Thesis no: MSSE-2015-12

Agile in the context of Software Maintenance

A Case Study

Gopi Krishna Devulapally

Faculty of Computing
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden

This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Software Engineering. The thesis is equivalent to 20 weeks of full-time studies.

Contact Information:
Author(s):
Gopi Krishna Devulapally
E-mail: gode14@student.bth.se

University advisor:
Prof. Conny Johansson
Dept. Software Engineering

Industrial advisor:
Mayank Dalal
Associate Director Quality
Capgemini, Mumbai

Faculty of Computing
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden

Internet : www.bth.se/didd
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57

Abstract

Context Adopting agile practices has proven to be successful for many practitioners both academically and practically in development scenario. But the context of agile practices adoption during software maintenance is partially studied and is mostly focused on benefits. The success factors of agile practices during development cannot be related to maintenance, as maintenance differs in many aspects from development. The context of this research is to study the adoption of different agile practices during software maintenance.
Objectives In this study, an attempt has been made to accomplish the following objectives: Firstly, to identify different agile practices that are adopted in practice during software maintenance. Secondly, identifying advantages and disadvantages of adopting those agile practices during software maintenance.
Methods To accomplish the objectives a case study is conducted at Capgemini, Mumbai, India. Data is collected by conducting two rounds of interviews among five different projects which have adopted agile practices during software maintenance. Close-ended questionnaire and open-ended questionnaires have been used respectively for first and second round interviews. The motivation for selecting different questionnaire is because each round aimed to accomplish different research objectives. Apart from interviews, direct observation of agile practices in each project is done to achieve data triangulation. Finally, a validation survey is conducted among second round interview participants and other practitioners from outside the case study to validate the data collected during second round interviews.
Results A simple literature review identified 30 agile practices adopted during software maintenance. On analyzing first round of interviews 22 practices are identified to be mostly adopted and used during software maintenance. The result of adopting those agile practices are categorized as advantages and disadvantages. In total 12 advantages and 8 disadvantages are identified and validated through second round interviews and validation survey respectively. Finally, a cause-effect relationship is drawn among the identified agile practices and consequences.

Conclusions Adopting agile practices has both positive and negative result. Adopting agile practices during perfective and adaptive type of maintenance has more advantages, but adopting agile practices during corrective type of maintenance may not have that many advantages as compared to other type of maintenance. Hence, one has to consider the type of maintenance work before adopting agile practices during software maintenance.

Keywords: Software maintenance, Agile practices, Agile, Advantages and Disadvantages.

Acknowledgments

Great and wonderful experiences are the words which I would use to describe my journey (Master Thesis). Conducting a master thesis at an organization is always a wonderful experience, got to meet many people and also got to learn many new things from them. Received very great mentor-ship from internal supervisor (at college) and from external supervisors (at organization) during my thesis.

I would like to show gratitude to my teacher and guide Assistant Professor. Conny Johansson of Software Engineering Department, for his valuable guidance and continuous encouragement during the course of the work. I am indebted for his support which made it possible for me to stand up to the challenges offered by the task and come out successfully.

I am grateful to Hrushikesh Mangalampalli, Vice-President HR, Capgemini Mumbai, for providing me the opportunity to conduct my thesis at Capgemini, Mumbai. I am very thankful to Mayank Dalal, Upendera Bhabal and Ravi Prakash Singh for their support and encouragement. Also, I am very thankful to all my colleagues at Capgemini, who had showed enthusiastic participation during my study.

Finally, I would like to show my heartfelt gratitude to my parents D. Raghu and Annapurna, my sister Krishna Priya and my dear uncle Narayan Rao for their everlasting support, love and wishes. I would forward my thanks to all people, who missed my mention inadvertently, and from whom I received direct or indirect help.

Thank you all.

Gopi Krishna Devulapally

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem Statement

"The need to minimize software cost suggests that large-program structure must not only be created but must also be maintained if decay is to be avoided or postponed" –Belady and Lehman.[1]

For many years software maintenance has been a critical challenge and a nightmare to many business users and IT management [2]. It is reported by Sneed and others in[3, 4, 5, 6], that more than 50 percent of IT Budget is spent on software maintenance by many companies who developed their own applications over the years. Even though previously developed systems have to be maintained to ensure their value and sustainability within an organization. Still software maintenance is always ill-treated in field of software engineering [7].

Software maintenance is considered to be different from software development, for instance software maintenance depends on comprehension, but it is not the case with software development [8]. Hence software maintenance has its own processes and models [9]. Many researchers came forward to propose processes and model(s) in regards to software maintenance. Quickly Modify Model, Boehm Model, IEEE Model, Iterative Enhancement Model, Osborne model and reuse-oriented model are quite few software maintenance models reported by Yongchang and Zhongjing et al in [10]. But these existing models and processes fit only to certain situations and do not provide permanent solution to the issues faced during software maintenance; for instance lack of team morale and visibility due to complexity of maintenance project and lack of communication techniques [10]. Choudhari and Suman [9] report that those existing models are developed based on the the traditional software development. Instead of decreasing the burden of software maintenance, it has compounded it.

Software methodologies are continuously changing due to change in demands from the customer and also change in technologies [11]. So in today's competitive business environment many emergent companies have been adapting their strategies, structures and polices to that environment [12]. Such companies need information systems which continuously evolve with changing requirements, but

such flexibility to adapt to the changing requirements is lacked by traditional plan driven approaches during development phase [11]. Hence agile approaches came into existence. Traditional approach focused on creating efficiency by separating brain work from manual work, so in order to minimize deviations from standard design, continuous inspect and adapt approach is necessary to tackle uncertainties, which is considered to be primary characteristics of an agile environment. Thus to incorporate more visibility and update customer priorities, agile methodologies are replacing traditional plan-driven approaches [11].

Lehman's first law of software evolution states that, *"a program that is used undergoes continual change or becomes progressively less useful"*[1]. So most of the systems are maintained and further enhanced functionally after the initial development. Although software maintenance is always ill-treated and under researched in field of software engineering, but it is the predominant activity in the field of software engineering [13]. Svensson and Host in [3] argue that many industrial projects are starting from an already developed code, i.e. most project nowadays are not just development projects but there are also projects which start-off with maintaining already developed system. So after the initial development project has to go maintenance, and at times maintenance activities are always carried out by different personnel. In this case those personnel's are called maintainers and they do face many work practice challenges in the way the personnel's who initially developed the system. Hence even the maintainers seek for the support of flexible processes and methodological support. An extensive research is done to study the applicability of traditional approaches in the context of software maintenance [13, 14, 15, 16, 17] , but little consideration has been shown towards applicability of agile in the context of software maintenance [13]. Agile methodologies were developed for software development, so it is a good assumption to consider that agile in the context of software maintenance may or may not have challenges [18]. Whilst there are different studies which focused on applicability of such agile approaches to software maintenance, investigation which focuses on the collection and analysis of practitioner's view of agile in the context of software maintenance is missing or less dealt. This study aims to conduct a empirical investigation through a case study at Capgemini, Mumbai, an Indian subsidiary of an multinational French based company. To investigate which agile practices have been adopted during software maintenance and what are the advantages and disadvantages of adopting those agile practices in the context of software maintenance, from a maintenance practitioners perspective.

1.2 Research Aim and Objectives

The aim of this thesis is to provide support to software maintenance practitioners in visualizing the advantages and disadvantages of adopting agile during software maintenance.

In order to meet the aim/goal, following research objectives are set:

- Identifying agile practices which are adopted during software maintenance at a software company operating globally.
- Analyzing the results of agile practices adoption during software maintenance at a software company operating globally.

1.3 Research Questions and Instrument

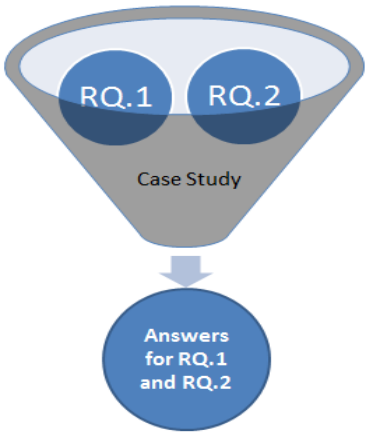


Figure 1.1: RQ answering Instrument

Both Research questions reported in this section are answered by conducting a case study at Capgemini, as shown in figure 1.1. Based on the aims and objectives, the research questions and their motivation can be summarized as follows.

RQ1: What are the agile practices adopted during software maintenance?

Motivation: Dynamic business environment has led many organizations to adopt agile, since it constantly helps to meet the changing requirements of the customer. Hence, many empirical studies identified different agile practices that can be adopted during software development (such as in [19] and [20]). Though there are many studies, adoption of agile practices during software maintenance is rarely looked into [13]. The goal is to identify which agile practices are adopted during software maintenance, so that it further helps organizations in agile adoption depending on their context.

RQ2: What are the advantages and disadvantages of adopting agile practices during software maintenance?

Motivation: The result of agile adoption are only studied in the context of software development [14]. In order to better understand the agile methodology, it is also important to know the results of agile adoption even during software maintenance. The goal is to analyze the results of adopting agile practices during software maintenance

1.4 Expected Research Outcomes

This thesis will present the knowledge gained through answering the above mentioned research questions. In particular, the thesis will describe the Agile practices which are adopted during software maintenance. Besides, it will also explore the results, i.e. advantages and disadvantages of adopting agile practices during software maintenance.

1.5 Structure of the Thesis

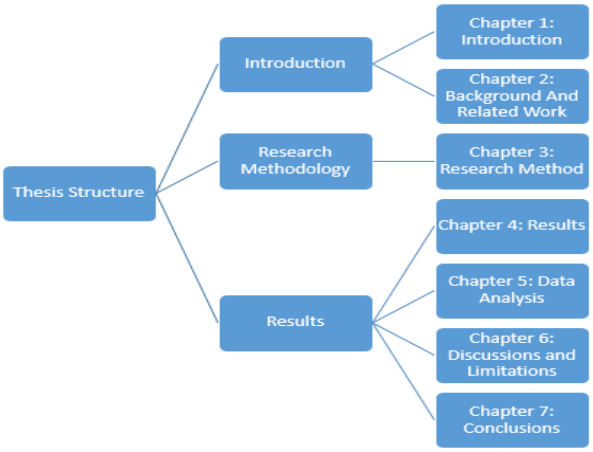


Figure 1.2: Thesis Structure

This thesis consists of seven chapters: Introduction, Background and Related Work, Research Method, Results, Data Analysis, Discussions and Limitations, and Conclusions. The introduction (Chapter 1) begins with an account of the Problem Statement (section 1.1) to the study. Then, the Research Aim and Objectives (section 1.2) are presented by, which is followed by Research Questions (section 1.3), Expected outcomes (section 1.4) and Thesis structure. Background and Related Work (Chapter 2) consists of three main parts: Agile Methodology

(section 2.1), Software Maintenance (section 2.2) and Agile Maintenance (section 2.3). Chapter 3 entails a description of the Research Method used in this study, forming the third chapter of the report. The chapter focuses on the methods that were used in conducting the empirical research, namely Literature Review and Case Study. Chapter 4, presents the results of the data collection and data analysis of the interviews, which were conducted as a part of case study. Chapter 5 Chapter 6 Chapter 7 concludes the findings of the research and includes an assessment of the research findings reliability and validity and Potential areas for further research are lifted, and the academic and practical value of the findings is discussed.

Chapter 2

Background And Related Work

In order to situate the present research it is necessary to explore the area software maintenance and its context in agile from the literature. Hence, this chapter introduces background of software maintenance and agile methodology in sections 2.1 and 2.2 respectively, after which presents the related work in section 2.3.

The aim of literature review in this study was two-fold. Firstly, to understand and present background of the study. Secondly, to acquire knowledge of different agile practices adopted during software maintenance. A simple literature review is conducted as it helps acquire knowledge of related work in short interval of time. Literature review used in this study is well-explained in chapter 3.

2.1 Agile Methodology

2.1.1 Overview

Agile Methodology constitute a set of practices that have been created by experienced practitioners for software development such as scrum. The activities done in any agile methodology such as daily stand up meetings are referred to as agile practices [21]. Today's dynamic business environment needs services from the service provider who can continuously adapt to unpredictable scenarios and this unpredictable scenarios arise when there is continuous change in customer demands. As agile addresses the challenge of unpredictable scenarios, therefore many organizations are adopting to their development methodology to agile. These methods can be seen as a reaction to traditional waterfall or plan-based methods, which often claim that predictable solutions exist for every problem, thereby traditionalists believe in extensive planning and processes. Ericksson et al in [22] defines agile as follows:

"agility means to strip away as much of the heaviness, commonly associated with the traditional software-development methodologies, as possible to promote quick response to changing environments, changes in user requirements, accelerated project deadlines and the like". (p. 89)

It was Agile Alliance, which first coined the term *Agile Software Development*, through agile manifesto [23].

Manifesto for Agile Software Development

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

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

That is, while there is value in the items on the right, we value the items on the left more" [23].

There are certain features which makes a development method an agile method, as reported by Abrahamsson et al in [24]. These features are iterative (i.e. frequently released small software); cooperative (it is a people oriented process rather than process, document or code oriented); adaptive (it is designed to accept changes); small teams, having a simple design; development cycle lasting for 2 to 4 weeks of time; documentation only when needed; mostly communicate face to face; testing regularly; collective code ownership and focus towards quality of code (through refactoring) and product. These characteristics have shown significant improvements in development scenario; for instance Li et al [25] reported that, by using Scrum (one of the agile method) defects were discovered and corrected at very early stages of project, thereby increasing defect fixing efficiency. But applicability of these characteristics in maintenance scenarios is still under-researched area [13].

2.1.2 Family of Agile

A description of different agile methodologies as reported by Abrahamsson et al in [24] are shown in the table 2.1.

Project Code	**Domain**

Extreme Programming	Evolved after the problems caused due to initial traditional development cycles. Consist of different practices: Planning game, Small releases, Metaphor, Simple Design, Testing, Refactoring, pair programming, collective ownership, Continuous integration, 40 hour week, On-site Customer, Coding Standards and Open Work Space
Scrum	Scrum originally taken from the game rugby, which denotes getting out-of Play ball back into the game with teamwork. Adapted early in 1986 by Takeuchi and Nonaka for product development process. Mainly focuses on project management. It believes in self organizing team. Consist of different practices: Product Backlog, Sprint, Sprint Planning Meeting, Sprint Backlog, Daily Scrum Meeting and Sprint review meeting.
Crystal family of methodologies	Includes number of different methodologies based on size of the team and criticality of the project. Each methodology is marked with color: Clear, Yellow, Orange and Red. Darker the color, heavier the methodology. Most common methodology is Crystal Clear. Consist of different practices: Staging, Revision and Review, Monitoring, Parallelism and flux, Holistic diversity strategy, Methodology-tuning technique, User viewings and Reflection workshops. It suits for projects having less criticality and teams which are co-located.
Feature Driven Development	It is a combination of model-driven and agile methods. It does not cover whole development process, but rather focuses on design and development phases. As it doesn't cover whole development process it is designed to be adaptive with other activities during software development. It consist of different practices: Domain Object Modelling, Developing by Feature, Individual Class (Code) Ownership, Feature Teams, Inspection, Regular Builds, Configuration Management and Progress reporting. Unlike other agile development methodologies, it is especially used for developing life-critical applications.

Dynamic System Development Method	One of the framework representing RAD. It consists of different phases: feasibility study, business study, functional model iteration, design and build iteration, and implementation. Nine practices define ideology for DSDM, which are called principles: Active user involvement is imperative, DSDM teams must be empowered to make decisions, The focus is on Frequent delivery of products, Fitness for business purpose is the essential criterion for acceptance of deliverables, Iterative and incremental development is necessary to converge on an accurate business solution, All changes during development are reversible, Requirements are baselined at a high level, Testing is integrated throughout the life cycle and A collaborative and cooperative approach shared by all stakeholders is essential. Mostly suited for developing business systems of varying sizes.
Adaptive Software Development	ASD is carried out in three phases: Speculate, Collaborate and Learn. Apart from being incremental and iterative, it also encourages constant prototyping. It proposes very few practices, which limits the usage of ASD. Unlike other methodologies it supports distributed development, by providing solutions to the challenges faced during distributed development.

Table 2.1: Family of Agile Methods.

2.1.3 Agile: Benefits and Practices

With the advancement in software engineering research field, many software industries are rapidly shifting their focus to new development trends in order to adapt to the features of today's dynamic environment i.e. agility and time-to-market. Based on these facts, development organizations are adopting agile methods as they focus on early release of working product (time-to-market) through collaborative practices such as daily-stand-up meetings, pair-programming and frequent communication with customer. Apart from the collaborative practices, there are many other agile practices, such as:

- Continuous Integration.
- Small releases.
- Simple Design.
- Collective Code Ownership.

- Pair-Programming.
- On-site customer.
- Coding Standards.
- Daily Stand-up Meetings.
- Daily Builds.
- Test Driven Development.
- Test Automation.
- Acceptance Test.
- Sprints/Iteration.
- Sprint Review.
- Scrum of Scrum.
- Product Backlog.
- Sprint Demo.
- Task Boards.
- Code Reviews.
- User Stories.
- Planning Game.

These practices are most widely used agile practices as reported in studies [26, 27]. Apart from these practices there are other practices such as, feature driven development and crystal methods. Upon adopting those agile practices there are benefits which are perceived by practitioners [27, 28] and these are:

- Productivity Improvement.
- Cost Reduction.
- Time-to-market compression.
- Quality Improvement
- Improvements in meeting customer needs
- Increased flexibility in development teams.

Having these benefits during development phase may be productive, but it is also important to investigate whether these benefits are realized even during software maintenance phase. So that it helps practitioners in visualizing the risks and benefits of adopting agile for software maintenance.

2.2 Software Maintenance

2.2.1 Overview

The term "maintenance" has been in use from many years, i.e. since 1960, to describe change or modification applied to an existing system [14]. Software maintenance as defined by IEEE,

"Modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment".

So it is a separate stage wherein, a product moves after its initial development. During this stage the software undergoes many changes or modifications, which means the software is under continuous evolution [29]. Maintenance was initially seen as a single post-delivery activity [30], but it is considered that the activities undertaken during software maintenance vary greatly [31]. So Bennett and Rajlich proposed a software life cycle, which concentrated mainly after development. Those phases are as follows:

- **Initial development**, during this period first version of the software is developed and made live for the end users or customers.
- **Evolution**, takes place after the initial development (if successful). In this stage, the system is adapted to changing requirements and also involves correcting the faults.
- **Servicing**, takes place after evolution, in which only tactical changes such as patches, code changes and wrappers are made to the system.
- **Phase-out**, no more servicing done to the system, but still remains to be in production.
- **Close-down**, system use is disconnected and the users of the system are directed towards replacement.

However, this life cycle seems to be longer than normal conventional software life cycle. Hence maintenance is often considered as a departmental unit rather than time-limited project [18]. Software maintenance starts in response to software failures, environmental changes and in response to change requests made by users. So Bennett and Rajlich also proposed a model for those changes, i.e. change mini-cycle:

- Request for Change
- Planning phase
    - Program Comprehension
    - Change impact analysis
- Change Implementation
    - Restructuring for change
    - Change propagation
- Verification
- Re-documentation

2.2.2 Types of Maintenance

Maintenance is always considered as bug-fixing activity, but Lientz and Swanson in [30], argued that approximately 75 percent of maintenance activities are related to new development. There are different types of maintenance activities, a good classification was again given by Lientz and Swanson in [30], which was again revised in ISO/IEC 14764 [32]. The purpose of classification was to understand different activities during maintenance. A good classification as proposed by Lientz and Swanson in [30], helps different personnel in maintenance to understand different activities. Different types of maintenance activities are as follows:

- Adaptive: Adapting to new environment
- Perfective: Adapting to new requirements
- Corrective: Fixing errors
- Preventive: Making changes to prevent problems in future; for instance identifying undiscovered bugs in the system and correcting them.
- Emergency: After revision done by IEEE, it has added a new type of maintenance known as emergency maintenance. It is a special case of maintenance, which are unscheduled and corrective in nature, mostly occurs in case of system failure.

