This technique is used in finding the cause-effect relationship. It makes it easier in finding the cause of the error by slicing down the program in small related pieces.

According to [68] up till now many types of slicing have been proposed i.e. forward slicing & backward slicing, static slicing & dynamic slicing, intra-procedural slicing & inter-procedural slicing, sequence program slicing & current program slicing, object-oriented program slicing, aspect-oriented program slicing, conditioned program slicing, relevant
slicing, union slicing, hybrid slicing, amorphous slicing, denotation slicing, chopping, dicing, specification slicing and so on. The selection of appropriate slicing technique according to the nature of the problem is not an easy task but it could help in better results [46, 68].

4.7.7 Ripple Analysis
Ripple Analysis is a comprehension activity to locate that part of code that might be affected by implementing a certain change. Ripple Analysis is also known as forward program slicing as it is the process to know the which part of the code could be affected by a change while in common/backward program slicing the focus is on the sources that could have affected the value of a variable. Ripple analysis is also helpful in finding the extent and nature of dependencies among different code segments [46].

4.7.8 Program Decomposition
In the understandability of program, complexity is the major problem. To minimize that problem, program decomposition is the remedy to some degree. Modular decomposition is a technique that is used to break down large program into significant small modules. It is easier to comprehend small modules instead of large program. Program should be modularized in such a way that each module should have a central purpose i.e. high cohesion. Module should be designed that other modules don’t have to meddle with its internal data. Loose coupling between modules may also be helpful in localizing the effect of implemented change in maintenance process [2, 54].

4.8 Summary
This chapter shed light on software comprehension process and is of great importance in heading towards the goals of our study. As comprehension is the main concern behind software maintenance and evolution, this chapter acquaints the readers with software comprehension process and its purpose.
A detailed description of information required for software comprehension is also given. This chapter also reports the problems in code comprehension found through literature review and the factors that can be helpful in improving the code understandability. This chapter also sets the ground for the empirical study to find the problems faced in industry to verify the problems found through literature review.
5 SURVEY DESIGN

5.1 Introduction
As the name indicates, this chapter is about survey design. The description about the purpose of survey is followed by the rationale behind choosing survey as a strategy for data collection. The way how we reached the respondents is also presented in this chapter. Another main issue regarding to survey is ethical considerations, which are also specified in this chapter. The relation between survey questionnaire and research questions is also presented to show their validity.

5.2 Purpose of the Survey
A survey is a strategy of inquiry for quantitative research method which provides a numerical description of the trends, attitudes and opinions of a target population. Researchers study a selected sample of population and then generalize the results for whole population [3].
In our study we conducted an industrial survey. The purpose of the survey was to get the industrial point of view about software maintenance and evolution. To explore the problems they face pertaining to the software comprehension and their overall impact on the software maintenance and evolution process. To get an idea about the methodologies they adopt to make the software more maintainable and evolvable in future. Another purpose of this study was also to validate our findings through literature review, from industrial evidences and to get the answers of our research questions by technical experts in the industry.

5.3 Why used this Strategy of Inquiry?
We preferred to use this strategy of inquiry because of it is widely acceptance for data collection. Another reason to use survey was economy of design and rapid turnaround in data collection [3]. Due to the limited time and resource constraint we used survey as a data collection procedure.

5.4 Form of Data collection
Different forms of survey are available for data collection. According to the survey may be in written, oral or electronic form. Self- administered questionnaires, interviews, structured record reviews and structured observations are also used as a form of data collection in survey [3].
We used an online software surveygizmo [72] to create a web based survey. We designed a webpage for own questionnaire and administered it online. Online survey is helpful both for administrators and respondents due to its convenience, availability and cost. We also sent survey MSWord document to some respondents.

5.5 The Population and sample
The focus of our survey was local and international software industry providing both development and maintenance services. We sent our questionnaire to 50 different respondents working in software industry globally. We used our own contacts to identify the related individuals for survey. Individuals having good knowledge and reasonable practical experience of software design, development and maintenance were selected. First we sent them an invitation email with cover letter to inform and then sent them the survey on their approval.
5.6 Ethical Consideration
As we used electronic form of survey for data collection we considered the following ethical issues for the survey [69].