2.2.3 Maintenance: its different from development

As stated earlier and also observed in the change mini-cycle proposed by Bennett and Rajlich in [31], that program comprehension is one of the criteria which makes software maintenance unique from software development. The purpose of program comprehension during software maintenance is to identify application domain concepts in the code [31]; for instance if there is a change request in a particular application of a system, then program comprehension helps in identifying the the code where that particular application of the system is implemented. Apart from program comprehension, change impact analysis during maintenance also makes it unique from development. The purpose of the change impact analysis is to identify the impacted areas if a particular change is been implemented in the system; for instance if a change request is received to an application of the system, and after analysis it was found that making a change to that application of the system might also effect the other application of the system. So change impact analysis provides different areas in a system, which gets impacted if a change request is implemented. It is also known that there would be many customers to the system depending on the type of application. So it can be concluded that, tasks during maintenance are not just plain tasks dealing just with one instance of the system rather involves tasks which deal with different instances and different customers of the system [31]. This may be a good reason to adopt agile during maintenance, since it requires more of customer involvement and interaction.

2.2.4 Maintenance: Explored and Unexplored

Most of the research in software maintenance is focused towards different aspects such as metrics [33, 34], estimation [35, 36], testing [29], management and coordination [37], documentation [37], and few towards maintenance models [10]. To address some of the issues many studies have proposed solutions, which to some extent have reduced the burden of software maintenance. The above stated research address few issues of maintenance. But few studies [18, 38, 39], report that integrating agile to software maintenance may offer something to it.

2.3 Agile Maintenance

Previous sections have provided a background to this study, but complementing related work with background would further enhance the understandability of the reader. Hence this section provides fragmental view of agile practices adoption and its consequences in the context of software maintenance. Apart from reporting the related work, it also provides the area of study.

2.3.1 Area of Study

In this section the area of study is emphasized. As shown in figure 2.1, Pink shaded area represents Agile Methodology, while Yellow shaded area represents Software Maintenance. The overlapped area between pink and yellow shaded area represents the context of this study. The overview of the overlapped area, i.e. Agile Maintenance is presented in section 2.3.3.

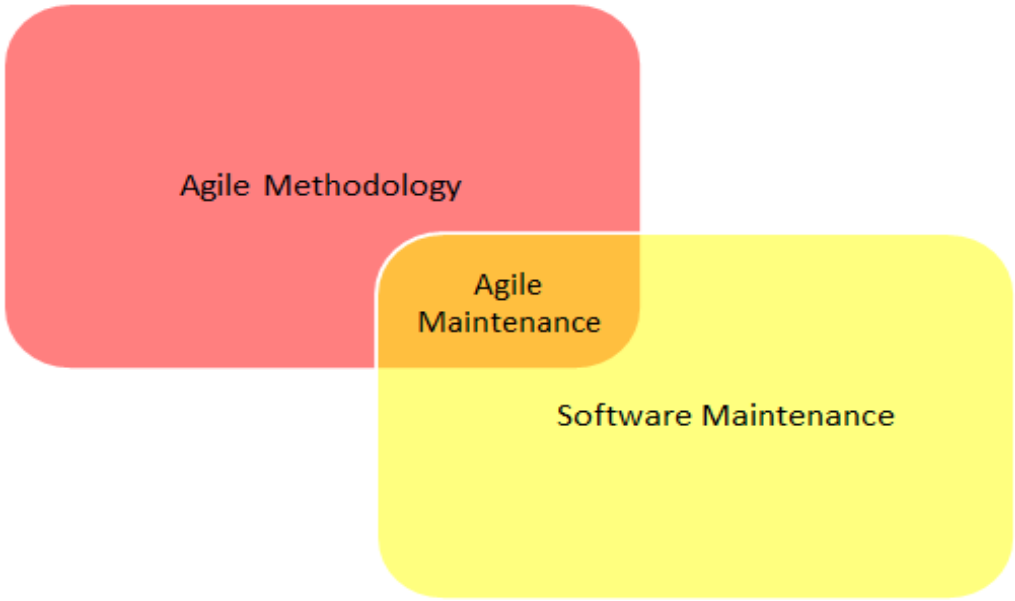


Figure 2.1: Context of Study

2.3.2 Agile Practices adopted during Software Maintenance

There are various agile practices which have been adopted by software maintenance practitioners as reported by different empirical studies in [13, 38, 39, 40, 41]. Svensson and Host in [38] have studied one of the agile methodology (i.e. extreme programming) adoption in an maintenance software development environment for eight months. Their study concluded that adopting XP during maintenance is difficult, hence changes have to be made to the existing practices. Jain in [39] reported his experience of introducing agile (i.e. extreme programming) in an offshore project which dealt with software maintenance. In his report [39], different agile practices used during maintenance were planning game, 2-week iteration, story cards, iteration planning meetings, small releases, refactoring, pair-programming, collective code ownership, continuous integration Test driven development (TDD), stand-up meetings, 40 hour week and coding standards and did not include practices such as on-site customer, simple design, automated testing and metaphor. Shaw in [41], studied use of agile practices, (which were

mainly practices of extreme programming) in the context of software maintenance of complex system. Similar to this study Shaw has rated each practice in terms of usage during maintenance. Few practices which are ought to be followed during maintenance as reported by Shaw are On-site customer, integration and code ownership. In similar to [41], Poole and Huisman in [40] reported the status of adoption of various agile practices (extreme programming) during a re-engineering project. Kajko-Mattsson and Nyfjord in [13] have derived a agile model for maintenance. This model mainly consisted of various practices from both extreme programming and scrum. Kajko-Mattsson and Nyfjord created a agile model which consisted of different practices from extreme programming and scrum, as they thought that creating a model from those two methodologies of agile would include practices from management perspective and engineering perspective. This model encouraged various practices such as product backlog, test first and code, iteration planning, prioritize tasks,stand-up meetings and retrospectives. Based on the above empirical studies different agile practices adopted during maintenance are enlisted in the table 2.2. The enlisted practices in table 2.2 are mainly from scrum and extreme programming as those two methodologies are widely adopted [13]. Those identified agile practices from the literature are used in the case study to know its usage and also to investigate whether benefits reported in section 2.1.3 are also perceived by software maintenance practitioners.

Agile Practice	**Source**	**Agile Practice**	**Source**
Pair-programming	[38, 39, 40, 41]	Coding Standards	[38, 39, 40]
Refactoring	[38, 39, 40]	Unit Testing	[38]
Planning Game	[38, 39, 40]	System Metaphor	[38, 40]
Stand-up Meetings	[39]	Co-located Teams	[38]
Retrospectives	[38]	Code Reviews	[38]
Scrum of Scrums	[13]	Acceptance Test	[13]
Test Driven Development (TDD)	[39]	40 hour Week	[38, 39, 40, 41]
Iteration Review	[13]	Simple Design	[38, 40]
Product Backlog	[13]	Small Releases	[38, 39, 40]
Automated Testing	[38]	Continuous Integration	[38, 39, 40]
Planning Meeting	[13, 39]	User Stories	[13, 39]
Short Iteration	[39, 41]	Task Prioritization	[41]
On-site Customer	[38, 40, 41]	Daily Builds	[42]
Demos	[13]	Small Teams	[38]
Collective Code Ownership	[38, 39, 40, 41]	Velocity	[41]
Bug Tracking	[39]	Release Planning	[40]
Task Board	[38, 39, 40]	Scrum Master	[13]

Table 2.2: Agile Practices Adopted during Software Maintenance

2.3.3 Overiew

Reyes et al in [14] reported that there has been some investigation on applicability of agile methodologies in software maintenance context, but those studies ([9, 13, 16, 39, 40]) covered only technical, procedural, activities and benefits, and lacked practitioner's perspective. As stated earlier that there is difference between software maintenance and development, hence practices, tools and techniques aligned to agile during development may or may not produce good results during maintenance [43]. Different studies reported different results on the application of agile during software maintenance. Kajko-Mattsson and Nyfjord [13], initially proposed an agile evolution and maintenance model (combination of XP and Scrum) by conducting a literature review. The model had two main phases pre-implementation phase and implementation phase. After proposing a model Kajko-Mattsson and Nyfjord [13] have compared it with standard industry software process model, to identify discrepancies. Finally, Kajko-Mattsson and Nyfjord [13] have concluded that there should be some changes to their model, before adapting it to software evolution and maintenance. Even though this

study [13] had practitioner's perception, Kajko-Mattsson and Nyfjord have only concentrated on calibrating there model to industry standard software process, rather than reporting anything on benefits and challenges of using that model. Choudhari and Suman [9], reported that unstructured code, poor visibility, team morale, lack of communication techniques and lack of test suites are quite a few challenges of complex maintenance life cycle. Hence, Choudhari and Suman have proposed a new model, i.e. iterative maintenance lifecycle using Extreme Programming, which helps in mitigating those issues, but this study seems to be just an literature review and haven't tested the proposed model with practitioner's approach, which could have given more insights about applicability of it during maintenance. Knippers [16], conducted a literature review to find the relationship between software maintenance and agile software development. It [16] reports that there exist eight factors, which effect software maintainability, and those eight factors have different relationship with agile methodologies. Even though Knippers [16] reported few advantages on using agile methodologies, but context of his study did not relate to applicability of agile during maintenance. Poole and Huisman in [40], reported their experience on using Extreme Programming during a re-engineering project, where they have found that pair-programming improved their code quality, which effected their productivity in a positive way. Jain in [39], reported his experience on applying agile during a offshore software maintenance project. Jain [39] reports that initially the project had faced many problems such as lack of trust, delay in feedback and loss of context, but on applying agile in this scenario many of those issues were resolved. Jain [39], also reports that characteristics of agile such as automation and refactoring are success factors during offshore maintenance project. Svensson and Host in [38], also believe that initial assessment is required before applying agile during maintenance, and continuous testing may not be realized during maintenance. Apart from those studies, a new software development methodology DevOps, tries to bring agile thinking into aspects of development and operations [44]. DevOps brings both development and operation teams together to improve the performance of each other and also to reduce the communication gaps between them. Although this methodology looks into some challenges of software maintenance, it is relatively a very new agile methodology which made it difficult to gather resources which deals with the adoption of DevOps during maintenance.

Most of the investigation reports either advantages of applying agile to maintenance environment or proposes a maintenance model which adapts agile characteristics (especially XP) using literature review. Only few studies report practitioner's perspective, and even those studies only report advantages of using agile in software maintenance or state their experience of using XP during maintenance; none of them discussed which practices work during maintenance and none of them discussed both advantages and disadvantages of applying agile practices during maintenance.

Chapter 3

Research Method

This section describes the research method, which will be used in this study to accomplish the aforementioned objectives (in section 1.2) of the research. The selection of research method is motivated. It also describes, which type of study is used to answer the research questions.

Motivation for Research Method?
There are quite a few research methodologies, which are used to conduct research. Runeson and Host [45] reported that there are four major research methodologies, Case study, Experiment, Survey and Action Research.
Action research, in this study no new actions are proposed, instead it aims to understand different agile practices that are adopted during maintenance. So this study aims to understand the real-life context, rather than proposing new actions to the context.
Survey, is collection of opinions from different practitioners. In this study, identifying different agile practices is one objective but it also aims to understand how the adopted practices benefit or be a challenge to the practitioners during maintenance. So collecting such data from the opinions of the practitioner would be very difficult using surveys.
Experiments, is study of *"measuring effects of manipulating one variable on another and subject are assigned to the treatments by random"*, but in this case variables (agile practices) are not known upfront, so it makes it difficult to assign a subject to study the context (software maintenance). Also it is very difficult to replicate the experiment.

As this study aims to identify different advantages and disadvantages that are caused due to the adoption agile practices during maintenance, hence a case study is selected as the research method, since case study provides an in-depth investigation of a single or small number of units over a period, hence case study is considered to be appropriate research methodology in this context.

Few steps are followed to conduct a literature review in this study and those steps are aligned to the steps reported by Rowley and Slack [46]. The nature

of literature review conducted in this study aims to identify problem domain; to find related work for this study; and to facilitate in building certain knowledge of agile practices adopted during software maintenance. The steps followed in this typical nature of literature review are as follows:

Step 1: Evaluating the information resources: As reported in [46] there are quite a few information resources which are to be evaluated, but for the convenience and for this study to be more effective, information resources which are selected in this study are from online databases and Books.

Step 2: Literature searching in the above mentioned information sources: Different keywords will be formed using certain guidelines provided in [46], and different sources are retrieved. The availability of the source and its abstracts are generally used as an initial evaluation criteria for the selection sources to this particular study. If the sources are available and abstract found to be related to this study, then the sources are completely read to further evaluate the source relevance to this literature study. It is also worthy to note that major information sources used in this study are online databases, as the researcher had the access many online databases. This study used Google Scholar, IEEE, Scopus and ScienceDirect, as an online information sources for retrieving the published research articles.

Step 3: Drawing up the sources together: The selected sources for literature review are studied. These sources are categorized using certain themes. Using those themes relevant studies are identified, using those relevant studies, problem domain is derived and agile practices adopted during software maintenance are identified.

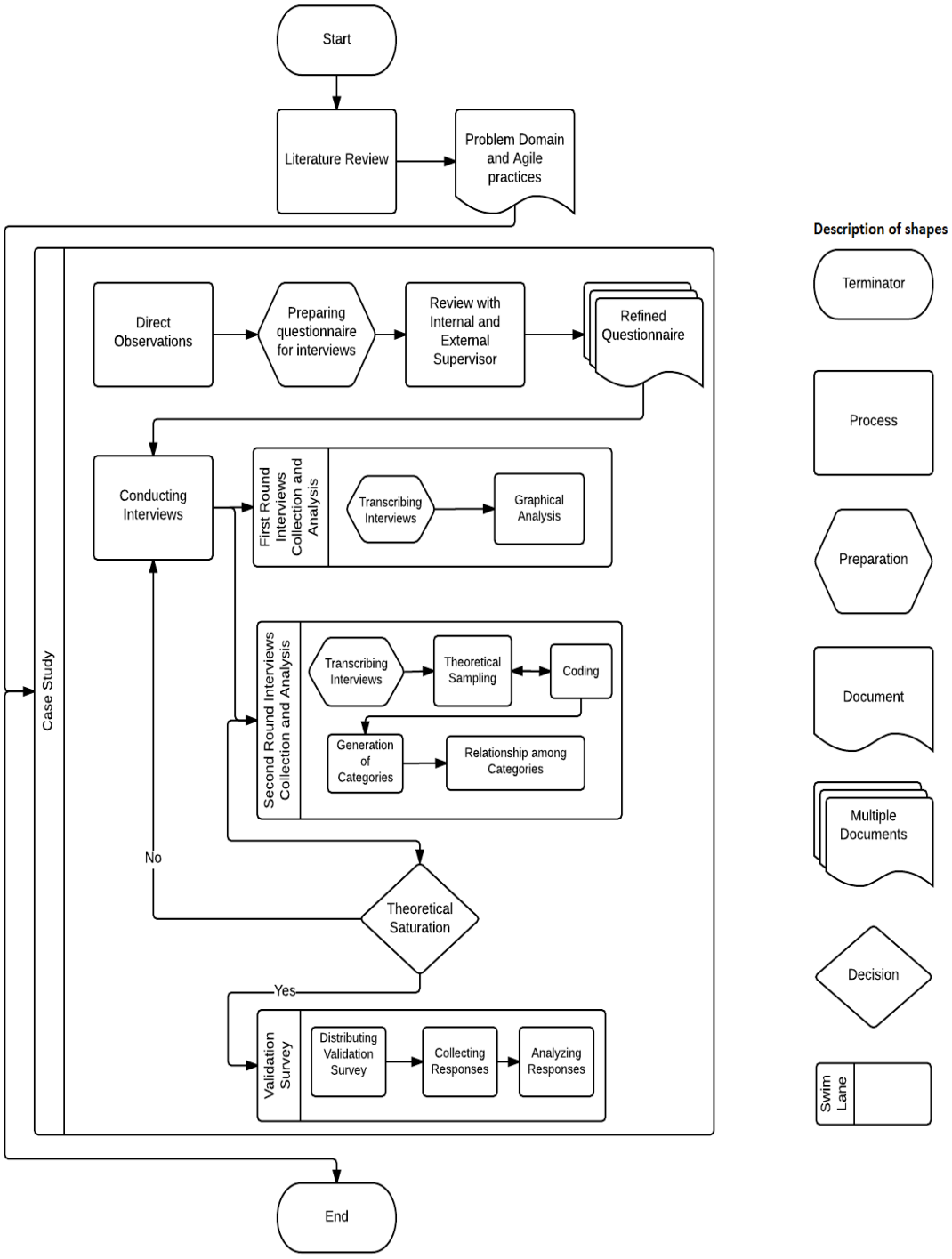


Figure 3.1: Research Design

3.1 Case Study Design

The research design is explained in detail in this section.

3.1.1 Case and its Unit of Analysis

Capgemini, is a large-scale French based software company, that has many subsidiaries spread across the world. Its Indian subsidiary is selected to conduct this study, since the researcher had good connections with Vice-President HR of Capgemini. So upon the request of the Vice-President HR at Capgemini, associate director quality at Capgemini, has accepted to mentor this study to closure. As defined by Yin in [47], case is something which depicts *"contemporary phenomenon in its real-life context"*, so in this study the case would be a large-scale software company, i.e. Capgemini and the unit of analysis would be different agile projects mostly involving post-delivery activities, i.e. software maintenance.

3.1.2 Case Study Protocol

Study protocol is developed to conduct the case study. It includes two parts: Data collection protocol and analysis protocol. Each protocol is explained in detail in following sections.

3.1.2.1 Data Collection Protocol

3.1.2.1.1 Direct Observations

Direct observation is considered to be one of the data collection methods in this study. The main concept behind it is to discover the first-hand behavior of participants involved in the activity of direct observation, which might not be discovered using other methods in a qualitative study [47]. In addition the information gathered from direct observations was also utilized in framing interview questionnaire. Motivated by these reasons this study considers direct observation as one of the data collection method. The main motive of this research is to study Agile practices adoption during software maintenance, hence different direct observations are made, wherein the researcher doesn't gets involved in the activity which is being observed [48].

3.1.2.1.2 Selection of interviewees

This case study is focused towards adoption of agile practices during software maintenance. During the case study along with direct observations semi-structured interviews are conducted within different departments working on different projects and hence allowing to set the focus of this case study at department and project levels both. Various project specific stakeholders involved during software maintenance in different projects are asked to participate in this case study. Information

about different projects at Capgemini was provided by external supervisor and also by an agile coach, i.e. snowball sampling was employed in the selection of sample [49]. The information provided by external supervisor and agile coach at Capgemini, contained project name and contact person (mostly project manager). In total there were 11 projects which adopted agile during software maintenance. An Invitation Letter (in appendix 1) was sent through email to each of those projects to participate in the case study (i.e. to contact person). Received positive response from 5 projects. Preference for selecting participants for interviews was all in-house stakeholders (or project roles) involved during maintenance, i.e. Project managers, developers, architects, testers and team leads.

3.1.2.1.3 Interview Design

Interviews conducted during this study were semi-structured interviews. Both close-ended questions and open-ended questions were used during the interviews. Close-ended aimed at gaining insights into agile practices that are adopted during maintenance, whereas open-ended were used to understand different advantages and disadvantages of adopting agile practices during software maintenance. Since, each questionnaire (i.e. close-ended and open-ended) aims to answer different research questions in this study, two different rounds of interviews are conducted. This ensures to use different analysis methods for both rounds of interviews. First round is conducted among project managers of each project using a close-ended questionnaire, to identify the usage of agile practices during software maintenance. Second round is conducted among different project specific stakeholders (including project managers from the first round) using open-ended questionnaire, to identify advantages and disadvantages of adopting agile practices during software maintenance. The design of First and Second round of interview is presented below:

3.1.2.1.3.1 Design of First Round Interviews

First round of interviews are conducted in three phases and these are as follows:
Phase 1: Initial Set-up- The interview goals and objectives are presented in brief.
Phase 2: Enlisting Agile Practices- A list of agile practices are presented to interview participant.
Phase 3: Rating of Usage- After presenting the list, interview participant were asked to rate upon the usage of the agile practice in their projects (Ratings used: *"Not Used"*, *"Low Usage"*, *"Medium Usage"* and *"High Usage"*).

3.1.2.1.3.2 Design of Second Round Interviews

Second round of interview is also conducted in three phases and these are as follows:
Phase 1: Initial Set-up- The interview goals and objectives are presented in

brief.
Phase 2: General Information- In this phase general information about the participant is collected, which includes:

- Experience with Capegemini in Years.
- Role presently working in.
- Responsibilities in the present role.
- Experience with other roles.
- Responsibilities in those roles
- Experience with other methodologies (agile, waterfall etc.)

Phase 3: Impact of Agile practices- Interview Participants in this phase are interviewed about the advantages and disadvantages of adopting agile practices during software maintenance, using a open-ended questionnaire.

3.1.2.1.4 Formulation of Interview Questions

This section describes about the process which is followed in the formulation of both the questionnaire (close ended and open-ended). Formulation of both close and open-ended questionnaires involves the certain steps. There are three steps and one deliverable. The following steps in the formulations of questionnaires (close and open) is presented below:

Step1: Questions formulation.
The first step is to formulate the questions in the questionnaire.

For close-ended questionnaire, questions are formulated by referring to literature. Different agile practices adopted during software maintenance are enlisted from existing empirical studies and it is listed in table 2.2 of section 2.3.2. The same set of agile practices were used for the formulation of close-ended questionnaire. In the questionnaire those practices were further categorized into different section. A snap shot of questionnaire is shown in appendix 2.

For Open-ended questionnaire, questions are formulated by referring to three major sources and these are: Literature review, Analysis of previous interview Transcripts and Direct observations. Open-ended questionnaire aims to investigate the impact of agile practices during software maintenance. So literature is referred to explore different advantages and disadvantages of adopting agile during maintenance from existing studies. The gathered information about advantages and disadvantages using literature are formulated as questions and are added to the open-ended questionnaire. Doing this way ensures the researcher to know whether those advantages and disadvantages (gathered using literature) are

perceived at Capgemini also. Direct observations is other source for formulation of questions in open-ended questionnaire. Initially, conducted before the interviews. If anything was unclear during the direct observations, it was formulated into a question for further clarification. The motivation behind the inclusion of data collected during observations was to improve the understandability of the researcher in the context of the study. Apart from literature review and direct observation, analysis of previous interview transcripts are also used as additional source for question formulation and refining of open-ended questionnaire.

Step2: Frame/Re-frame Questions for Interview.
In this step researcher tries to frame questions using the data from previous step. In case of close-ended questionnaire different agile practices are identified in different categories, such as requirement and design, implementation, testing, process and organization. Similarly, open-ended questionnaire is formulated which were mostly stakeholder specific, such as questionnaire for testers, developers, managers.

Step3: Review from external and internal supervisor.
After formulation of both the questionnaires, the draft is sent to both external and internal supervisor for the revision. Reviews received from both supervisor are considered and revisions to it are again drafted, which means step 2 and step 3 are in loop. If no more reviews to the draft are received then it is considered as exit-criteria for the loop. So at the end of this step a deliverable is produced i.e. interview questionnaire document, which contains both close-ended questionnaire (used in 3.1.2.1.3.1) and open-ended questionnaire (used in 3.1.2.1.3.2).

3.1.2.1.5 Transcription

Transcription of first and second round of interviews was different. First round of interviews was done through close-ended questionnaire, hence participants involved in first round were directly given a hard copy of questionnaire and were asked to answer them. After collecting the responses for close-ended questionnaire, researcher manually entered the answers into word documents for further analysis. Second round of interviews was done using open-ended questionnaire. So asking the participants to write up the answers for open-ended questions is time taking process hence, researcher used field-notes and audio-recorded the interviews. As few second round interviews were not face-to-face audio recording was difficult, during which field notes facilitated the researcher to collect the views of interview participants. Field notes used during second round of interviews mostly contained, date, time, place, name of the participant, participant role, participant project code. Transcribing of second round of interviews was done at the end of the day, but interviews which were not audio-recorded were transcribed immediately after the interview. Word to word transcription was

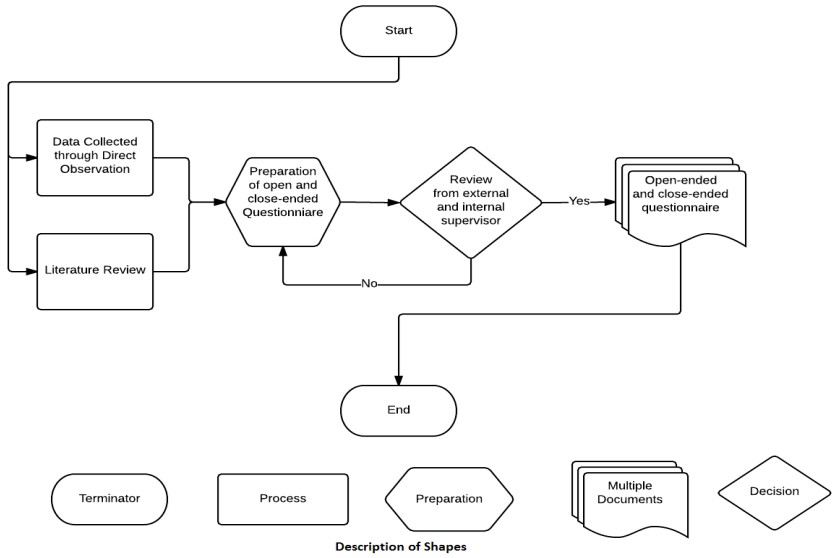


Figure 3.2: Flow Chart for Questions Formulation

used in this study using a transcription tool called *"ExpressScribe"*.

3.1.2.2 Analysis Protocol

As the aims and objectives of first round and second round of interviews differ, hence analysis of those two rounds of interviews also differs. Apart from aims and objectives, even the type of questionnarie used, population of both the interviews differ, hence researcher used different analysis protocol for both the rounds of interviews. Initially, first round of interviews were planned to be analyzed by statistics (mean and standard deviation). But upon researcher consultation with internal supervisor, first round of interviews were analyzed using visual analysis (using Graphs). Since the data points (i.e. population of close-ended questionnaire) were very less to be analyzed using statistics (mean and standard deviation). Second round of interviews, which includes open-ended questionnaire was analyzed using Grounded theory. Glaser and Strauss initially launched GT, but both the authors have separated and developed their own versions of GT. Strauss and Corbin version of GT assumes that research questions are pre-set and researchers using this version of GT enters the study with some exposure to literature, whereas Glaser version of GT assumes that research questions and problem are framed while coding begins. As the research questions are pre-set in this study and researcher has used literature review to frame the research problem/domain, therefore it encourages to use Strauss and Corbin version of GT [50]. Besides many studies [51, 52, 53, 54] have proved that GT can be used in the field of software engineering. Hence, this study used GT for analyzing data collected

during seocnd round of interviews. Different steps followed during analysis of second round of interviews are given below. **Note: In the following steps, the word interview means second round interviews.**

3.1.2.2.1 Post-Interviews (Theoretical Sampling)

This is the first step in the analysis protocol of interviews. This is simultaneously done along with data collection. The purpose of this step is to refine the open-ended questionnaire and also to transcribe the interviews. Thorough reading of interview transcripts is done after each interview. Express Scribe, a transcription software was used to transcribe the audio-recorded interviews. After each interview was transcribed using Express Scribe it was then imported to word processor. This is done to correct few grammatical mistakes and spellings during the transcription with Express Scribe. After importing and correcting grammatically, these interviews were thoroughly read by the researcher. Memoing is one of the key step during theoretical sampling, as it helps the researcher to capture few findings, these findings might be new questions or theoretical notes (ideas or thoughts). Conducting interviews is an ongoing process during GT, it is important to always capture and record such findings, so Comments feature of the word processor was used to facilitate in memoing during GT. In this way the questionnaire used in the interviews as part of data collection is refined, hence aligning the questionnaire to the emergent results.

3.1.2.2.2 Codification (Coding)

After importing the transcripts to word processor, those word files are again imported to MAXQDA, a data analysis tool [55]. Using this tool different codes are created manually. The imported word files are thoroughly read line by line to allocate codes to the text. Codes are labels which are assigned to some part of text in the interview transcripts. Those texts addresses certain issues related to the research questions in this study. So any text in the transcripts which address a topic is tagged to a code. As stated earlier, a code is short representation of texts of all interview transcripts which specifically talks about a topic. For example, if any interviewee says that comprehensibility of the system is improved using agile methods, then a code *"Comprehensibility"* is created in MAXQDA. After creation of the code in MAXQDA, the text (in the interviewee transcripts) which says that comprehensibility of the system is improved using agile methods, is tagged to *"Comprehensibility"* code. Similarly, if any other interviewees speak about this topic then the text in those interviewee transcripts is selected and is tagged to that code. So it can be concluded that a code is a collection of responses of interviewees which represent particularly about a topic. This way code *"Comprehensibility"* collects all the views of interviewees on that topic. By doing this it becomes easy to analyze responses of interviewees on a particular code (Concept).

Features such as activating imported interview transcript documents and codes helps to retrieve the text tagged for a particular code in MAXQDA. For example, if a particular code is to be analyzed, then all interview transcript documents which are imported into MAXQDA and a code (which is to be analyzed) is activated. Then MAXQDA retrieves the text from the activated documents, which are assigned to that particular activated code. During analysis of a particular code, features of MAXQDA such as memoing helps to create memos against a particular code.

To identify text from interviewee transcripts for a particular code, MAXDICTIO feature of the MAXQDA is used. For example, a code *"Comprehensibility"* is created. Now to check if other interviewees spoke about this topic, MAXDICTIO is used. Code word is added to MAXDICTIO Dictionary; in this case let us say *"Comprehensibility"* is added into MAXDICTIO Dictionary. Search *"Only Dictionary Words"*, sub-feature in MAXDICTIO is used. Wherein the words added to dictionary are searched with *"exact matches"* all along the imported documents. All the documents which contain that particular word are retrieved. After which each document is looked into to see what exactly was the context. If the context was related to the code *"Comprehensibility"* then that particular text was assigned to that code. It is similarly done for other codes, for easy identification of the context in different interview transcription document. Using this feature constant comparison of GT is achieved.

When there are no new codes created/generated, then it is considered to be exit-criteria for this step. The point is called Coding Saturation/Theoretical Saturation. Both Post-transcription and coding steps comes to an end when it reaches saturation point.

3.1.2.2.3 Generation of Categories by code merging (Axial Coding)

Constant comparison of codes is used to generate categories during data analysis. Categories and sub-categories are higher level abstraction of codes. All the similar codes are gathered into one sub-category and all the similar sub-categories are gathered into one category. During coding process, the new codes generated are constantly compared with already generated codes, this is done to analyze whether there is any difference between the newly generated code and already present code. Also, constant comparison of codes helps to identify common characteristics among them, which leads to generation of sub-categories and categories. During this step different categories of codes have emerged. All codes which referred to advantages of agile practices during software maintenance are grouped under advantages. In the same way codes which referred to disadvantages are categorized into disadvantages.

3.1.2.2.4 Relationships among Categories (Selective Coding)

During this phase of data analysis relationships among the categories are ex-

plored. The purpose behind this phase is to generate a theory by exploring the relationship among the categories, which in turn produces a core category around which a theory can be generated [50]. For instance, what are advantages and disadvantages of adopting those agile practices during software maintenance? There are two categories for this question, one would be *Advantage* and other would be *Disadvantage*. On exploring the relationships between each category, it is found that the core category would be *Software maintenance*, since all the other categories explained it.

3.1.2.2.5 Validation Survey

Validation of data extracted during data analysis is the last part of the Analysis protocol. The purpose of this step is to mitigate construct validity (further explained in section 6.3.1) and also, to improve the generalisability of data extracted during data analysis. A survey with closed-ended questions (mostly) is formulated by using the codes which were extracted during data analysis. This Survey is called as Validation survey, as this survey helps to validate the data extracted during data analysis. It is distributed among interviewee participants and also among other practitioners of software maintenance and agile outside the case company. Validation survey is created using an web-application called *"survio.com"*. Apart from creating the survey, this tool helps to collect the responses of individuals who participated in the survey and provides excellent graphs for analysis. Analyzing the data collected using close-ended survey questionnaire is done through statistics, i.e. by calculating mean and standard deviation [56]. Interview Participants were invited using email (the content of the email is presented in appendix 4). Convenience sampling was used in selecting the sample space who would take the survey [49]. The survey was posted in different forums, such as *"LinkedIn Groups"*, which helped in finding survey participants among other practitioners of software maintenance and agile. These participants were also requested to circulate the survey questionnaire among their network. This way snowball sampling technique was also incorporated in selecting the sample space [49]. The purpose to involve practitioners outside the case company was to increase the generalisability of results.

Chapter 4

Results

4.1 Agile Adoption for Software Maintenance at Capgemini

All projects which adopt agile during software maintenance follow a agile model as defined in Capgemini Quality Management System (QMS).

The agile model adopted for software maintenance at Capgemini is depicted in the figure 4.1. The approach towards product owner role in Capgemini is bit different, as that role is taken up by different representatives from each department of project, such as project management, quality assurance and development team. These representatives are called as Change Control Board (CCB), as it would be difficult to provide just one role to that group of representatives. CCB was mainly responsible for prioritizing different tasks for development team (developers and Quality Assurance) and act as customer proxy. Also, CCB helps development teams by creating product backlog list and decides different themes for a release. As seen in the figure 4.1 different tasks or work requests in maintenance project are mainly enhancements, architecture changes or defect fixing. But these tasks are represented as ticket. A ticket represents units of work. Different projects chose different tracking systems to track those work requests (tickets). If a customer has work request (either enhancement, defect fixing or architecture change), he/she reports that particular work request as a ticket in the tracking system, by which the project teams get notified. The team organization in each project is similar. It consists of Quality Assurance (QA) team and developers team. QA team is further divided into integrated QA with Developers and QA for regression. Summary of different phases of agile adopted during software maintenance is presented below:

- **Discovery Phase (Planning the Release)-** Planning for the release begins with looking at ticket tracking system used by projects at Capgemini. All the tickets which show the status as open, are considered to be product backlog list. From this initial list, a secondary list is created, based on release criteria. This secondary list (release backlog) is given to development

team (both developers and QA) to provide initial estimations. Based on those estimations, different iterations are planned for a release. Apart from that, CCB uses those estimations from development to prioritize different tickets in release backlog list. Taking a step forward, development team provides further specific estimations to decide upon the count of tickets in an iteration. Based on those specific estimation and prioritization, the count of tickets in an iteration are planned.

- **Iteration (Implementation Cycle)-** After the selection of tickets for an iteration, developers pair with testers to analyze each ticket. During analysis developers and testers upon consulting with specific customer (person at customer side who has raised the ticket) decide upon the impact of implementing the ticket and take a decision either to accept it for further implementation or reject it. If the ticket gets accepted, then developer would write unit test case for it and implements the solution. After the implementation of solution, a code review is done by development team representative. During this time testers from Dev-integrated QA would build automated test cases, and upon receiving it would test it. After the initial testing by testers from Dev-integrated QA, it is then given to testers from QA regression for further testing. Testing it in Dev-integrated QA environment and QA regression environment would provide confidence to the team before the release. Defects found during testing by Dev-integrated QA and QA regression would be instantly fixed by Developers, if the fixing of defect doesn't exceed more than 4 hours, else it is moved back to product backlog list.

- **Post-Release-** When different tasks in a release are completed by the team (Developers and QA), they then call for a meeting for release review. During the meeting, team discusses on different issues that have occurred during the release. The main goal of the meeting is to improve the development process and software quality, by addressing those issues. Before meeting, representatives from each department (Developers and QA) collects the release review from his/her team members. This is done to shorten the meeting and not prolong it for hours. Representatives from each department presents those reviews in the meeting and further discuss about it. Mostly, those reviews are issues faced by different team members. Also project management representative would list out all the names who have performed very well in the release. Information about performance is given by each department representative to project management representative before the meeting. Tracking of performance is done by different tools as defined by QMS. At the end of meeting team also decides about the place for celebration and allocates budget for it.

Agile adoption during software maintenance is not perfectly defined by QMS at Capgemini. Hence, many projects at Capgemini which deal with software maintenance don't risk to adopt agile during software maintenance, as they have little knowledge about the results of adopting different agile practices during software maintenance. To provide support to software maintenance practitioners in visualizing the advantages and disadvantages in adopting different agile practices, interviews were conducted. The interviews were with 14 Software maintenance practitioners (from different projects) who have adopted agile during maintenance at Capgemini, Mumbai.