- Fair Sample representation
  Selection of appropriate population sample is very important to get the fair and unbiased responses. Faulty sampling is one of the responsible factors for external validity threats. Appropriate population sample should be consists of individuals of gender, race and different geographical background. But use of online survey might be a risk when population sample is of aforementioned diverse nature due to the limited access of computer resources to different individuals. But as in this study our purpose was to find out the major code comprehension problems that a maintenance team has to face in software maintenance and evolution process. In this study our target group was the maintenance personnel. So there is no risk of unavailability of computer resources. To avoid or mitigate the biasness we selected only those participants for the survey having good knowledge and reasonable practical experience of software design, development and maintenance. Similarly the size of the sample is also very important to get the significant results. So in this study we sent our survey to 50 different respondents. Although it is not too big to generalize the results but due to the limited resources we think that it was a reliable sample size for our study.

- Careful Data Analysis
  In this study we performed the careful data analysis of the responses we got. We did not consider the partially submitted answers because those might lead us to draw the wrong inferences. But the email spoofing might be a threat to external validity of the study.

- Respondent’s Anonymity
  As according to [69] “an electronic response is never truly anonymous, since researchers know the respondents’ e-mail addresses”. In this study we guaranteed our respondents their complete anonymity, and promised them not to share any information that identifies them with anyone outside our research group.

- Responsible quotations
  As in this study we paraphrased the suggestions to improve the code comprehension from different maintenance experts, so we tried to correct typographical or grammatical errors before quoting our respondents since they don’t have the ability to edit their responses after final submission of the survey. We tried to report their casual language responses in formal language.

5.7 Questionnaire Building
Writers started the questionnaire building phase with the planned brainstorming sessions. Literature review and some previous surveys [70] also helped a lot in selecting the suitable questions for questionnaire that were highly relevant to our research area. Planned pilot survey also made it possible for us to build an appropriate questionnaire.

5.8 Survey Instrument
We used a self designed questionnaire as an instrument for data collection in the study. The questionnaire contains both open and closed ended questions. The questionnaire was categorized into three different sections which were: the information section, the open-ended questions section and closed-ended questions section. The brief mapping between survey questionnaire and research question is given below to discuss the contribution of each question. The detailed questionnaire is presented in the Appendix section (8.2).
<table>
<thead>
<tr>
<th>Survey questionnaire</th>
<th>Explanation</th>
<th>Supported research question</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Information Section</strong></td>
<td></td>
<td>N/A</td>
</tr>
<tr>
<td>Personal Information</td>
<td>The purpose of information section was to validate the respondent and categorize the answers to analyze them in a proper way.</td>
<td></td>
</tr>
<tr>
<td>Name, Designation, Company, Working domain, Maintenance experience, email etc.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Open-ended questions Section</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Does the type of maintenance performed (Corrective, Adaptive, Perfective, and Preventive) influence the approach or methodology adopted?</td>
<td>The purpose of this question was to know whether or not maintenance personnel are well aware of the different types of maintenance activities because each type requires different level of information for software comprehension.</td>
<td>RQ-1: What are the problems that a maintenance team has to face in software comprehension and their effects on overall maintenance and evolution Process?</td>
</tr>
<tr>
<td>What are the three major problems that you face in the maintenance process?</td>
<td>There were two purposes behind this question: (a) To find out the major problems that maintainers face in maintenance process. (b) To find out where the software comprehension’s problems lie among other software maintenance problems.</td>
<td></td>
</tr>
<tr>
<td>What are the three major problems that you face in code comprehension?</td>
<td>To find out the major problems that maintainers face in code comprehension process and to confirm that, are the problems found in literature same to the problems that maintenance personnel have in real environment.</td>
<td>RQ-1: What are the problems that a maintenance team has to face in software comprehension and their effects on overall maintenance and evolution Process?</td>
</tr>
<tr>
<td>Describe the approaches or</td>
<td>To find whether or not</td>
<td></td>
</tr>
<tr>
<td>methodologies you use in code comprehension?</td>
<td>they use suitable methodologies and approaches.</td>
<td></td>
</tr>
<tr>
<td>Do you use any tool(s) for code comprehension in the maintenance process? Yes/no: If yes, name it/them?</td>
<td>To know whether or not they use any tool and if they use then the degree of usage.</td>
<td></td>
</tr>
<tr>
<td>How can code comprehension be improved?</td>
<td>To get any new idea from maintenance experts that how code comprehension can be improved.</td>
<td></td>
</tr>
<tr>
<td>Do you think code comprehension can be improved in a system which is currently under maintenance? Yes/no: If yes, in what way?</td>
<td>To know is it possible to code comprehension can be improved by using design techniques in maintenance process.</td>
<td></td>
</tr>
<tr>
<td>RQ-1: What are the problems that a maintenance team has to face in software comprehension and their effects on overall maintenance and evolution Process?</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Closed-ended questions Section**

| How would you compare software maintenance and development, which provides more of the following, or there is no difference? (challenge, responsibility, carrier growth, salary, Learning opportunity) | The purpose of this question was to do confirmation about the negative concept of maintenance in the literature. It is often considered that development is more demanding task than maintenance. |
| How would you compare software maintenance and development, which requires more of the following, or is there no difference? (Experience, Design Skills, programming skills, problem solving skills, foreseeing/prediction skills,) | Same as above |
| Which of the following is more time consuming activity in software maintenance? [write down an answer from 1(Low) to 6(High)] (Comprehension(understandability), Analysis, design, Implementation, Testing, Documentation) | The purpose of this question was to know which one of the asked activities is considered on top while doing maintenance. |
| | |
| | | Indirect relation with RQ-1 |
Which of the following do you think require(s) more (time and cost) when you are maintaining a system that has been developed by your organization?  
Change request understanding  
Code comprehension  
Change Implementation  
Cause-effect relationship

The purpose of this question was to know the difficulty level of different activities in maintenance when the system under maintenance has been developed by the same organization.

RQ-1: What are the problems that a maintenance team has to face in software comprehension and their effects on overall maintenance and evolution Process?

Which of the following do you think requires more (time and cost) when maintaining a system which is developed by some other organization?  
Change request understanding  
Code comprehension  
Change Implementation  
Cause-effect relationship

The purpose of this question was to know the difficulty level of different activities in maintenance when the system under maintenance has been developed by the some other organization.

RQ-1: What are the problems that a maintenance team has to face in software comprehension and their effects on overall maintenance and evolution Process?

Which of the following measures and metrics do you use to improve code understandability?  
Complexity measures  
Size measures  
Descriptiveness measures  
Consistency measures  
Modularity measures

The rationale behind this question was to identify the industry adopted methodologies for code comprehension.

RQ-1: What are the problems that a maintenance team has to face in software comprehension and their effects on overall maintenance and evolution Process?

To what extent it is beneficial to properly document each change request?

The purpose of this question was to find out to which extent documentation is beneficial for software comprehension.

Indirect relation with both RQ-1 and RQ-2

To what extent it is beneficial to properly document every implemented change

The purpose of this question was to find out to which extent documentation is beneficial for software comprehension.

Indirect relation with both RQ-1 and RQ-2

<table>
<thead>
<tr>
<th>Table 5.1 Mapping of survey questionnaire and research questions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

5.9 **Survey Pilot**

Before sending the survey to our intended respondents we conducted a pilot/trial survey with BTH software engineering students to identify the design problems in the survey and ambiguities in questions. It helped us to remove the questions that were not very relevant to our study and to improve the understandability of our questionnaire. Pilot survey made it possible for us to know that average required time to complete the survey was 30 minutes.
After deep analysis of their suggestions we improved our questionnaire and also shortened it. After the final approval by our supervisor we sent the survey to our intended respondents.

5.10 Survey Execution

In the very beginning, after the selection of survey as a strategy of inquiry for our research study we sent invitation to our intended respondent (suggested by our contacts) to get their consent and prepare them mentally for this study. We also sent them a cover letter to let them know the purpose of our study.

After their confirmation to participate in the study we sent them the questionnaire. To avoid any kind of bias we did not brief them about the contents of the questionnaire. Due to limited time and resources, we decided 10 days execution period for the survey. During the execution period we constantly monitored the surveygizmo [71] to see the responses. After the decided execution time period we removed the link of the survey from surveygizmo and started the analysis.
6 RESULTS AND ANALYSIS

6.1 Introduction
The purpose of our survey was to identify the problems regarding to the code comprehension in software maintenance and evolution process and find the suggestions to improve the comprehension process. This chapter presents the results of survey and answers the research questions set for the study. For understandability the results are presented in the form of graphs with explanation. Results validity is ensured by considering the different validity threats discussed in chapter 1 section 1.6. The comparison of results from survey and literature study is also presented. The last part of this chapter also discusses the limitation of the study.