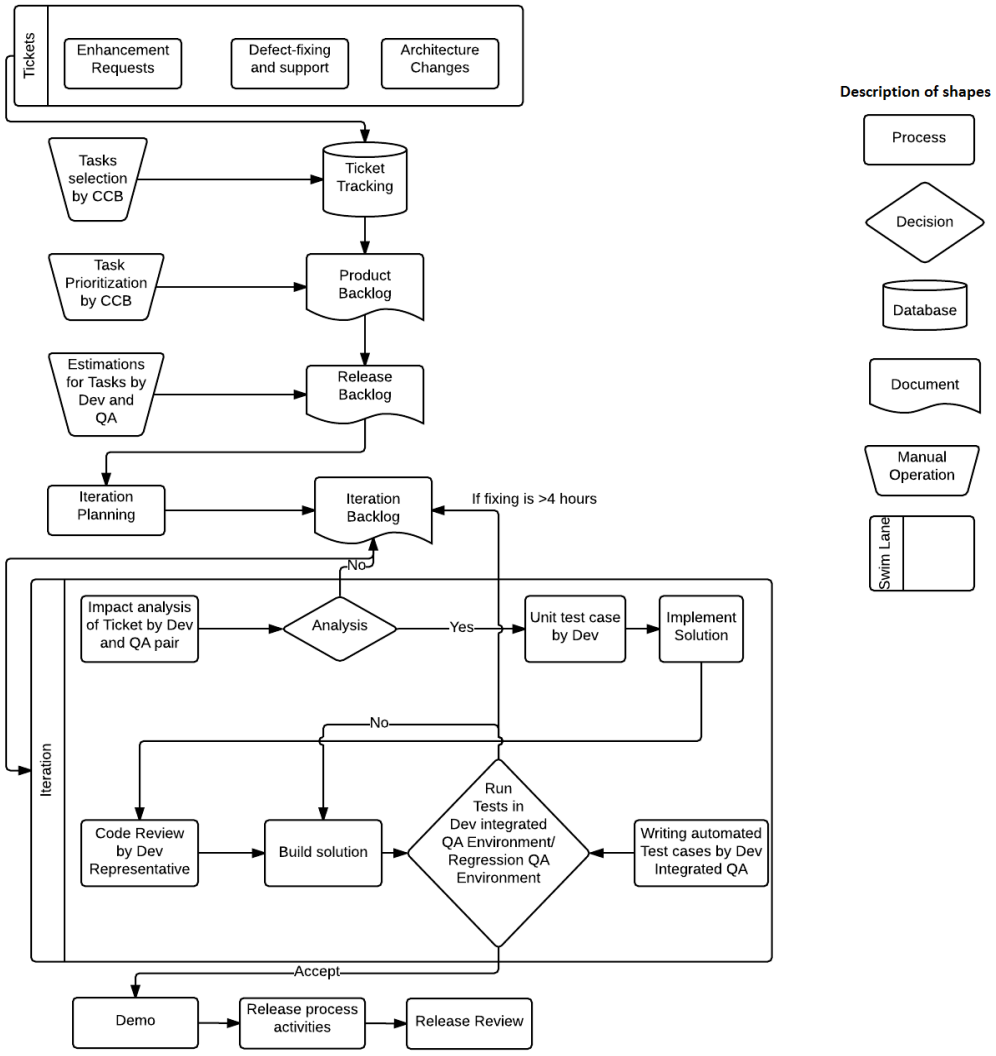


Figure 4.1: Agile Model for Software Maintenance at Capgemini

4.2 Summary of First Round Interviews

Project Code	**Domain**	**Place**	**Role**	**Maintenance Type**
P1	Insurance	Mumbai	Project Manager	Perfective and Corrective
P2	Insurance	Mumbai	Service Delivery Manager	Perfective and Corrective
P3	Staffing	Mumbai	Project Manager	Adaptive and Preventive
P4	Finance	Hyderabad	Scrum Master	Perfective
P5	Insurance	Mumbai	Service Delivery Manager	Perfective and Corrective

Table 4.1: Overview of the First Round Interview Participants

Initially, an invitation (as seen in appendix 1) is sent to different projects which have adopted agile during software maintenance at Capgemini, to conduct case study. Five projects have accepted the invitation to participate in the case study as stated in section 3.1.2.1.2. After accepting the invitation, the contact person from each project were requested to participate in the first round of interviews. As those contact persons from each project were at managerial position, it also ensured correct population for the first round of interviews. Since first round of interview required good experience and also a role which defines the maintenance process of the team. Expect one interview during first round, all the other interviews were done face to face. Interview which was not face to face, was conducted on-call and the questionnaire was shared using office communicator.

4.3 Results of First Round Interviews

In this round of interview, participants were given a close-ended questionnaire. That close-ended questionnaire contained questions about the usage of different agile practices retrieved from literature. To answer the questionnaire about the usage of different agile practices in their project, they were asked to choose in between a given range. (*"Not Used"*, *"Low Usage"*, *"Medium Usage"* and *"High Usage"*). The answers to the questionnaire were collected using a hard-copy and were entered into a word processor manually. After manually transcribing those interviews using word processor, MAXQDA was used to make visual analysis

of those transcripts. Close-ended questionnaire was categorized into different sections, based on those sections results of the interviews are presented below:

Agile Practice	**Not Used**	**Low Usage**	**Medium Usage**	**High Usage**
Product Backlog	1	0	3	1
User Stories	0	0	1	4
Planning Game	3	0	1	1
Task Prioritization	0	0	2	3
System Metaphor	5	0	0	0

Table 4.2: Agile Practices for Requirement Analysis and Design

The above table 4.2 presents the responses of first round interview participants about the usage of agile practices for requirement analysis and design.

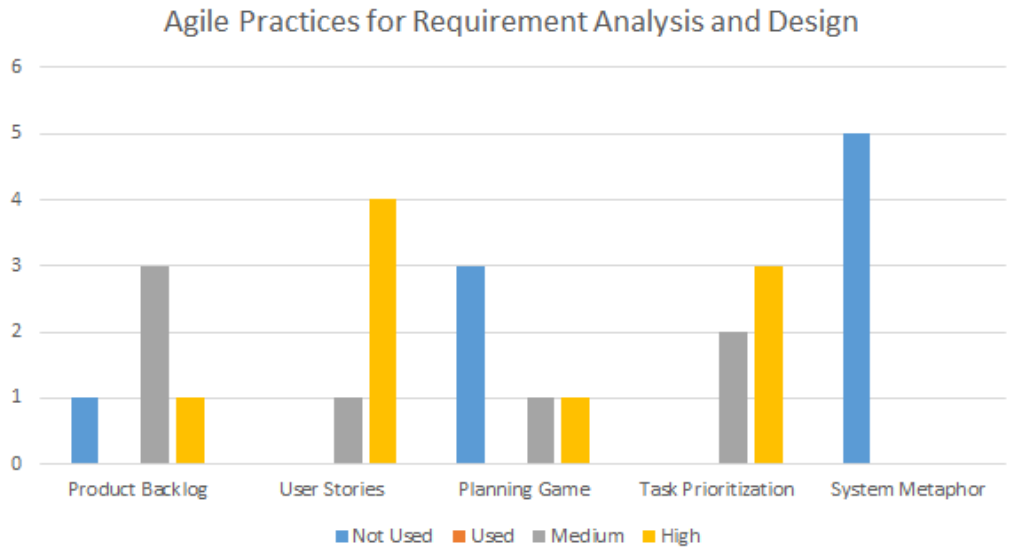


Figure 4.2: Statistics of table 4.2

From the responses of first round interview participants as seen in figure 4.2, it can be concluded that agile practices such as *"Product Backlog", "User stories"* and *"Task prioritization"* are more often adopted and used, whereas *"System Metaphor"* and *"Planning Game"* are less adopted and used during requirement analysis and design.

Agile Practice	**Not Used**	**Low Usage**	**Medium Usage**	**High Usage**
Coding Standards	0	1	2	2
Test Driven Development	1	3	1	0
Pair-Programming	3	1	0	1
Refactoring	1	0	3	1
Collective Code Ownership	0	0	0	5
Daily Builds	1	1	1	2
Continuous Integration	1	1	2	1
Code Reviews	0	0	1	4
Bug Tracking	0	0	1	4
Small Releases	0	0	3	2

Table 4.3: Agile Practices for Implementation

The above table 4.3 presents the responses of first round interview participants about the usage of agile practices for Implementation.

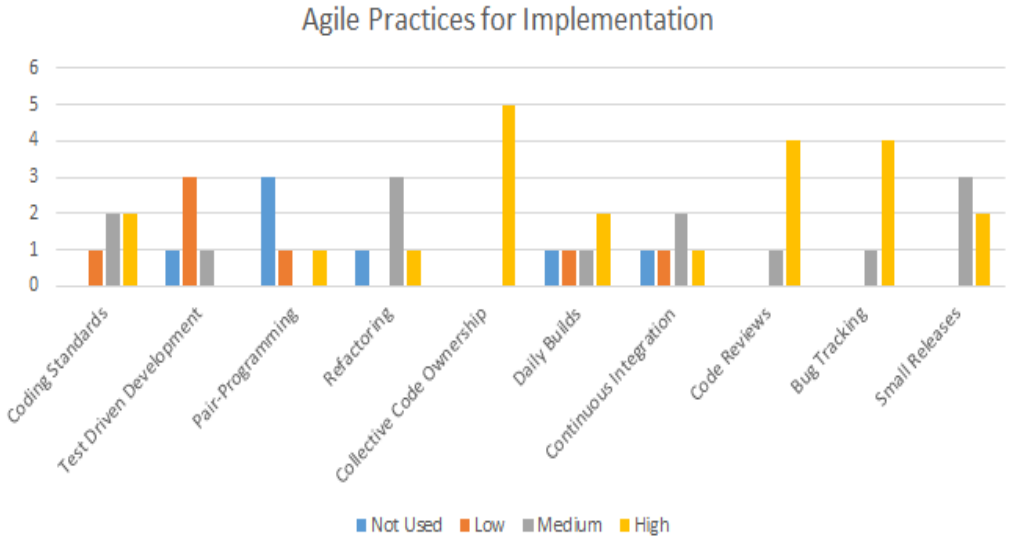


Figure 4.3: Statistics of table 4.3

From the responses of first round interview participants as seen in figure 4.3, it can be concluded that agile practices such as *"Coding Standards", "Pair-Programming", "Refactoring", "Collective Code Ownership", "Daily Builds", "Continuous Integration", "Code Reviews", "Bug Tracking"* and *"Small Releases"*are

more often adopted. But the usage of *"Collective Code Ownership", "Code Reviews"* and *"Bug Tracking system"* is more. Whereas practices such as *"Test Driven Development"* and *"Pair-Programming"* are less/not adopted and used during implementation.

Agile Practice	**Not Used**	**Low Usage**	**Medium Usage**	**High Usage**
Unit Testing	0	0	0	5
Automated Testing	3	0	1	1
Acceptance Test	0	0	2	3

Table 4.4: Agile Practices for Testing

The above table 4.4 presents the responses of first round interview participants about the usage of agile practices for testing

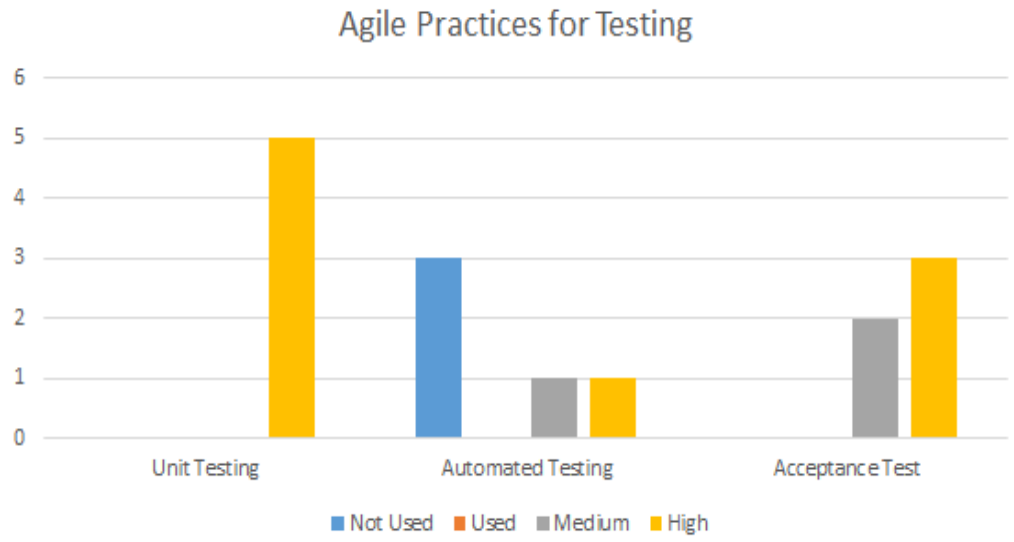


Figure 4.4: Statistics of table 4.4

From the responses of first round interview participants as seen in figure 4.4, it can be concluded that agile practices such as *"Unit Testing"* and *"Acceptance Testing"* are more often adopted. In addition usage of *"Unit Testing"* and *"Acceptance Testing"* is also high. Practices such as *"Automated Testing"* are less adopted and not used during testing.

Agile Practice	**Not Used**	**Low Usage**	**Medium Usage**	**High Usage**
Short Iteration	0	0	0	5
Release Planning	0	1	0	4
Iteration Planning	0	1	1	3
Task Board	1	0	0	4
Daily Stand-up Meeting	0	0	1	4
Velocity	1	1	0	3
Demo	1	0	0	4
Retrospective	0	2	1	2

Table 4.5: Process Related Agile Practices

The above table 4.5 presents the responses of first round interview participants about the usage of agile practices for processes.

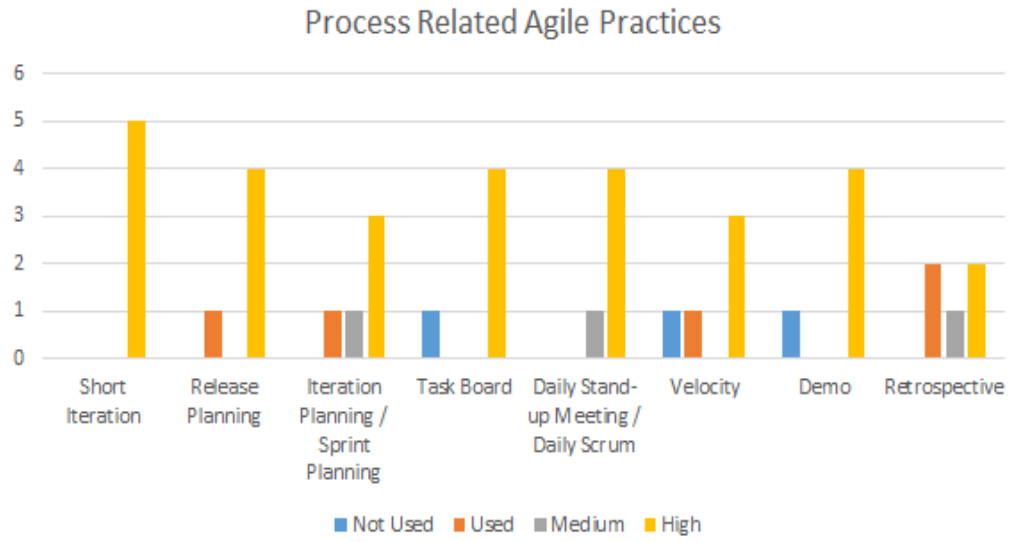


Figure 4.5: Statistics of table 4.5

From the responses of first round interview participants as seen in figure 4.5, it can be concluded that agile practices such as *"Short Iteration", "Release Planning", "Iteration Planning", "Task Board", "Daily Stand-up Meeting", "Velocity", "Demos"* and *"Retrospective"* are more often adopted. But practices such as *"Short iteration", Release Planning", "Task Board", "Daily Stand-up Meetings"* and *"Demos"* are highly used. *"Retrospectives"* are adopted and but not used to high extent.

Agile Practice	**Not Used**	**Low Usage**	**Medium Usage**	**High Usage**
Small Team	0	1	1	3
Co-located Team	2	0	1	2
On-Site Customer	0	0	1	4
Scrum Master	0	1	0	4

Table 4.6: Agile Practices for Team Organization

The above table 4.6 presents the responses of first round interview participants about the usage of agile practices for team organization.

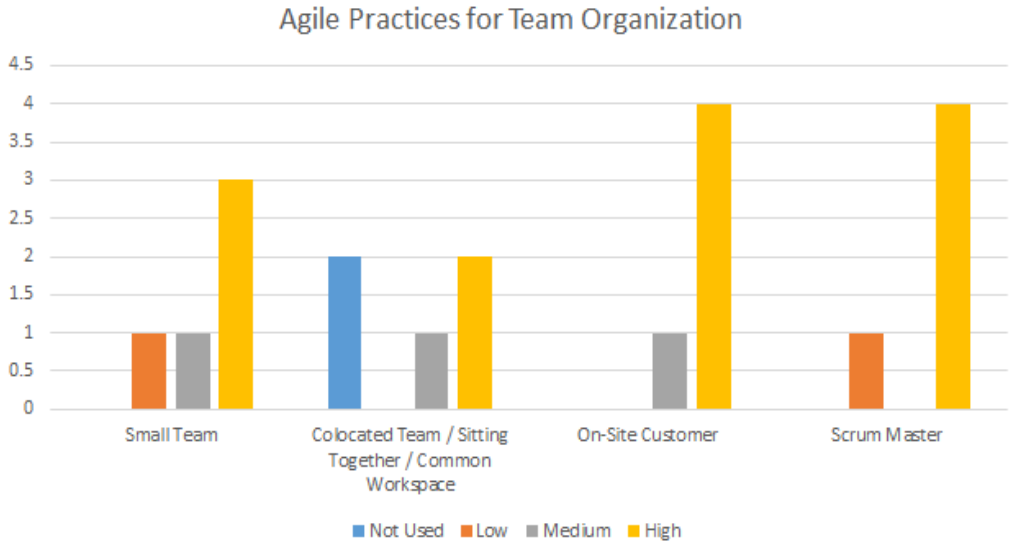


Figure 4.6: Statistics of table 4.6

From the responses of first round interview participants as seen in figure 4.6, it can be concluded that agile practices for team organization such as *"Small Teams", "On-site Customer"* and *"Scrum Master"* are more often adopted for team organization. But Roles such as *"On-site Customer"* and *"Scrum Master"* are highly created in the teams. *"Co-located Team"* is less adopted and if adopted it is practiced to high extent.

4.4 Summary of Second Round Interviews

Around 14 interviews were conducted among 5 projects at Capegemini during second round. Four projects were present at researcher working place and one project was situated away from researcher working place. Totally 11 interviews

Project Code	**Domain**	**Place**	**Number of Interviews**	**Maintenance Type**	**Agile Methodology**
P1	Insurance	Mumbai	2	Perfective and Corrective	Extreme Programming
P2	Insurance	Mumbai	6	Perfective and Corrective	Scrum
P3	Staffing	Mumbai	1	Adaptive and Preventive	Scrum
P4	Finance	Hyderabad	2	Perfective	Scrum
P5	Insurance	Mumbai	2	Perfective and Corrective	Hybrid

Table 4.7: Overview of the Project

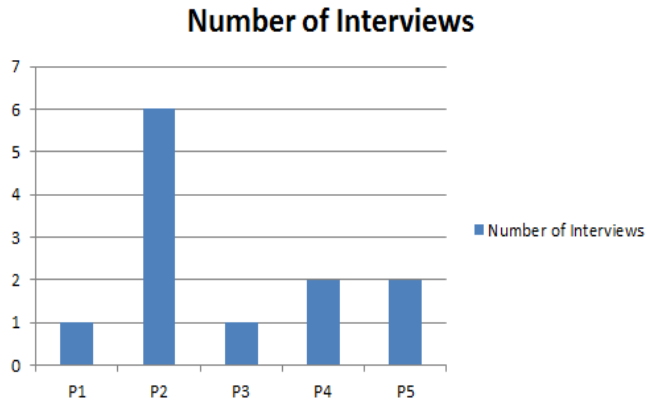


Figure 4.7: Number of Interviews in Each Project

were conducted face-to-face and were audio-recorded, whereas 3 interviews were conducted on-call. The overview of the projects is present in the table 4.7.

The second round interview was conducted with different objectives. Firstly, to know advantages and disadvantages of adopting agile practices during software maintenance. Secondly, to refine the open-ended questions used in phase 3 of interview design for next interview participants. Second round of interviews was conducted using a open-ended questionnaire as shown in appendix 3. After few interviews, open-ended questions used in phase 3 of second round interview are refined and were added into questionnaire in appendix 3. Around 7 interviews in second round interviews were conducted with initial open-ended questionnaire and the rest were conducted using refined questionnaire. The overview of the participants is given in table 4.8

Role	Project Code	Experience	Audio Recorded	Duration (mins)	Rounds
Scrum Master	P4	2	No	42	1
Testing Lead	P2	2.5	No	30	1
SDM	P2	8	Yes	80	2
Architect	P2	4.5	Yes	45	1
Developer	P2	4.5	Yes	104	2
Tech Lead	P2	5	Yes	30	1
Developer	P1	5.5	Yes	33	1
Quality	P4	4	No	28	1
Tester	P5	4	Yes	30	1
Project Manager	P3	11	Yes	30	1
Project Manager	P1	8	Yes	30	1
Developer	P5	5	Yes	30	1

Table 4.8: Overview of the Interview Participants

4.5 Results of Second Round Interviews

In this section data collected during second round of interviews is presented. The following subsections are different steps which are followed as a part of grounded theory used in analysis of second round interviews.