6.2 Response Rate

![Response rate chart]

Figure 6.1 Survey response rate

This section shows the response rate of the survey. The survey was distributed to 50 different respondents of which 28 % (14 out of 50) fully cooperated and submitted the complete answers of the survey. Some respondents 18% (9 out of 50) partially submitted the survey while 54% (27 out of 50) didn’t give any response. We didn’t consider partially submitted answers so they lie in same category as nonparticipants do.

The respondents, who partially submitted their responses, were contacted to know whether or not they will complete the survey and they assured us to complete it. We had very short time of 10 days for the survey execution. However, we didn’t get their responses till the last day. We didn’t consider the partially submitted answers, as we were not quite sure the reasons for they didn’t complete the survey. The reasons might be their time limitation or they might want to change their answers etc. These reasons could lead us to draw wrong inferences. Hence, we put them with non participants. The response rate shown above in the graph is low, but limited timing constraint didn’t allow us to go back and follow the reasons for the low response rate.
### 6.3 Experiences in Maintenance

![Figure 6.2 Respondent’s maintenance experience](image1)

The above graph shows the experience of the respondents in software maintenance. We found the respondents with different experiences from 1-5 years, 2 out of 14 were 1 year experienced, 4 out of 14 were 2 years experienced, 7 out of 14 were 3 years experienced and 1 out of 14 was 5 years experienced. The logic behind asking the experience from the respondents was to verify the validity and the credibility of the results inferred from their responses. The graph also shows that there are only few people with more experience (5 years) in maintenance. This might be due to the lack of career growth in software maintenance as compared to software development.

### 6.4 Maintenance Team Management

![Figure 6.3 Maintenance team management](image2)

According to the survey response 71.4% (10 out of 14) of respondents reported that same team is used for both maintenance and development while 28.6% (4 out of 14) said that
different teams are used for maintenance and development. Our findings validate the results of Stephen W.L. Yip [70].

The above shown graph is the illustration of question number 1 of the survey questionnaire. The purpose of this question was to verify the involvement of maintainers in the development phase of SDLC. Hence, from the study it is proved that in most cases the same people act as maintainers and developers which is an ideal case for software maintenance. The answer of this question narrowed down our focus to other problems rather than on the maintainer’s involvement. But there are some important aspects that are still unclear i.e. the extent to which the maintainers are involved in the pre-development phase. No doubt maintainer’s involvement in development phase contributes to improve the maintainability of the product under development but their involvement in pre-development phase (software design phase) can contribute much more. Similarly employee turnover might be a problem in above case and software comprehension can also be a problem when the system under maintenance is developed by some other organization.

### 6.5 Maintenance Vs Development

![Maintenance vs. development](image)

The above graph depicts the responses of question number 9 of the survey questionnaire. The question used behind the above graph was taken from Stephen W.L. Yip’s survey [70] about ‘software maintenance in Hong Kong’. The purpose of this question was to compare the software maintenance and development activities and the associated benefits with them.

The values mentioned on the bars show the number of responses for each category. According to our survey results it is clear that salary and career growth are low in software maintenance as compared to software development while on the other hand challenge and responsibility seem bit more. Hence, the results of our survey validate the statement of H. Y. T. Matt [21] i.e. maintenance is harder and less rewarding job as compared to software development.

But our results also negate Stephen W.L. Yip’s [70] findings that high staff turnover is due to lack of challenges and responsibility in maintenance work. According to our findings high turnover in maintenance work might be due to the highly challenging nature of maintenance job and lack of incentives and growth opportunities in return.
6.6 Problems in Software Maintenance

![Bar chart showing the distribution of problems in software maintenance](chart.png)

Figure 6.5 Problems in software maintenance

The above graph shows the responses of question number 4 of the survey questionnaire. The problems in software maintenance process found through the literature review are already discussed in chapter 2 (section 2.12). In response to the open-ended question related to software maintenance problems we received 10 different types of problems. We classified these ten types of problems highlighted by the respondents in three categories i.e. management related problems, user related problems and system related problems. Our classification of maintenance problems is somehow similar to Sasa Dekleva’s [22] classification of maintenance problems (Appendix C).

Most of the respondent ranked system related problems as the most crucial type in maintenance and evolution process. Majority (60%) of the system related problems were about code comprehension.
6.7 Problems in Code Comprehension

The above graph represents the responses of question number 7 of the survey questionnaire. Graph shows the problems in reported by respondents in code comprehension alphabetically. 10 out of 14 respondents are agreeing that the major problem they face in code comprehension is the poor program representation. Insufficient comments are considered to be the second major problem in code comprehension as 8 out of 14 respondents reported this figure. Rest of problems and their ranking is shown in the above graph with the percentage of respondents on the top of each bar. Complex control flow, unordered block of code, ambiguous or unclear dependencies, poor visual layout, alignment and indentation etc are the hindrance in program representation.

The respondents reported almost the same problems in code comprehension that are found through the literature review and already discussed in chapter 4 (section 4.6). The survey results verify the existence of these problems in the industry and their severity level is found through the number of responses, which is mentioned in the graph on each bar. Greater the value on the bar greater the severity level of the problem is.
6.8 Suggestions to improve code comprehension

This section deals with responses to the question number 8 in the survey questionnaire. It was an open-ended question to find out effective suggestions from the maintenance experts. Different professionals gave different hints for making program comprehension easier; we paraphrased them in the following table. The suggestions are sorted out according to the severity level of the problem behind them.

<table>
<thead>
<tr>
<th>No.</th>
<th>Problem</th>
<th>Suggestions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Program representation</td>
<td>The source code should be ordered hierarchically. Inter and intra modular dependencies should be clear. Code should be well aligned and indented. Good visual layout of the code makes it easier to understand.</td>
</tr>
<tr>
<td>2</td>
<td>Insufficient comments</td>
<td>Comments should be as much as they can clearly describe the logic and meaning of the code.</td>
</tr>
<tr>
<td>3</td>
<td>Poor naming</td>
<td>Identifiers Should be named in consistent way. Identifiers should be meaningful and descriptive.</td>
</tr>
<tr>
<td>4</td>
<td>Modularity</td>
<td>Program should be decomposed in possible independent parts. Module should be loosely coupled so that the effect of an implemented change can be localized. Layered architecture should be used.</td>
</tr>
<tr>
<td>5</td>
<td>Deep nesting /DIT</td>
<td>Avoid deep nesting.</td>
</tr>
<tr>
<td>6</td>
<td>Stupid use of high intelligence</td>
<td>Write the code in simplest possible form. Avoid the use of complex algorithms. Code should be written in a way so that other can understand easily. Write from top to bottom.</td>
</tr>
<tr>
<td>7</td>
<td>Code Duplication</td>
<td>Programmers should be motivated to avoid excessive use of code duplication. Programmer should be well aware of the difference between code duplication and reuse.</td>
</tr>
<tr>
<td>8</td>
<td>Dead code</td>
<td>After every maintenance activity all methods, classes or modules that are no more useable should be identified and removed.</td>
</tr>
</tbody>
</table>

Table 6.1 Suggestions to improve code comprehension
6.9 Discussion on survey

Majority of the results found through the survey were the same as found in literature review while the differences between the survey results and the literature review is discussed below:

- Respondents highlighted the need of sufficient comments but not a single one argued about the need of co-evolvement between source code and comments (discussed in section 4.6.7).
- Similarly we could not find any suggestion about the use of annotation as it is considered to be more helpful for better comprehension (discussed in section 4.7.5).
- Majority of the respondents suggested that identifier should be self descriptive but no one discussed the need of mapping between identifiers and domain concepts (discussed in 4.6.4).
- Respondents were aware of the problems due to code duplication but no one described the violation of contextual relationship due to insensitive cut, copy and paste (discussed in 4.6.10).
- Depth of inheritance tree and deep nesting is considered a problem in code comprehension but we could not find any suggestion about the level of nesting and inheritance (discussed in section 4.6.8).

6.10 Discussion and Answers of research question

In this research study, we found from the literature review that maintainer’s non involvement in the process of development could be a problem in the software maintenance; as software maintenance is a difficult task which requires a lot of expertise and thorough knowledge of the structure of software. In the first phase of our survey we tried to find out the whether or not the maintainers are involved in the process of software development. As from the survey results (section 6.4) we found that 71.4% of the total population reported that they use same team for the development and maintenance which shows that maximum maintainers are involved in the process of development as same teams are being used for both processes. Hence our findings validated the results of Stephen W.L Yip [70], and as we found maintainer as somehow involved in the process of software maintenance it narrowed down our focus to other problems of software maintenance. However the extent to which the involvement of maintainers could help more for the software maintenance is still unclear.