4.5.1 Transcription

In total 14 interviews were conducted, of which 11 interviews were audio recorded. Interviews which are not audio recorded are transcribed using Field notes as shown in figure 4.8.

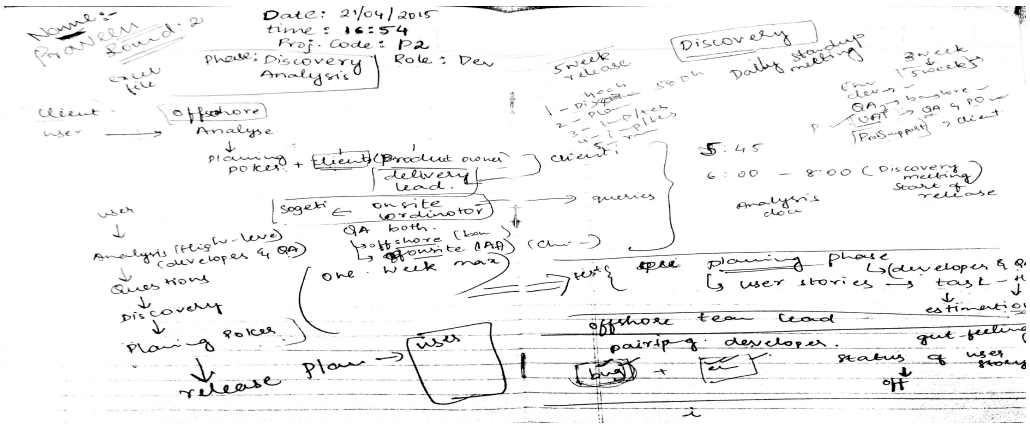


Figure 4.8: Field Notes

Audio recorded interviews were transcribed word to word. Express Scribe facilitated in transcribing audio recorded interviews, but lack of spell check and grammar check made it difficult. So after transcribing using Express Scribe, transcriptions were imported to Word processor, which had spell check and grammar check features in it. Features in Express Scribe such as playing the audio file at variable speed helped in transcription, figure 4.9 provides an example.

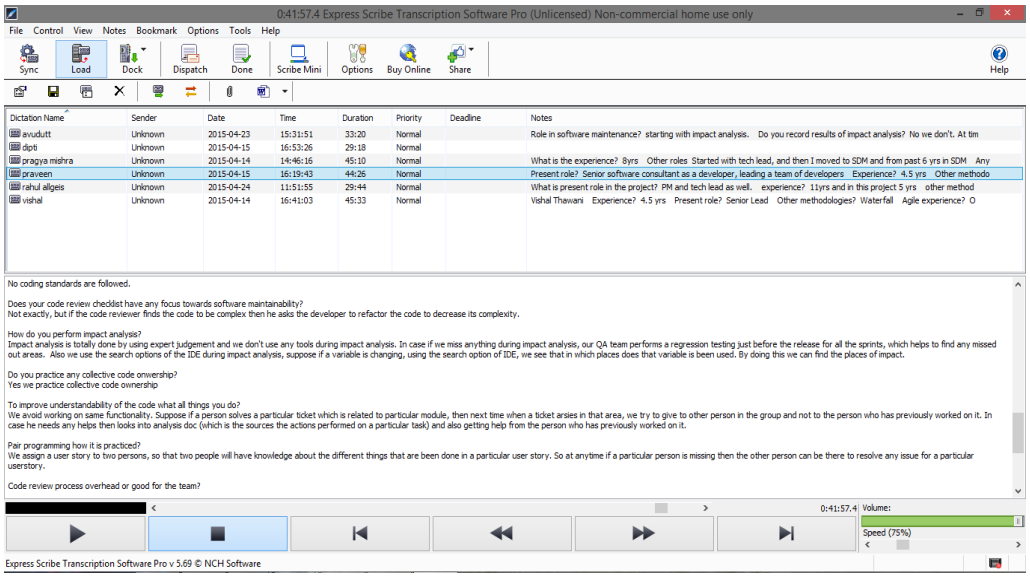


Figure 4.9: Screen Shot for Express Scribe

4.5.2 Post-Interviews (Theoretical Sampling)

Example for Post-Transcription is shown in figure 4.10 where in spell check of word processor is used to correct the spelling mistakes made during transcription and comment feature is used to record the findings in the transcripts and of-course it is used to record any new questions. Word files are imported to MAXQDA as shown in figure 8.1. After Importing the word files to MAXQDA, transcriptions are read to identify any codes. In case researcher identifies a code for any context in transcription, then a code is created or checks if there are any already existing codes in code system. To tag a text from transcription, author highlights the text from transcription, activates the code to be tagged and tags it to the activated respective code, this whole process can be seen in appendix 7.

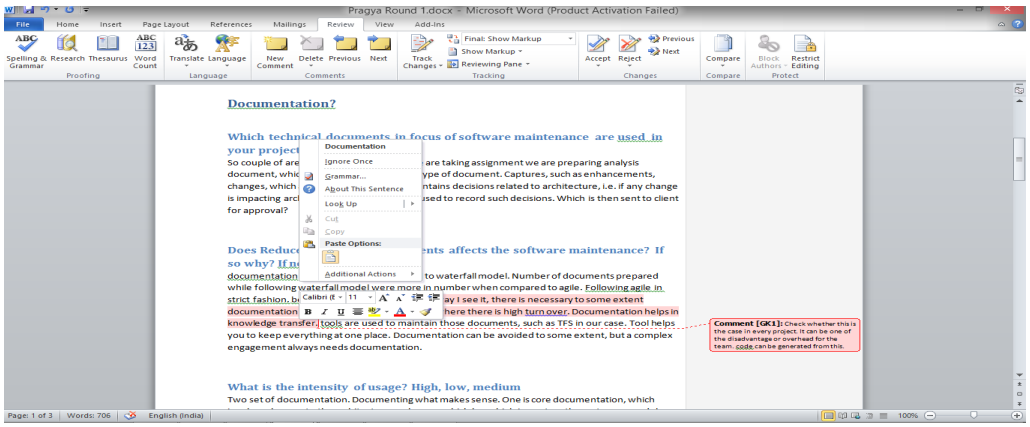


Figure 4.10: Screen Shot for analysis using Word Processor

4.5.3 Codification (Coding)

After transcription of each interview, a thorough reading of the interview transcripts is done to identify codes in the transcripts. Any keywords from the transcripts which are relevant to the research questions is considered to be a code. Apart from identifying the codes in interview transcripts, literature review and word frequency also facilitated in identifying some codes. These codes are added into the code system of MAXQDA. To identify the context of the code in each interview transcripts, MAXDICTIO one of the feature of MAXQDA was used. Figures below show how the MAXDICTIO helped in identifying context from interviewee transcripts for an existing code i.e. *"comprehensive documentation"*.

Step 1: Add code words to Dictionary eg. *"comprehensive"* and *"documentation"*, as shown in the figure 4.11

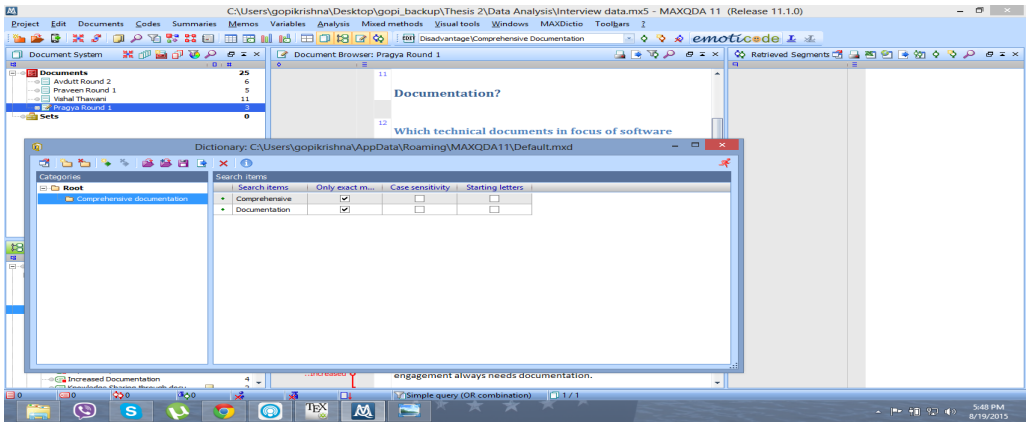


Figure 4.11: Screen Shot for Words adding into MAXQDA Dictionary

Step 2: Identifying the particular word in the interviewee transcripts

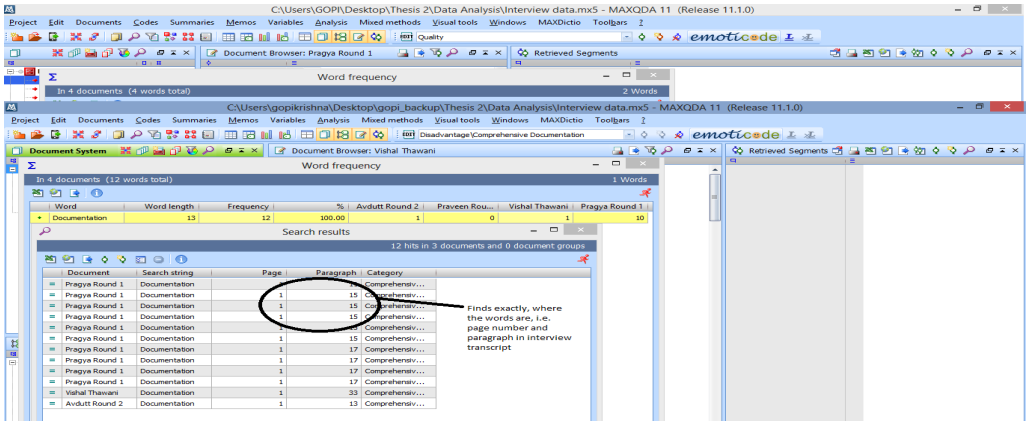


Figure 4.12: Screen Shot for Words searching in interview transcripts using MAXQDA Dictionary

By following those two steps, context for each and every code from the transcripts is retrieved. After retrieving the context for each code from every transcripts, a thorough reading is once again done to decide upon the assignment of context to the selected code from a particular interview and also from pre-existing transcripts. Codification during data analysis is in continuous loop until each and every interview was transcribed, this can be understood from the figure shown below. It should be noted that the codes added to MAXDICTIO from earlier interview transcription are not removed since it is important to identify the context of the existing code from new interview transcription.

In this way new codes are created and already existing codes are tagged to different parts of the text in interviewee transcripts. A snapshot of a code *"comprehensive documentation"* from MAXQDA is shown in figure 4.13, where text assigned to that particular code is retrieved by activating the code and all the interviewee transcripts, shows how many times the code *"comprehensive documentation"* is appeared in each interviewee transcripts and also which interviewee transcripts discussed about it. After finishing interviews and interview transcription, generating codes and tagging codes to different interview transcripts, each code and every interview transcripts are activated. By doing this MAXQDA retrieves context or text assigned to that particular code from each interview. So on reading the retrieved context or text tagged to that particular code helps to evolve concept or understanding around that code.

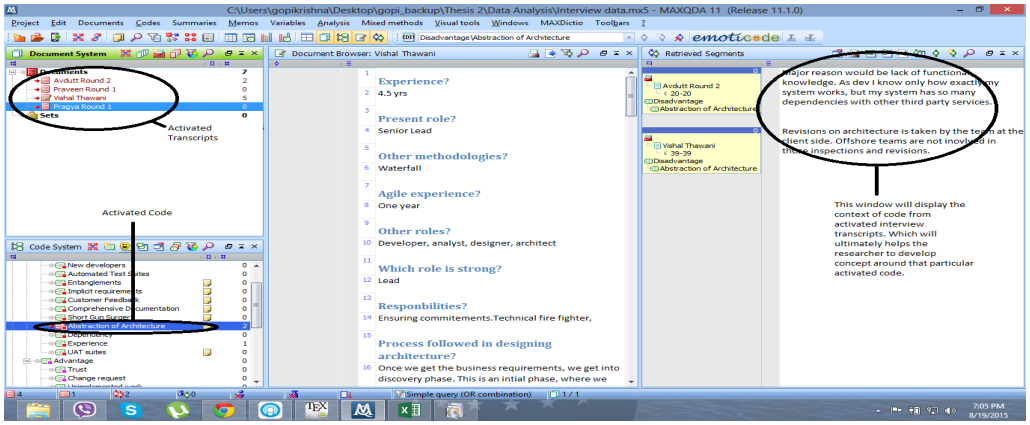


Figure 4.13: A snapshot of code and its context in MAXQDA

In this way codes identified manually from interview transcripts (during theoretical sampling phase) are tagged to different texts in the interview transcripts in codification phase. So in total 28 codes are identified, which are enlisted in table 4.9. Apart from identifying codes manually, word frequency is also used to identify the codes. This is to ensure that during analysis most frequently used words/text parts in transcripts are not left without analyzing. Until the analysis reaches theoretical saturation, i.e. where no more new codes can be identified from the transcripts, the above two steps (i.e. theoretical sampling and codification) are in loop.

Extracted Code	**Extraction Method**	**Extracted Code**	**Extraction Method**
Productivity	Frequency	Team morale	Frequency
Bug-fix	Frequency	Prioritization	Frequency
Cross-training	Frequency	Estimates	Literature
Test Suites	Literature	Maintainability Index	Literature
Customer Satisfaction Levels	Literature	Code Quality	Frequency
Complexity Analysis	Literature	Stable design	Frequency
Comprehensibility	Literature	Change request	Literature
Trust	Literature	Modifiability	Literature
Short Gun Surgery	Frequency	Documentation	Frequency
Customer Feedback	Literature	Abstraction of Architecture	Frequency
Dependency	Frequency	Implicit requirements	Frequency
Entanglements	Frequency	UAT suites	Frequency
Automated test suites	Frequency	Experience	Literature
New developers	Frequency	Stand-ups	Frequency

Table 4.9: Extracted Codes

4.5.4 Generation of Categories by Code Merging (Axial Coding)

The generated codes are again grouped into different categories to show the commonality between different sets of codes. This step is done after the completion of theoretical sampling and codification phase during data analysis. Generating categories helps to develop a relationship between them. Categorizing the codes can be done by constantly comparing the codes. But the categories for the codes in the table 4.9 are identified from RQ.2. As the second round interview aimed at collecting data for RQ.2. So the codes identified from interviews are categorized into two different categories, i.e. *Advantages* and *Disadvantages*. Those two categories revolve around one core-category, i.e. *Software Maintenance*. The distribution of codes from table 4.9 in two categories is presented in the table 4.10.

Category	**Number of Codes**
Advantages	16
Disadvantages	12
Total	**28**

Table 4.10: Distribution of Extracted Codes

From the figure 4.14 below, it is seen that more than 50 percentage of codes are categorized into *"advantages"* category.

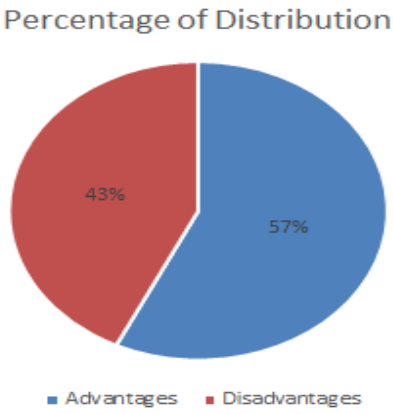


Figure 4.14: Percentage Distribution of Codes among Categories

4.5.5 Selection of Codes

Initially, there were 28 codes identified during theoretical sampling and codification phase. But during categorization of codes only 20 codes are considered. Selection of codes is done by using a feature in MAXQDA, i.e. *Code Matrix Browser*. Using this feature, frequency of different codes can be known. It displays a window as shown in figure 4.15, and shows the frequency of codes. As seen in the figure 4.15, Y-axis represents codes, sub-categories and categories. and X-axis represents interview transcripts. Frequency of codes can be calculated by counting number of red circles in each row.

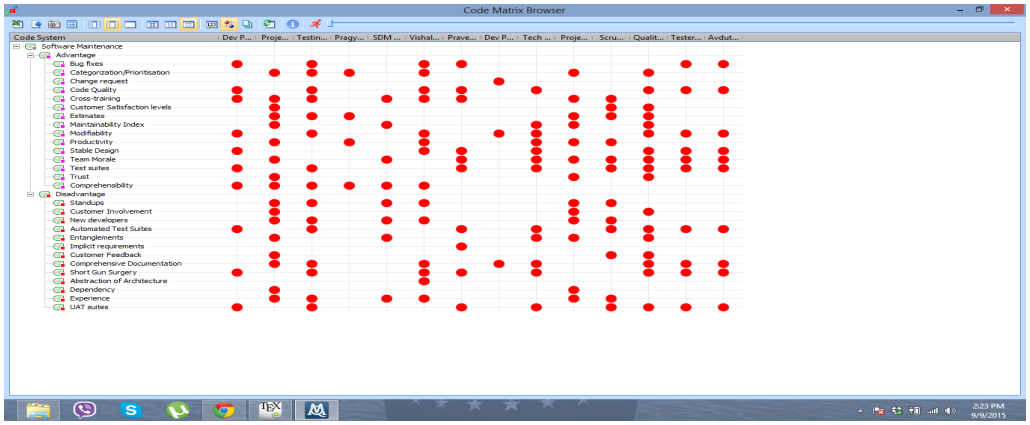


Figure 4.15: Screen Shot for Code Matrix Browser in MAXQDA

Frequency of codes can be defined as number of times a code has been tagged to an text from interview transcripts. For this study threshold frequency of codes is considered to be 6, i.e. if any code has been tagged to text from 6 different interview transcript, then it is considered for categorization. Ratings of codes is considered from the study of Deepika Badampudi [57], where in a code is rated as:

- **most important-** if it is talked by more than 1/3 of interviewees (>67 percentage).
- **medium important-** if it is talked by more than 1/4 of interviewees (>25 percentage).
- **least important-** if it is talked by more than 1/10 interviewees (>20 percentage).
- **not important-** if it is talked by less than 1/10 interviewees (<20 percentage).

From the above rating of codes, most important and medium important codes are considered for categorization. As it ensures to analyze only the codes which are most spoken by interviewees. Statistics of codes in advantage category and disadvantage category are shown in figure 4.16 and 4.17.

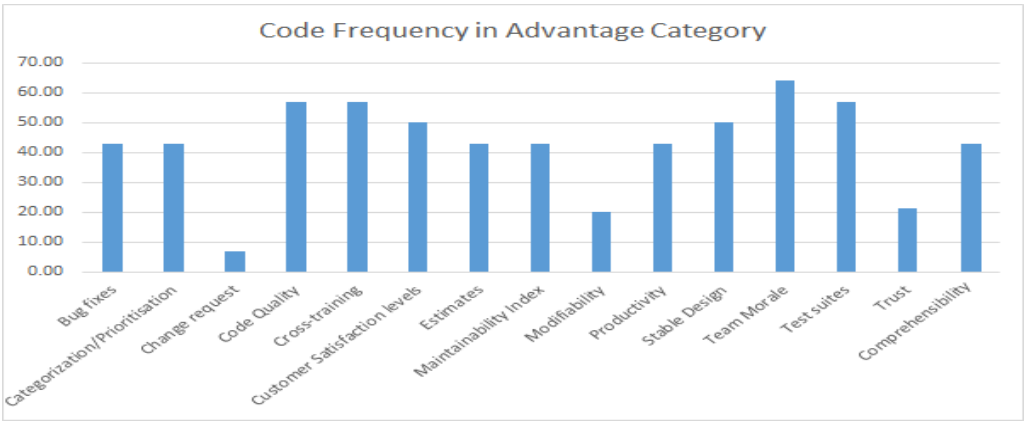


Figure 4.16: Code Frequencies Percentage in Advantage category

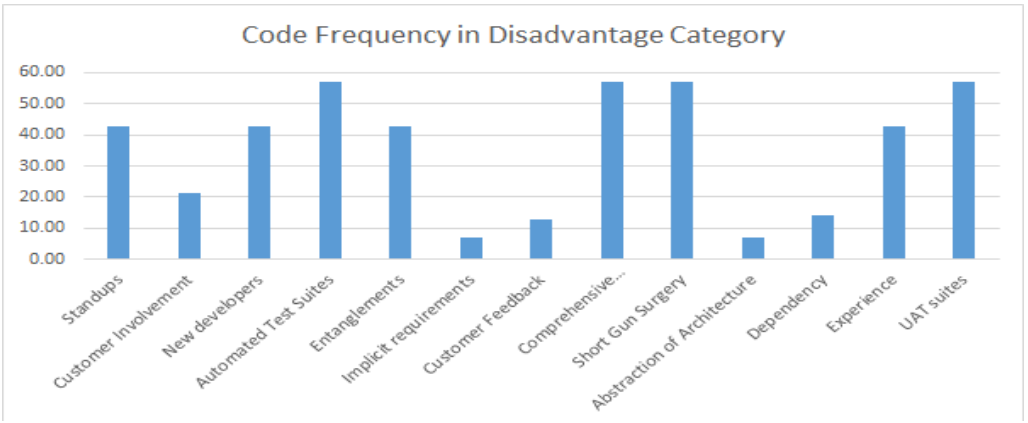


Figure 4.17: Code Frequencies Percentage in Disadvantage category

Hence from the figures it is observed that, codes which have frequency percentage less than 42 percent (i.e. less than 6 interviewees talking about it) are discarded during categorization (axial coding).

4.5.6 Relationship among Categories

Different codes generated during open coding were categorized into two categories and these are *"Advantages"* and *"Disadvantages"*. Now it is important to explain the relationship between the categories by identifying the central phenomenon/core category. As mentioned in the previous chapter 3 in the section 3.1.2.2.4, it is found that *"Software maintenance"* can be used as a core category or fulcrum to define relationships between other categories. The root node of the conceptual framework would be *"Software maintenance"* and it is a predecessor of other two categories, i.e. *"Advantages"* and *"Disadvantages"*. *"Advantages"* determines the advantages of adopting agile practices during software maintenance

and *"Disadvantages"* determines the disadvantages of adopting those practices during software maintenance. This is depicted in the figure 4.18.

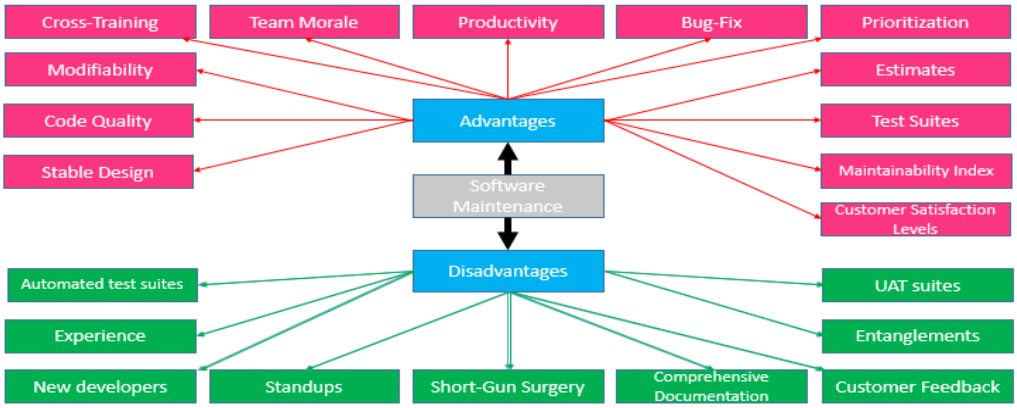
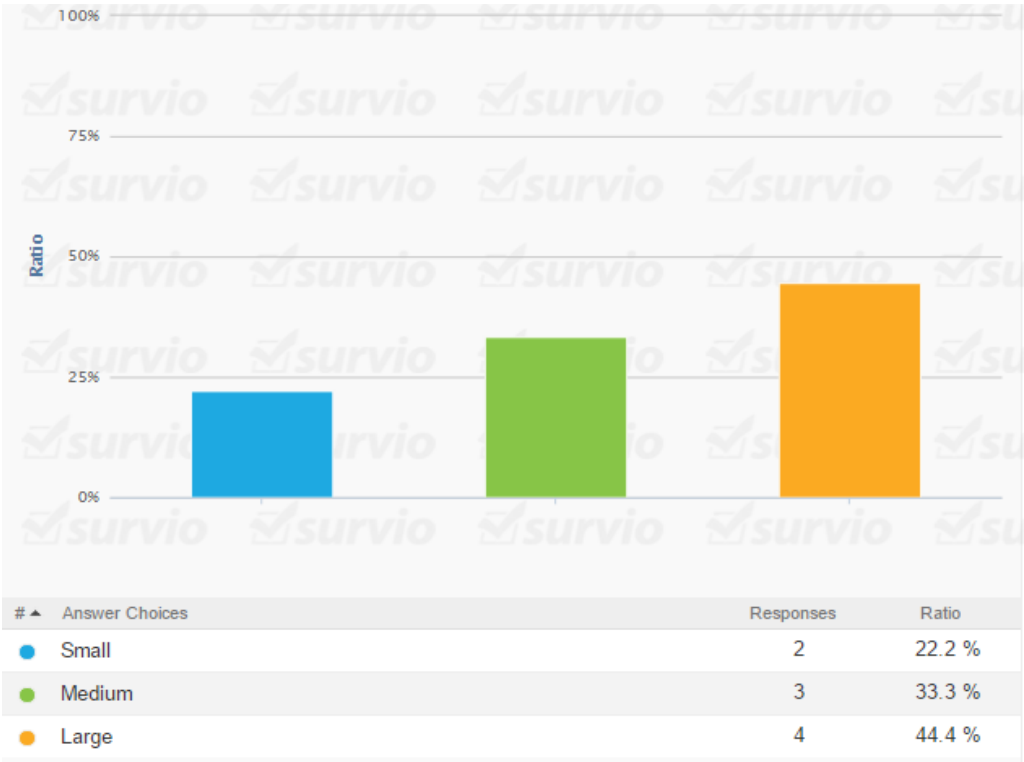


Figure 4.18: A snapshot of Code Tree

4.5.7 Validation Survey

This is final step in data analysis of second round of interviews. Validating the data extracted helps to mitigate construct validity (further discussed in section 6.3.1). Validation of data collected during second round interviews is done using a validation survey among participants of second round interview and also among other practitioner's who have adopted agile during software maintenance. Considering the opinions of other practitioners outside the case study helps to improve the generalisability of the results. So apart from 14 second round interview participants, there were around 9 other participants (practitioners outside the case study) have answered the validation survey. Validation survey had questions which were Close ended. Each question was formulated by using different codes extracted during data analysis of second interview. To improve the understandability, the codes in form of close-ended questions had help-text, which explained what exactly the code meant in the perspective of researcher. Those codes in the form questions should be answered by rating the score from 1-5, where 1 meant strongly disagree and 5 meant strongly agree.

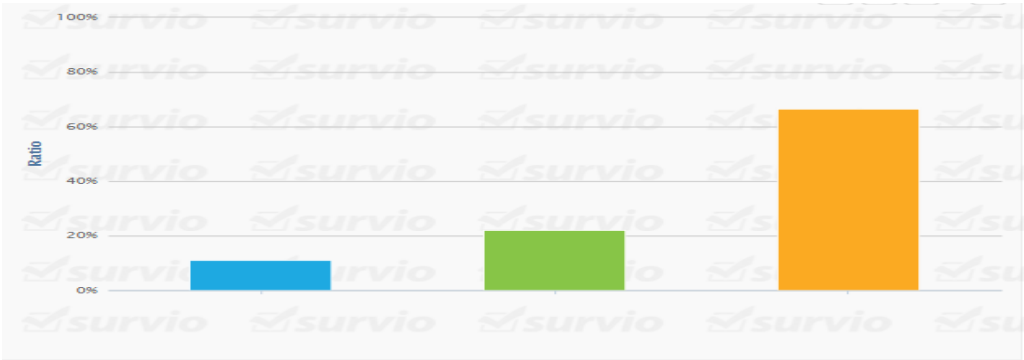
After distributing the validation survey, participants from second round interview were invited to answer and inclusion of other practitioners opinions for validation survey is done through ease of access, i.e. researcher distributed the validation survey among his network who have adopted agile during software maintenance. Along with second round interview participants, 9 other practitioner's have answered the survey. Figures below presents the overview of those practitioners. **(Note: Practitioner term used during the description of figures meant participants of survey from outside the case study)**



#	Answer Choices	Responses	Ratio
1	Small	2	22.2 %
2	Medium	3	33.3 %
3	Large	4	44.4 %

Figure 4.19: Categorization of Participants in terms of Size of Organization

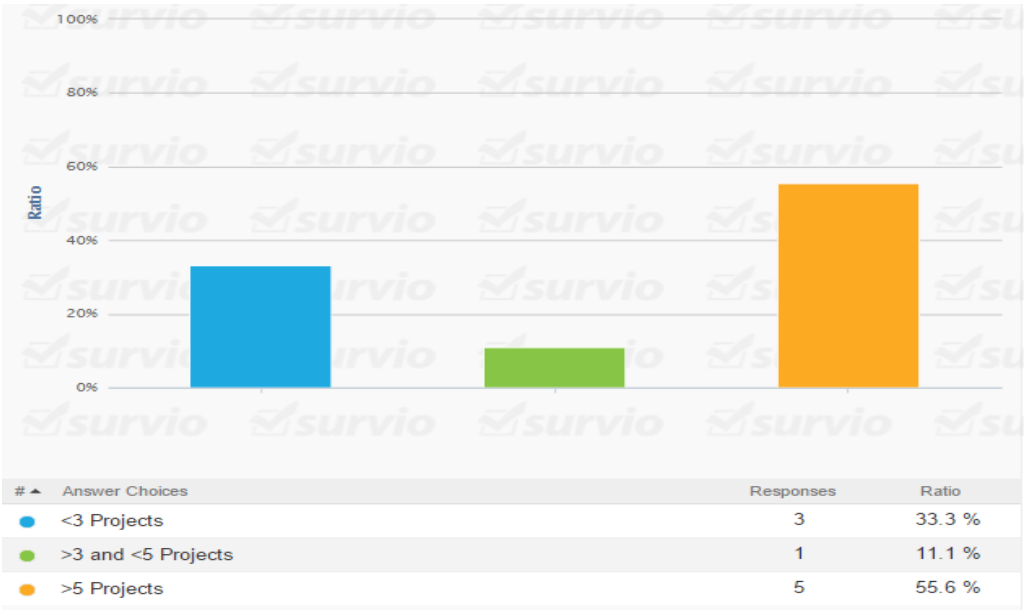
As seen in the figure 4.19, 44.4 percent practitioners mostly belonged to large organizations. Large in this context meant more than 500 employees.



#	Answer Choices	Responses	Ratio
1	<3 Years	1	11.1 %
2	>3 and <5 Years	2	22.2 %
3	>5 Years	6	66.7 %

Figure 4.20: Categorization of Participants in terms Experience in Software Maintenance

As seen in the figure 4.20, 66.6 percent practitioners answering this survey had more than 5 years of experience in software maintenance.



#	Answer Choices	Responses	Ratio
●	<3 Projects	3	33.3 %
●	>3 and <5 Projects	1	11.1 %
●	>5 Projects	5	55.6 %

Figure 4.21: Categorization of Participants in terms Number of Software Maintenance Projects they have worked for

As seen in the figure 4.21, 56.6 percent practitioners answering this survey have worked for more than 5 projects which dealt software maintenance.

Majority of practitioners (answering the validation survey) outside the case study are well matched with second round interview participants. Hence the opinions of those practitioners have been considered for validation of codes extracted during second round interviews. The results of the survey answered by both second round interview participants and practitioners outside case study are gathered and analyzed using statistics. Mean and standard deviations are calculated upon which results are inferred and codes are further classified into different categories. Classification of codes using validation survey is further discussed in chapter 5.

Code	**MAX Value**	**MIN Value**	**Mean**	**Standard Deviation**
Productivity	5	2	4.11	0.92
Team Morale	5	2	4.22	0.97
Bug-Fix	5	1	3.67	1.22
Prioritization	5	1	3.78	1.20
Cross-training	5	2	3.78	1.09
Estimates	5	2	3.44	1.13
Test Suites	5	2	3.44	1.01
Maintainability Index	5	1	3.44	1.24
Customer Level Satisfaction	5	3	4.11	0.78
Code Quality	5	1	3.67	1.22
Stable Design	5	1	3.44	1.24
Comprehensibility	5	1	3.44	1.13

Table 4.11: Validation of Codes for Advantages

Code	**MAX Value**	**MIN Value**	**Mean**	**Standard Deviation**
Short Gun Surgery	5	2	4.11	0.93
Comprehensive Documentation	4	1	3.33	1.00
Entanglements	5	1	3.44	1.33
UAT suites	4	2	3.11	0.78
Automated Test Suites	5	1	3.11	1.27
Experience	4	1	3.11	1.17
New Developers	5	1	3.22	1.30
Stand-ups	4	1	1.78	1.09

Table 4.12: Validation of Codes for Disadvantages

From the above two tables 4.11 and 4.12, it is inferred that codes *"Customer Feedback"* and *"Stand-ups"* from disadvantages sub-category are considered to be least agreed codes from the rest of codes in both sub-categories.

Chapter 5

Data Analysis

In the previous chapter results of first round interview, second round interview and validation survey are presented. Results of those data collection methods infer different information about the research. It is therefore in this section an analysis of those results is done to infer answers for the research questions.

5.1 Relating identified agile practices with Advantages

Section 5.1 discusses the advantages identified during second round interview. These advantages are again combined with associated agile practices in a cause-effect relationship. There are in total 12 advantages which are identified.

5.1.1 Advantages

During analysis of second round interviews 16 codes for advantages were identified. Four codes related to advantages are filtered out based on importance of the codes. The importance of codes is calculated using the frequency of codes (as explained in section 4.5.5). Performing validation survey on those codes has provided further insights into analysis of those codes. Codes are ordered in terms of most agreed, medium agreed and least agreed codes as the result of validation survey. All the codes for advantages, whose mean value as seen from table 4.11 is more than 4 are categorized under *"Most agreed advantages"*. Ratings used during validation survey are between 1-5, i.e. 1 represents strongly disagree and 5 represents strongly agree. Any code whose mean value is more than 4 is categorized under *"Most agreed advantages"*. In similar fashion, any code for advantages whose mean value between 3.5 and 4 are categorized under *"Medium agreed advantages"* and any code for advantage whose mean value is less than 3.5 are categorized under *"Least agreed advantages"*. The categorization of those codes can be seen in table 5.1.

Most Agreed Advantages			
No	**Code**	**Actual Code Description**	**Associated Agile Practices**
1	Team Morale	Improved Team Morale	User Stories , Short Iteration, Task Board
2	Productivity	Improved Productivity	Planning Game, Task Prioritization, Collective Code Ownership, Automated Testing, Stand-up
3	Customer level Satisfaction	Improved Customer satisfaction index	Planning Game, Acceptance Test, Demo
Medium Agreed Advantages			
No	**Code**	**Actual Code Description**	**Associated Agile Practices**
1	Prioritization	Improved Prioritization of tasks	Task Prioritization, User Stories, Product Backlog
2	Cross-training	Improved cross-training amongst many modules of the code base	Pair-Programming, Code Reviews, Coding Standards, Small Team
3	Bug-Fix	Merging of bug fixes rapidly	Unit Testing, Continuous Integration
4	Code Quality	Improved Code Quality	Coding Standards, Code Reviews, Daily Builds
Least Agreed Advantages			
No	**Code**	**Actual Code Description**	**Associated Agile Practice**
1	Estimates	Improved Estimates	User Stories, Product Backlog, Velocity
2	Test Suites	Improved Test Suites	User Stories, Task Prioritization, Refactoring
3	Maintainability Index	Improved Maintainability Index	Refactoring, Coding Standards, Iteration Planning
4	Stable Design	Produces relatively mature and stable design at abstraction level	Pair-Programming
5	Comprehensibility	Improved Program Comprehension	Retrospective, Refactoring

Table 5.1: Categorization of Advantages

5.1.1.1 Most Agreed Advantages

Description of different codes under *"Most agreed advantages"* category is described below.

Improved Team Morale (code:Team Morale) :

Improved team morale in this context implies that different practitioners in maintenance team adopting agile practices, perceive that there is lot more confidence in the work they do. A developer provides his view by stating *"short iterations while handling different maintenance tasks always help our teams to achieve the results faster. Short iterations helps to narrow our focus to few tasks, hence providing solutions which are very much verified and validated before delivering. When our solutions are verified and validated to large extent, then it brings in confidence, which ultimately improves team morale"*. Other developer stated that *"using of tasks board helps the team to see to what extent the work has been done. And when something is visually representing the progress of different tasks, then from the progress of completed tasks team recognition can be seen"*. From the above views figure 5.1 shows a cause-effect relationship between agile practices and the advantage *"Improved Team Morale"*.

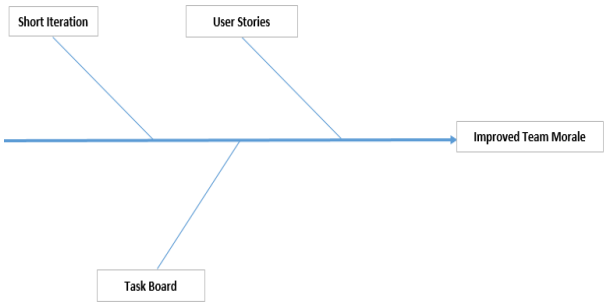


Figure 5.1: Agile Practices causing Improved Team Morale

Improved Productivity (code:Productivity) :

Productivity of the teams adopting agile during maintenance always had a positive effect at case company. In this view one of the Service delivery manager reported that *"adopting agile has always helped me in improving productivity of my team. Being religious about few agile practices especially testing related practices such as automated testing, have always improved productivity of the team during maintenance. Also, answering questions at daily-stand-ups helped team members to resolve few roadblocks, which ultimately reduced time for fixing it"*. One of the developer came up with his perspective about the productivity, *"as whole team is responsible for the code, then any issue will be resolved by taking*

collective responsibility". Apart from those two views a project manager added that *"Prioritizing the tasks based on the dependencies improves the productivity and reduces the rework. Also, Planning game helps to estimate the effort for User Stories especially for product backlog items and value drivers. Thus increasing the productivity of the team"*. From the above views figure 5.2 shows a cause-effect relationship between agile practices and the advantage *"Improved Productivity"*.