As aforementioned (regarding to section 6.5) software maintenance require a lot of expertise but we found there are less experienced people in it as compared to software development due to high turnover. According to Stephen W.L. Yip’s [70] high turnover in the software maintenance is due to less challenging and responsibility but in the survey we found its inverse. Challenge and responsibility is high in software maintenance as compared to software development and on the other hand salary and carrier growth is less so it is the reason for high turnover. Hence, the results of our survey validate the statement of H. Y. T. Matt [21] i.e. maintenance is harder and less rewarding job as compared to software development and negate Stephen’s point of view. So if the companies provide appropriate salary and show carrier enhancement to their employees in the field of software maintenance, high turnover could be minimized and it might keep the employees stick to this field.

Regarding to section number 6.6 we tried to investigate current problems in software maintenance and along with other problems we found the system related problems to the high extent i.e. code comprehension problems. This investigation could help more we could find any different problem in the industry than literature but almost the same problems were found from the industry. Form the literature we put the problems together from different sources and validated them form industry and then we prioritize them according to their severity level. The problems were poor program representation, insufficient comments, poor
naming, bad program decomposition/modularization, excessively deep nesting/inheritance, stupid use of high intelligence, code duplication and dead code. Unless the code is well comprehended the change implementation is impossible. Moreover, these problems might also lead the maintenance team to introduce new problems besides resolving the ongoing problem(s). The severity level of the problems of code comprehension is not available in literature and we prioritize these problems from our survey according to their severity level.

We tried to find out suggestion from the maintenance experts to improve the code comprehension. We found many suggestions and they are discussed in earlier section (6.8) and there was a gap between industry and literature which is also discussed in detail in section (6.9). Hence that gap might affect the maintainer directly to code comprehension and indirectly to software maintenance. As more than half of the software maintenance time spends in code comprehension, the gap should be minimized to get quick and easy understanding of software. This could minimize total cost of software maintenance.

RQ-1: What are the problems that a maintenance team has to face in code comprehension and their effects on overall maintenance and evolution Process?

Answer: In this research study, the careful literature review and survey revealed eight different problems that are the major hindrance in code comprehension. The severity level of these problems is found through the survey and they are prioritized according to their severity level. The problems are poor program representation, insufficient comments, poor naming, bad program decomposition/modularization, excessively deep nesting/inheritance, stupid use of high intelligence, code duplication and dead code. Unless the code is well comprehended the change implementation is impossible. Moreover, these problems might also lead the maintenance team to introduce new problems besides resolving the ongoing problem(s). In result the quality of the product under maintenance is terribly compromised and faulty software evolves. Hence these comprehension problems have great impact on overall software maintenance and evolution process.

RQ-2: How does the good understanding of source code contribute in speeding up and improving the software maintenance and evolution process?

Answer: According to the literature review and the survey conducted in this study, we found the code comprehension as a most time taking activity in overall software maintenance and evolution. Good code comprehension helps the maintainers not only in concept location but also in ripple analysis after the change implementation. So in this way the good program or source code understanding can improve the maintenance and evolution.

6.11 Study limitations
The study is carried out for the code comprehension in the context of software maintenance and evolution improvement. The validity threats to the study have already been discussed in chapter number 1(section 1.6). There are some limitations of the study that are being addressed in this section.

- As the study results are based on the limited number of responses so they are not enough for broad generalization. The findings can be refined and made more robust by replicating the study on large sample of population.
- As in this study we did not discussed the tool support available for the code comprehension so further empirical studies are needed to find the pros and cons of the existing tool support.
6.12 Conclusion and Future work

This thesis is carried out to find problems in program comprehension that might affect the maintainability and the evolvability of software system. The suggestions found in this study can be used as a framework for writing comprehensive programs which are easy to maintain and evolve.

Study also shows that there is an apparent gap between the methodologies discussed in literature and practices in industry. Adaptation of these methodologies can improve the understandability of product being developed and decrease the time consumption in later maintenance and evolution.

The results presented here are based on small sized sample so there is a need to replicate the same study on large sized sample. Similarly the available tool support and their shortcomings should also be considered to make comprehension process easier.

During the study we found some other important aspects that should be considered. They are as follows:

There is enormous scope for empirical studies to know the level of maintainer’s involvement in the pre-development phase of software. Consensus building on programming standards and on software vocabulary selection can open the new avenues for software evolution. Moreover, there should also be generalized tools for code comprehension of both proprietary and not proprietary software. In academic level some software maintenance courses should be taught as students are often taught software development courses but not the courses on software maintenance.
7 References

What is evolution?