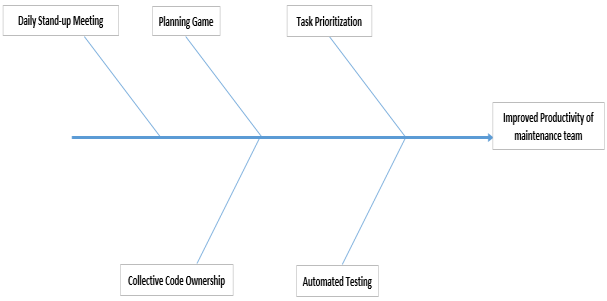


Figure 5.2: Agile Practices causing Improved Productivity

Improved Customer Satisfaction Index (code:Customer Level Satisfaction) :

Customer satisfaction is calculated using a metric i.e. customer satisfaction index. Customer satisfaction index (CSI) is calculated by conducting survey among the customers and this survey considers various factors. The metric is calculated for every release. Adopting agile during maintenance has always shown positive effect on the customer satisfaction index at case company. In relation to this a service delivery manager stated *"as a SDM, my goal is to improve CSI after each release. It is because, positive growth in CSI will tell us that customers are happy and when the customers are happy business between us is healthy. But to point out specifically to the agile practices which helps in achieving my goal are, planning game, acceptance test and demo. During a planning game, work orders for a release are dealt. Allowing a business person to partake in the decision making process always makes customers feel that they have stake in the project. In a similar way, demos to the customer helps them to provide their view and point out few corrections. Handling those reviews and corrections in the same release would give them more confidence"*. From the above views figure 5.3 shows a cause-effect relationship between agile practices and the advantage *"Improved Customer Satisfaction Index"*.

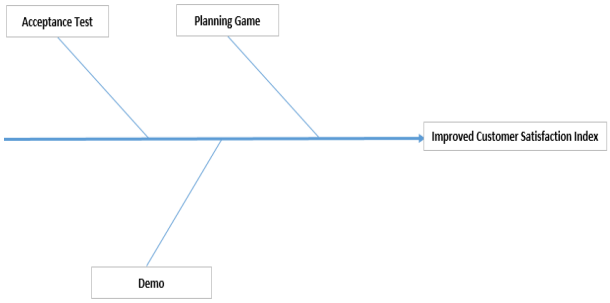


Figure 5.3: Agile Practices causing Improved Customer Satisfaction Index

5.1.1.2 Medium Agreed Advantages

Description of different codes under *"Medium agreed advantages"* category is described below.

Improved Prioritization of Tasks (code:Prioritization) :

Change control board (CCB) which acts as product owner helps in prioritizing tasks during software maintenance. As this board consist of representatives from all the departments from project, each had their on view on this particular advantage. One of the project manager states that *"getting the requirements in the form of user stories helps us to prioritize the tasks very easily. As the requirements are stated from the perspective of user"*. A service delivery manager states that *"we collect or work orders from the specific customer using a tool called GIPS, this actually collects all the information about a user story, such as description and priority. We use that bucket as our product backlog. GIPS also helps to provide information about the status of the tickets. In this way our CCB team can take an effective decision about the tasks to be accomplished in coming iteration and release"*. From the above views figure 5.4 shows a cause-effect relationship between agile practices and the advantage *"Improved Prioritization"*.

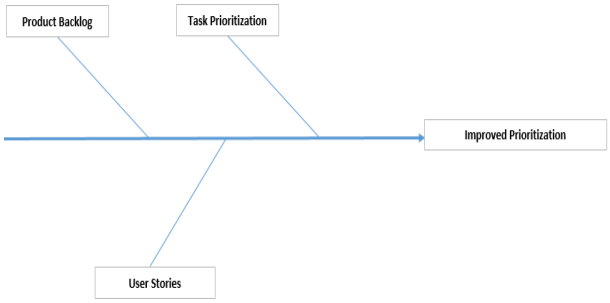


Figure 5.4: Agile Practices causing Improved Prioritization

Improved Cross-training amongst many modules of the code base (code:Cross-training) :

All projects at case company which participated in case study, have involved many platforms, languages and dependencies among different modules of the system. The whole environment was bit complex. Before adopting agile, each module had gurus, i.e. an expert dealing with the system. By doing that they faced few productivity issues, when those gurus left the project. To recover from such situations, they took many months as it is difficult to handle that knowledge debt among the team members. In such situations adopting agile practices have helped them to tackle those scenarios more effectively. In this view one of the developer stated *"using of practices such as pair-programming had always played a major role in transferring the knowledge among the team members equally. For few tasks we pair people handling different systems, which not only helps us to know about the other systems but also helps us to equally distribute the knowledge among other team members"*. It is also seen that each system had a tech lead, who is responsible for code reviews and prioritizing tasks for the team. One of the tech lead adds his view stating that *"we use a template for code reviews. This template helps a reviewer to review code on different factors such as naming conventions. Using this template I can review the code of other teams working on different module. Doing this way tech leads get introduced to other modules and also they get to know the functioning of other modules"*. Another developer stated that *"in our project each module is developed using different languages. In such situations pairing the developers from different teams doesn't help much. But specific coding standards are set for each language, used during development of modules. Having some knowledge of other technologies and complemented with coding standards, any member of other teams can get involved with different modules. And working of the module could be very well understood if the teams are small"*. From the above views figure 5.5 shows a cause-effect relationship between agile practices and the advantage *"Improved Cross-training amongst many modules of the code base"*.

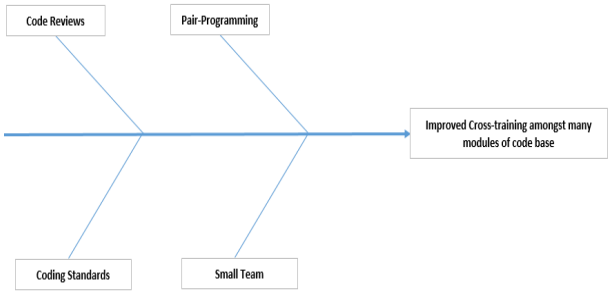


Figure 5.5: Agile Practices causing Improved Cross-training

Merging Bug-Fix rapidly (code:Bug-Fix) :

Bug-fix mostly relates to corrective maintenance. All the projects participating in this case study had this type of maintenance (mostly). In relation to this one developer reported that *"each bug-fix is assigned to different team members in the team. Each writes a unit test case to check whether any solution to bug-fix has passed its unit test. Doing this way, bug-fixes are resolved rapidly, ensuring us to check our solution"*. Other developer reported that *"continuous integration also helps us to merge bug-fixes rapidly into production, hence taking less time to deal with an bug-fix"*. From the above views figure 5.6 shows a cause-effect relationship between agile practices and the advantage *"Merging Bug-Fix rapidly"*.

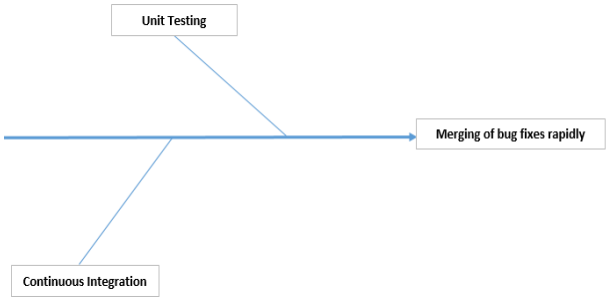


Figure 5.6: Agile Practices causing Merging Bug-fixes rapidly

Improved Code Quality (code:Code-Quality) :

All the projects at case company always focused at improving code quality. High code quality ensures them to mitigate few challenges, especially it improves understandability. When understandability of the code improves it improves the productivity of the team. A scrum master reports *"I faced problems when I rotate resources among modules, even though they had equal knowledge among other modules. Since, developers always had problems in understanding the style of code. But when the practices of agile such as coding standards, helped me to ensure everyone in team had few certain ways to write-up the code. This has lead to increased understandability"*. A tech lead also reports that *"code quality is directly proportional to few technical factors, so I regularly review the code which ensures such proportional factors are not effected"*. A architect states that *"we practice daily-builds. This is usually done at the end of the day. And we have predefined set of constraints that helps to check our builds, such as a build is formed if the number of warnings are less than 100 and 70 percent of those warnings should be false-alarm. If those build constraints are not satisfied I immediately check where exactly the problem arises and if the problem arises from a certain person from the team, I ask him to check his code to resolve build issues"*. In this way different practices helps practitioners to improve the code quality. From the

above views figure 5.7 shows a cause-effect relationship between agile practices and the advantage *"Improved Code Quality"*.

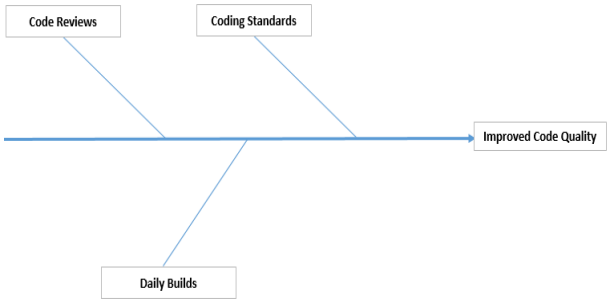


Figure 5.7: Agile Practices causing Improved Code Quality

5.1.1.3 Least Agreed Advantages

Description of different codes under *"Least agreed advantages"* category is described below.

Improved Estimates (code:Estimates) :

In any maintenance environment (i.e. corrective, perfective, adaptive or preventive), it is always difficult to estimate the tasks. Since many factors involve in estimating those tasks, especially it is always difficult for the practitioners to estimate time for analyzing the impact of performing those tasks. In such situations one of the quality lead states that *"estimating the tasks in maintenance is very difficult. Many times we either over-estimate or under-estimate those tasks. In such situations it is always unclear for us to know the count of tasks in an iteration. So to handle such situations we mainly measure our velocity, i.e. we calculate the percentage of expected tasks completed and actual tasks completed in an sprint. Doing this will helps us to know what exactly would be the count of tasks during the next iteration"*. In a similar way one of the developer shared his experience *"getting the requirements in the form of user stories helps us to know the perspective of user. Taking the perspective of user further helps us to provide exact solution, hence when we know the exact solution, then we can easily estimate our work"*. From the above views figure 5.8 shows a cause-effect relationship between agile practices and the advantage *"Improved Estimates"*.

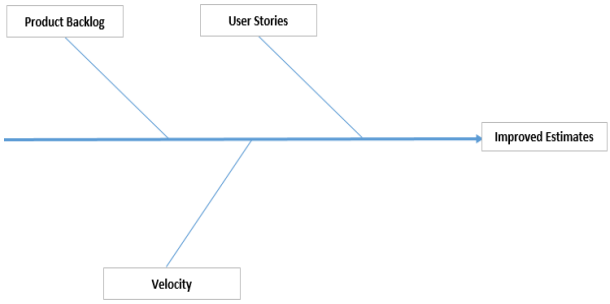


Figure 5.8: Agile practices causing Improved Estimates

Improved Test Suites (code:Test Suites) :
Test suite is a collection of test cases (i.e. unit test cases, acceptance test cases, integration test cases, etc.) which aims to test the system at different levels (i.e. unit, component or system). Having a good test-suites provides the team a lot of confidence, hence improving team morale and identifying flaws in the system. A tester states that *"different agile practices made sure that our test suites are very effective in finding defects or bugs in the system. Different practices such as user stories, task prioritization and refactoring always made our test suite simple and effective. User stories and task prioritization pre-defines the scope of development and sets the expectations, which ultimately helps us to know what exactly to test. Similarly, refactoring aims to reduce complexity of the system, so when the complexity decreases number of lines to be checked are also decreased, hence test cases are also made very simple"*. From the above views figure 5.9 shows a cause-effect relationship between agile practices and the advantage *"Improved Test suites"*.

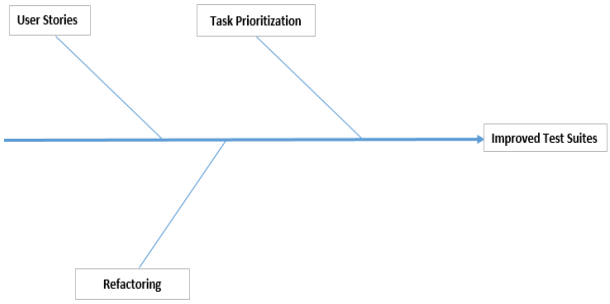


Figure 5.9: Agile practices for Improved Test suites

Improved Maintainability Index (code:Maintainability Index) :
Maintainability Index (MI) is one of the metric which is used to know, to what extent the system is maintainable in terms of analyzability, changeability, stability, testability and maintainbility compliance [58]. It is calculated by a formula and this formula has two versions [58]. Different projects use different versions of

formula at Capgemini. Internal Capgemini Quality Assurance team (ICQA) regulates quality management system at Capgmeini. Hence it collects a pre-defined list of metrics (includes MI) from all the projects for analysis. After which ICQA provides overview of the analysis for collected metrics and suggests few process improvements for each project. In this view one of the quality analyst quotes, *"as a quality analyst one of my responsibility is to provide metric information of the project to ICQA team. Over few major releases I have observed that our MI was gradually improving from the analysis report given by ICQA, this has been seen especially from the releases when few agile practices such as refactoring and coding standards have become religious".* Also one of the project manager added his view stating that *"our iteration plannings depend on few factors of which complexity of the system is one of major factor. So in every iteration we always check whether the system complexity doesn't get more complex".* As maintainability index also depends on complexity of the system, so indirectly iteration planning helps to improve the maintainability index of the system. The figure 5.10 below depicts the relation among agile practices and *"Improved Maintainability Index"* in a cause-effect relationship.

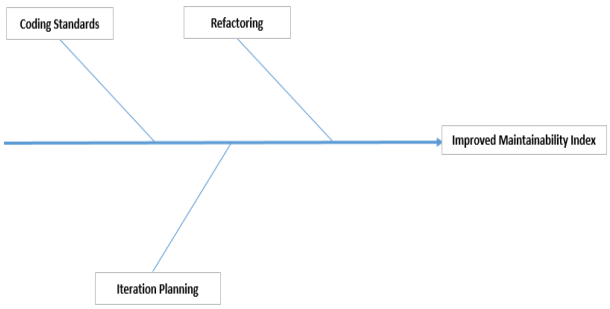


Figure 5.10: Agile practices causing Improved Maintainability Index

Produces relatively mature and stable design at abstraction level (code:Code-Quality) :
At times maintenance also involves re-engineering projects, i.e. changing the architecture of the system without effecting its functionality. It is initiated when the system architecture undergoes a decay. In such situations, different teams which involve in re-engineering projects always think in proactive way, i.e. preventing the system from the issues they have initially faced during software decay. In this context one of the architect presents his views stating that, *"our team was tasked to re-engineer the whole architecture. In such situations we often thought that re-engineering the design should solve the issues pertaining to old ones and also the ones which can arise in near future, i.e. re-engineering design should aim at reducing the complexity as compared to the old one. For us to achieve such matured design with less complexity, we had to look at the old design where complexity could be reduced using refactoring".* A scrum master adds *"retrospect-*

ing after each iteration helps us to know what went wrong and what went right. So in reach retrospect, we usually see at the architecture, to find permanent solutions to few recurring tickets. This way stable design and matured architecture is achieved". From the above views figure 5.11 shows a cause-effect relationship between agile practices and the advantage *"Produces relatively mature and stable design at abstraction level"*.

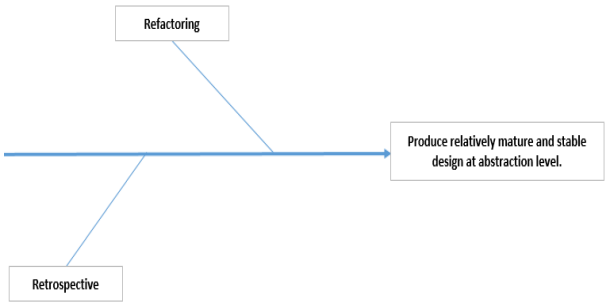


Figure 5.11: Agile Practices causing Stable design

Improved program comprehension (code:Comprehensibility) :
Program comprehension is an important factor during software maintenance [59]. It is part of software engineering which deals with understanding of program or system [59]. Many practices in agile facilitate in improving the program comprehension. In this context one of the developer reports that *"our tasks while dealing with corrective maintenance initially starts with analyzing the impact of resolving bug/defect. But analyzing the impact requires a person to understand the program flow and if the person taking up that task is different from the person who has initially developed it would complicate the situation even more. These scenarios are well managed if a team adopts agile methods pair programming. Hence, improving the program comprehension factor equally among the team members"*. The figure 5.12 below depicts the relation among agile practices and *"Improved program comprehension"* in a cause-effect relationship.

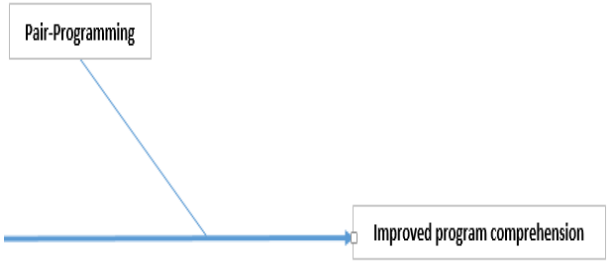


Figure 5.12: Agile Practices causing program comprehension

5.2 Relating agile practices with disadvantages

Section 5.2 discusses the disadvantages identified during second round interview. These disadvantages are again combined with associated agile practices in a cause-effect relationship. There are in total 8 disadvantages which are identified.

5.2.1 Disadvantages

During analysis of second round interviews 12 codes for disadvantages were identified. Four codes related to disadvantages are filtered out based on importance of the codes. The importance of codes is calculated using the frequency of codes (as explained in section 4.5.5). Performing validation survey on those codes has provided further insights into analysis of those codes. Codes are ordered in terms of most agreed, medium agreed and least agreed codes as the result of validation survey. All the codes for disadvantages, whose mean value as seen from table 4.12 is more than 4 are categorized under *"Most agreed disadvantages"*. Ratings used during validation survey are between 1-5, i.e. 1 represents strongly disagree and 5 represents strongly agree. So any code whose mean value is more than 4 is categorized under *"Most agreed disadvantages"*. In similar fashion, any code for advantages whose mean value between 3 and 4 are categorized under *"Medium agreed disadvantages"* and any code for advantage whose mean value is less than 3 are categorized under *"Least agreed disadvantages"*. The categorization of those codes is given in table 5.2.

5.2.1.1 Most Agreed Disadvantages

Description of different codes under *"Most agreed disadvantages"* category is described below.

Leads to short-gun surgery (code: Short-gun Surgery) :
In highly reactive situation, i.e. when there is lot of uncertainty of ticket arrival, iterations are at times disturbed. So if there is a situation wherein the iteration is ending, but a highly prioritized ticket arises. In that case people generally perform short-gun surgery to those tickets in-order to include it in the present sprint. A architect states that *"some time our iterations are often disturbed with highly prioritized tickets. In such cases to keep the momentum of the iteration, we usually do some quick-fix work to such highly prioritized tickets"*. From the above views figure 5.13 shows a cause-effect relationship between agile practices and the disadvantage *"Leads to short-gun surgery"*. The figure 5.13 below depicts the cause-effect relationship between agile practice and the disadvantage *"Leads to short-gun surgery"*.

Most Agreed Disadvantages			
No	**Code**	**Actual Code Description**	**Associated Agile Practice**
1	Short-gun Surgery	Leads to short-gun surgery	Iteration
Medium Agreed Disadvantages			
No	**Code**	**Actual Code Description**	**Associated Agile Practice**
1	Comprehensive Documentation	Lacks comprehensive documentation	Iterations
2	Entanglements	Increase in entanglements	Refactoring
3	UAT suites	Maintenance of Acceptance test suites	UAT, User stories
4	Automated Test Suites	Maintenance of automated test suites	Iterations
5	Experience	Requires experienced/competent personnel	None
Least Agreed Disadvantages			
No	**Code**	**Actual Code Description**	**Associated Agile Practice**
1	New Developers	Comprehensibility issues among new developers	None
2	Stand-ups	Time wastage due to Stand-ups	Daily Stand-up Meetings

Table 5.2: Categorization of Disadvantages

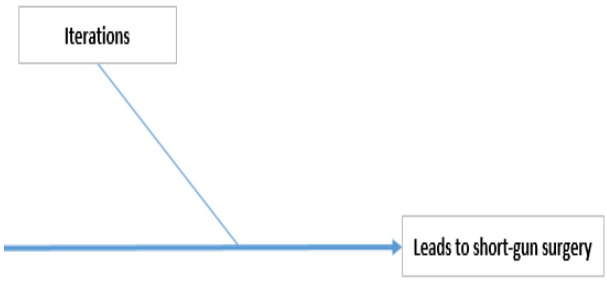


Figure 5.13: Agile Practice causing Short-gun Surgery

5.2.1.2 Medium Agreed Disadvantages

Description of different codes under *"Medium agreed disadvantages"* category is described below.