8 APPENDIXES

8.1 Appendix A: Survey Cover letter

Blekinge Institute of Technology
Department of Software Engineering and Computer Science
Ronneby, Sweden

Dear Respondent,

We request your help in a research project to study Software maintenance and evolution. This survey is part of a thesis which is to be submitted to the School of Software Engineering and Computer Science at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Computer science. The thesis is equivalent to 20 weeks of full time studies.

Included with this letter is a short questionnaire that asks a variety of questions about the impact software comprehension in software maintenance and evolution. We would like you to look through the questionnaire and, if you choose to do so, complete it and send [or give] it back to us before 8 December 2009. It should take you about 30 minutes to complete.

The results of this project will be a contribution to scientific knowledge about the software maintenance and evolution process. Through your participation we hope that the results of the survey will be useful for students and researchers in the field of software engineering. We hope to share our results by publishing them in a scientific journal and in the Electronic Thesis Archive of Blekinge institute of technology where students all over the world can use them.

We guarantee your complete anonymity, and promise not to share any information that identifies you with anyone outside our research group which consists of Usman Akhlaq, Muhammad Usman Yousaf and our thesis supervisor Jeff Winter.

If you do not feel comfortable handing in your survey to us personally, you may also return it to us by email, at usak08@student.bth.se, usyo07@student.bth.se or by regular mail through the address given below.

We hope you will take the time to complete this questionnaire and return it. Your participation is of course voluntary but would be greatly appreciated. Regardless of whether or not you choose to participate, please let us know if you would like a summary of our findings.

If you have any questions or concerns about completing the questionnaire or about participating in this study, you may contact us at the following address
Usman Akhlaq
Peder Holmsgatan 4A,
372 35, Ronneby
Sweden

Sincerely,
Usman Akhlaq and Muhammad Usman Yousaf
8.2 Appendix B: Survey Questionnaire

The personal information requested below is required for the analysis of the answers and will also help us in categorizing the results. We guarantee that the results will be treated in a way that maintains your anonymity.

Personal information
Note: It is important that you fill in the fields marked with *

Name: ___________________________
*Company: ___________________________
*Title/role: ___________________________
Telephone Number: ___________________________
*Email: ___________________________
Number of members in the maintenance team: ___________________________
*Working Domain: ___________________________
Hardware / Software environment: ___________________________
Experience in Maintenance (in years/months): ____________
Qualifications:____________________

Questions:
Note: The answers should be given on behalf of your organization instead of in general.

1- How are the development and maintenance teams structured?
   i- Separate teams for both maintenance and development. (y/n)__________.
   ii- The same team for both maintenance and development. (y/n) ___________.

2- Does the type of maintenance performed (Corrective, Adaptive, Perfective, and Preventive) influence the approach or methodology adopted?
   Yes/no: ________________
   If yes how?
   ___________________________________________________________________
   ___________________________________________________________________

3- Do you think code comprehension can be improved in a system which is currently under maintenance?
   Yes/no: ________________
   If yes, in what way?
   ___________________________________________________________________
4- What are the three major problems that you face in the maintenance process?
____________________________________________________________________
____________________________________________________________________
____________________________________________________________________

5- Describe the approaches or methodologies you use in code comprehension?
____________________________________________________________________
____________________________________________________________________
____________________________________________________________________

6- Do you use any tool(s) for code comprehension in the maintenance process?
Yes/no: ________________
If yes, name it/them?
____________________________________________________________________
____________________________________________________________________
____________________________________________________________________

7- What are the three major problems that you face in code comprehension?
____________________________________________________________________
____________________________________________________________________
____________________________________________________________________
____________________________________________________________________

8- How can code comprehension be improved?
____________________________________________________________________
____________________________________________________________________
____________________________________________________________________
____________________________________________________________________
____________________________________________________________________
9- How would you compare software maintenance and development, which provides more of the following, or there is no difference?

<table>
<thead>
<tr>
<th></th>
<th>Maintenance</th>
<th>Development</th>
<th>No difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Challenge</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Responsibility</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Salary</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Career growth</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Learning opportunity</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

10- How would you compare software maintenance and development, which requires more of the following, or is there no difference?

<table>
<thead>
<tr>
<th></th>
<th>Maintenance</th>
<th>Development</th>
<th>No difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Experience</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Design Skills</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Programming skills</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Problem solving skills</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Foreseeing/prediction skills</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

11- Which of the following is more time consuming activity in software maintenance? [write down an answer from 1(Low) to 6(High)]

<table>
<thead>
<tr>
<th>Activity</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comprehension(understandability)</td>
<td></td>
</tr>
<tr>
<td>Analysis</td>
<td></td>
</tr>
<tr>
<td>Design</td>
<td></td>
</tr>
<tr>
<td>Implementation</td>
<td></td>
</tr>
<tr>
<td>Testing</td>
<td></td>
</tr>
<tr>
<td>Documentation</td>
<td></td>
</tr>
</tbody>
</table>
12- Which of the following do you think require(s) more when you are maintaining a system that has been developed by your organization?

<table>
<thead>
<tr>
<th></th>
<th>Time and cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change request understanding</td>
<td></td>
</tr>
<tr>
<td>Code comprehension</td>
<td></td>
</tr>
<tr>
<td>Concept location</td>
<td></td>
</tr>
<tr>
<td>Change implementation</td>
<td></td>
</tr>
<tr>
<td>Cause-effect relationship</td>
<td></td>
</tr>
</tbody>
</table>

13- Which of the following do you think requires more when maintaining a system which is developed by some other organization?

<table>
<thead>
<tr>
<th></th>
<th>Time and cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change request understanding</td>
<td></td>
</tr>
<tr>
<td>Code comprehension</td>
<td></td>
</tr>
<tr>
<td>Concept location</td>
<td></td>
</tr>
<tr>
<td>Change implementation</td>
<td></td>
</tr>
<tr>
<td>Cause-effect relationship</td>
<td></td>
</tr>
</tbody>
</table>

14- Which of the following measures and metrics do you use to improve code understandability:

<table>
<thead>
<tr>
<th>Measures and metrics</th>
<th>Yes</th>
<th>No</th>
<th>If yes write the name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complexity measures</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Size measures</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Descriptiveness measures</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Consistency measures</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Modularity measures</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

15- To what extent it is beneficial to properly document every implemented change?

<table>
<thead>
<tr>
<th>Percentage</th>
<th>Not at all</th>
<th>0-30% (Low)</th>
<th>31-70% (Medium)</th>
<th>71-100% (High)</th>
</tr>
</thead>
</table>

16- To what extent it is beneficial to properly document each change request?

<table>
<thead>
<tr>
<th>Percentage</th>
<th>Not at all</th>
<th>0-30% (Low)</th>
<th>31-70% (Medium)</th>
<th>71-100% (High)</th>
</tr>
</thead>
</table>

17- Would you like us to have a summary of our findings to you?

Yes/no: ________________
### Appendix C: Sasa Dekleva’s Classification of Maintenance Problems [22]

#### Classification of problems into categories

<table>
<thead>
<tr>
<th>Category</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Maintenance Management</strong></td>
<td>Lack of managerial understanding and support</td>
</tr>
<tr>
<td></td>
<td>Large backlog</td>
</tr>
<tr>
<td></td>
<td>Changing priorities</td>
</tr>
<tr>
<td></td>
<td>Lack of maintenance methodology, standards, procedures, and tools</td>
</tr>
<tr>
<td></td>
<td>Inadequate testing methods</td>
</tr>
<tr>
<td></td>
<td>Performance measurement difficulties</td>
</tr>
<tr>
<td><strong>Organizational Environment</strong></td>
<td>Strategic plans</td>
</tr>
<tr>
<td></td>
<td>Adopting to the rapidly changing business environment</td>
</tr>
<tr>
<td></td>
<td>Lack of support for reengineering</td>
</tr>
<tr>
<td></td>
<td>Contribution measurement difficulties</td>
</tr>
<tr>
<td><strong>Personnel Factors</strong></td>
<td>Low morale due to the lack of recognition and respect</td>
</tr>
<tr>
<td></td>
<td>High turnover causing a loss of expertise</td>
</tr>
<tr>
<td></td>
<td>Lack of maintenance personnel, particularly experienced maintainers</td>
</tr>
<tr>
<td></td>
<td>Maintainers’ lack proper training</td>
</tr>
<tr>
<td></td>
<td>Understanding and responding to business needs</td>
</tr>
<tr>
<td><strong>System Characteristics</strong></td>
<td>Program code is complex and unstructured</td>
</tr>
<tr>
<td></td>
<td>System documentation is incomplete or nonexistent</td>
</tr>
<tr>
<td></td>
<td>Antiquated systems and technology</td>
</tr>
<tr>
<td></td>
<td>Integration of overlapping and incompatible systems or subsystems</td>
</tr>
</tbody>
</table>