Lacks Comprehensive Documentation (code:Comprehensive Documentation) :
All the projects develop their system using different platforms and technologies. Along with diverse nature of modules, dependencies among them would further complicate or makes the system complex. In this scenario having to understand the complexity of the system is bit difficult if there is no support in the form of documentation. A developer reports that *"adopting agile has brought many changes into our team. Even though we had some good effects of those changes there is also bad side to it. One of which I would state is documentation. Documentation has been always neglected, which effected me in understanding other modules when there were dependency issues among different modules"*. This usually brought in new challenge, i.e. fear of changing. Practitioners working on dependency issues (corrective maintenance) at times had problem in understanding other modules. A developer reports that *"lack of comprehensive documentation can be accounted to iteration. It is because working in such environments makes us to trade few things and majority of the thing would be neglecting the changes to documentation"*. The figure 5.14 below depicts the cause-effect relationship between agile practice and the disadvantage *"Lacks Comprehensive Documentation"*.

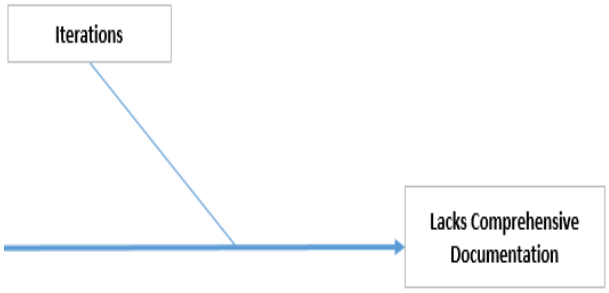


Figure 5.14: Agile Practice causing negative effect on comprehensive documentation

Requires experienced/competent personnel (code:Experience) :
Having new developers working for maintenance team is always a difficult situation, when the team lacks comprehensive documentation of the system. This may even intensify the situation if agile is adopted to software maintenance, as both require competent and experienced personnel using it and working with it respectively. In this regard a Service delivery manager states *"analyzing the impact of changes during corrective or perfective maintenance requires either experience*

with system/module or good documentation of the system/module. Agile has reduced the latter, hence we are left with choice to use only experienced resources for those tasks". As this disadvantage is a social factor it cannot be related to any agile practice.

Increased Entanglements (code:Entanglements) :

The challenge of having only experienced members on board can be negated with different agile practices such as pairing a experienced developer with new developer. But even this has few negative effects to it. One of the developer stated *"even though we transfer lot of knowledge to the new ones by pairing, in few situations they have to work alone. In such situations lack of comprehensive documentation leads to fear of changes. They always have problems in finding trivial set of refactorings. This increases entanglements and duplication of code"*. Other developer states that *"there some factors before initiating refactoring. And lack of documentation of the system makes newbies in the team to miss few factors, which leads to unstable design and architecture"*. So lacking comprehensive documentation and having a non-trivial set of refactorings both leads to increase in entanglements. The figure 5.15 below depicts the cause-effect relationship between agile practice and the disadvantage *"Increased Entanglements"*.

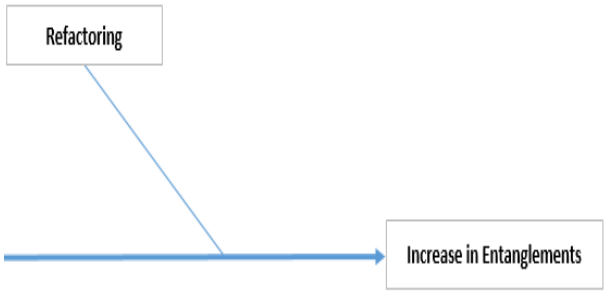


Figure 5.15: Agile Practice causing Increased Entanglements

Maintenance of Acceptance test suites (code:Test Suites) :

Projects particiapting in case study at case company have been developed using many modules and those modules are developed using different platform and technologies. Hence, size of the code is very large and have many dependencies among them. In such scenarios one of the testing lead states *"code coverage in our project is always been an issue. Many cross-references among different modules has negative effect test suites effectiveness in terms code coverage. It is very difficult for the testing team to have high coverage. It becomes really difficult to even maintain those test suites when the bug-fixing type of maintenance is dominant"*. So when the tickets related to bug-fixing are dominant, it is really difficult for the testers to maintain such test suites as the code involves many

dependencies. The figure 5.16 below depicts the cause-effect relationship between agile practice and the disadvantage *"Maintenance of Acceptance test suites"*.

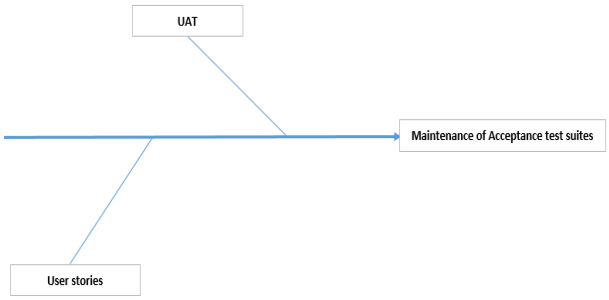


Figure 5.16: Agile Practices causing Maintenance of Acceptance Test Suites

Maintenance of Automated test suites (code:Automated Test Suites) :

Many benefits can be attributed to test automation such as productivity. But test automation has few challenges during adaptive, corrective and perfective maintenance. A testing lead stated that *"test automation is never considered in our project. It is because in our project we deal with adaptive maintenance, as our customer wanted their system to be updated always. Hence, change in technology occurs very often. In that case usage of test automation and maintenance of those automated test suites is additional cost for the team"*. Another tester states that *"we generally use test automation for regression, and corrective maintenance being dominant in our project always leads to change in test automation suites. Apart from that test automation becomes even more difficult if there is dependency issues among different modules"*. Working in fast development environments always gives less time for testing, in that situations maintaining automated test suites creates a overhead for the teams.

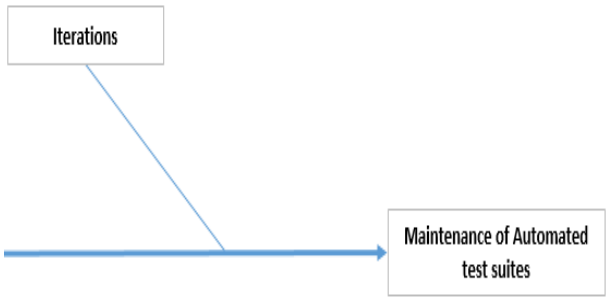


Figure 5.17: Agile Practices causing Maintenance of Test Automation Suites

5.2.1.3 Least Agreed Disadvantages

Description of different codes under *"Least agreed disadvantages"* category is described below.

Comprehensibility issues among new developers (code:New Developers) :

Teams face many issues when an experienced person leaves. Taking in newbies into team can only intensify the effects but does not resolve the issues. Newbies have always steep learning curve when involved in maintenance type of projects. As the work during maintenance not only needs creativity but also needs the skill of analyzing the impacts of making changes during enhancement of system (perfective) or fixing bugs of the system (corrective). In relation to this a project manager states that *"my productivity is always effected when there are newbies in my team. And adopting agile for maintenance even degrades the situation. Complex nature of our system makes them really difficult to understand the program flow and dependencies. New developers do not have good program comprehension, hence the usage of reusable parts of the system is not as effective as the old ones"*. As it is a social factor it cannot be related to any agile practice.

Time wastage due to stand-ups (code:Stand-ups) :

In some cases bug-fixing and providing support to the user is dominant than enhancement tasks. In such cases teams work a lot on fixing of bugs and provide support. Little development of new feature happens. People work on different tasks which are diverse in nature. So in this case sharing the knowledge about tasks during stand-ups would be less productive. In such situations a developer states *"when we adopted agile, we were very religious with our practices, but after a while we felt that few practices should not be followed religiously. It is because sometimes our bug-fixing activities are dominated over new development. In such cases we use daily scrum, but when our new development becomes dominant we re-modulate to daily scrum"*. No doubt agile practices are very useful but practitioners have to consider few things before adopting it during maintenance. The figure 5.18 below depicts the cause-effect relationship between agile practice and the disadvantage *"Time wastage due to stand-ups"*.

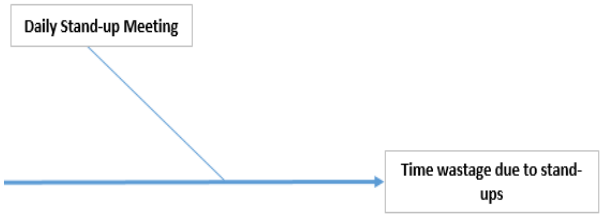


Figure 5.18: Agile Practice causing Time wastage due to stand-ups

Chapter 6

Discussion and Limitations

In this chapter results of the study are compared with literature and further discussed. Besides limitations to the study are explored.

6.1 Discussions on Advantages

As the first part of second research questions advantages as perceived by practitioners while adopting agile during software maintenance are identified by conducting interviews. Improved maintainability index is one of the least agreed advantage from the practitioners perspective. This view is quite contrasting from the literature reported by [58, 60, 61]. Improved maintainability index is widely accepted advantage from the literature perspective has compared to practitioners' perspective. As few projects involved in case study analyze maintainability index over releases, hence this might have caused such difference in perception as an advantage. In similar to that, improved cross-training amongst other modules and improved program comprehension are other advantages where the difference in perception of practitioners and literature differed. But in both the cases agile practices causing it are similar. Pair-programming, small teams and retrospective have been cause factors for those advantages. This perception of practitioners is even supported by [62, 63]. Other advantages identified advantages such as improved productivity, team morale, improved customer satisfaction index and merging of bug-fixes rapidly is also supported by Poole and Huisman in [40].

6.2 Discussions on Disadvantages

As similar to first part, other part of second research question, disadvantages as perceived by practitioners while adopting agile during software maintenance are identified by conducting interviews. One of the most agreed disadvantage was short-gun surgery. In a highly reactive situations, i.e. committing to resolve a highly prioritized ticket (task) would only provide temporary solutions. These temporary solution would even complicate the things, as it may effect other parts of the system, this is also supported in [17]. Maintenance of automated test suites

in a high-paced development such as iterations is always challenging situation for practitioners. Automating test cases would of course bring in lot of benefits but it has its own challenges in maintenance environment. In the first case implementing test automation would incur lot of cost along with that practitioners using it should have certain exposure to it [64], which means an additional cost for training [65]. If the system involves lot of dependency among different modules and third-party systems, changes to the system should also effect automated test suites. Hence, as supported by Skoglund and Runeson in [66], maintenance of automated test suites should be hand in hand with maintenance of the system. Maintenance of acceptance test suites is also an disadvantage, while adopting agile during maintenance. This becomes a problem when the system under goes major re-structuring or adaptive type of maintenance (i.e. change in technology within the system). In such situations, if the acceptance test suite is very large re-structuring or maintaining (updating) it would incur additional cost and overhead for the practitioners [67]. Few agile practices such as daily-stand up, demos might not be useful when corrective type of maintenance is dominant, since it doesn't make sense for the team members during such activities, a similar view is provided by authors in [65]. Adopting agile would effect the comprehensive documentation in a negative way is also discussed in [17]. So before adopting agile during maintenance, a careful consideration has to be made by the team about being religious about the practices. If it is highly reactive situation, wherein customers cannot wait for the iteration to be finished for a bug fix, then few practices should be left alone. But if majority of maintenance tasks involves in updating the system with new features or refactoring, then following agile religiously would benefit the practitioners

6.3 Threats to Validity

This study involves an empirical investigation to gain the results, but it is been reported in [68] that, every empirical study deals with certain threats to the validity of the results. Some of these threats are discussed in [47], and those are construct validity, internal validity, external validity and reliability. It is considered that every empirical study should involve those four tests to judge the quality of the results. Similarly, as proposed in [47], this study considers few threats to the validity of the results and also mitigation strategies to mitigate those threats

6.3.1 Construct Validity

As the researcher is new to data analysis, there may be a threat of construct validity. Since, the study may not represent what exactly researcher had in mind

and what exactly is investigated according research questions. Different measures have been taken to control those construct validity threats. Firstly, results gained after interviews are further investigated using data triangulation. For example, one of the advantage of using agile was improving the maintainability of the code (extracted from interview transcription), this was re-checked after measuring the maintainability of code base for about three releases. Similarly, a tool was used during data analysis to reduce manual errors and also this tool helps in tracing the results back to specific interview transcription. Apart from those two measures, a validation survey was conducted among interview participants and other practitioners of software maintenance (external to case company).

6.3.2 Internal Validity

Internal validity threat comes to surface when causal relations are investigated [45]. As this study mostly involves in investigating the causal relations, hence it might have a threat of internal validity. To control this threat participants during interviews at case company were specifically selected from different software maintenance projects (who had vast experience in software maintenance projects in any role). Thus, it made easy to pose questions to interviewees in relation to software maintenance.

6.3.3 External Validity

The results obtained in this case study are bit difficult to generalise, since this study mainly pertains to a single organization. So making these analytical generalisations to boarder field is bit risky. In order to mitigate this threat, results generated after analyzing the interviews was framed into a survey, which is not only distributed among case study participants, but it was also distributed among practitioners who work in maintenance environment using agile methodology.

6.3.4 Reliability

As this study is conducted by just one researcher, the chance of generating biased results is very high, i.e. there is a reliability threat to the study. Apart from that, majority of the study is conducted in India, due to which the researcher had to stay away from the supervisor. So it has even compounded the effect of reliability threat. To mitigate this situation, researcher provided weekly reports to internal supervisor, especially during transcription and analysis of interviews. Also external supervisor agreed to validate the codes extracted during data analysis for one

randomly picked interview transcription. The purpose was to minimize the bias and increase the accuracy during data collection.

Chapter 7

Conclusions

In this section conclusions of the study, answers to the research questions and future work to this study is reported.

7.1 Conclusions

In this study, researcher tried to explore the area of agile maintenance from practitioners perspective in order to understand different agile practices adopted during software maintenance and also to identify advantages and disadvantages of adopting those practices. A case study with different projects was conducted at Capgemini to accomplish the research objectives. Two rounds of interviews with different objectives were used as data collection instrument. Grounded theory was used as analysis method.

As a result, different agile practices which were mostly adopted and used were revealed, and along with that 11 advantages and 8 disadvantages were identified as a result of adopting those agile practices during maintenance. Most of the identified disadvantages in study were due to application of agile practices during corrective type of maintenance and most of the advantages identified were result due to the application of agile practices to perfective and adaptive type of maintenance. So it brings to the conclusion that in an environment if the corrective type of maintenance is dominant then few agile practices should be traded in. Similarly, if the perfective or adaptive type of maintenance is dominant then high usage of agile practices brings in lot of benefits. But in a project where both bug-fixing (corrective) and enhancement or refactoring (perfective or adaptive) have equal importance, then team organization in the projects should be done in such a way that a team should handle corrective type of maintenance by following agile practices to lesser extent and another team should handle tasks related to perfective or adaptive type of maintenance by following agile practices to larger extent. Based on customer priority a product owner can either shift a task to corrective maintenance product backlog (for high prioritized tasks) or perfective maintenance product backlog (for tasks which are prioritized on release basis).

From the above conclusions it can stated that the case company had adopted agile practices during maintenance, but did not have the knowledge about how

and to what extent the agile practices should be used. This study provides the practitioners at case company about the knowledge of different agile practices adoption from different perspective during software maintenance. Besides it also gives them an idea of what they should improve and spread around.

7.2 Answers to Research Questions

7.2.1 Research Question 1

What are the agile practices adopted during software maintenance?

Initial set of agile practices used during software maintenance are identified by conducting a simple literature review. Then the list is rated upon the usage by the practitioners at case company through an close-ended interview. As the data points were very less graphical representations was used for analysis to infer results. Practices are categorized into sections such as practices related to requirements and design, practices related to implementation, practices related to testing, practices related to process and practices related to organization. User stories, product backlog, task prioritization, coding standards, pair-programming, refactoring, collective code-ownership, daily builds, continuous integration, code reviews, bug-tracking, small releases, unit testing, acceptance testing, short iteration, release planning, iteration planning, task board, daily stand-up meetings, demos and on-site customer are identified as mostly adopted and used agile practices. Agile practices such as product backlog, user stories, task prioritization, pair programming, refactoring, code reviews and planning game have been associated with advantages and short iterations, daily stand-up meetings acceptance test are other agile practices which have been associated with disadvantages.

7.2.2 Research Question 2

What are the advantages and disadvantages of adopting agile practices during software maintenance?

After identifying different agile practices from the perspective of adoption and usage by practitioners, results of those agile practices are revealed by conducting a second round of interviews among different people who participated in case study. The interviews were mostly open-ended and were conducted using a open-ended questionnaire. For analysis of interviews grounded theory was applied. As a result 16 advantages and 12 disadvantages were initially identified. But after considering the response rate of interview participants, 5 advantages and 4 disadvantages were filtered. Improved team morale, improved productivity, improved customer level satisfaction, improved prioritization, improved cross-training, merging of

bug-fixes, improved code quality, improved estimates, improves test suites, improved maintainability index, stable design and improved comprehensibility are identified as advantages. Leading to short-gun surgery, lacking comprehensive documentation, increasing entanglements, maintenance of acceptance test suites, maintenance of automated test suites, experienced personnel, comprehensibility issues among new developers and time wastage due to stand-ups are identified as disadvantages.

7.3 Future Work

Many studies have reported that organizations have reaped benefits by adopting agile practices, but software maintenance as a factor is rarely considered. As the success stories of agile adoption during development are continuously heard through literature, software maintenance practitioners are also being attracted towards its adoption. Hence, this study has made an attempt to identify the results of adopting agile practices during maintenance. As the study reported that not every type of maintenance is suitable for agile adoption, there is a possibility of further extending this research to mitigate such disadvantages with respect to agile adoption during maintenance. In one possible way, a framework can be developed by collecting the views of experienced agile and software maintenance practitioners in near future, to turn the disadvantages into advantages. In another possible way it can also be extended by developing an agile software maintenance adoption framework, which helps practitioners to adopt agile practices in all aspects of maintenance irrespective of maintenance type.

Bibliography

[1] M. Lehman, "Programs, life cycles, and laws of software evolution," *Proceedings of the IEEE*, vol. 68, pp. 1060–1076, Sept 1980.

[2] M. McClure and J. Martin, "Software maintenance: The problem and its solutions," 1983.

[3] H. Sneed, "Offering software maintenance as an offshore service," in *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pp. 1–5, Sept 2008.

[4] R. D. Banker, S. M. Datar, and C. F. Kemerer, "A model to evaluate variables impacting the productivity of software maintenance projects," *Management Science*, vol. 37, no. 1, pp. 1–18, 1991.

[5] R. D. Banker, S. M. Datar, C. F. Kemerer, and D. Zweig, "Software complexity and maintenance costs," *Communications of the ACM*, vol. 36, no. 11, pp. 81–94, 1993.

[6] B. P. Lientz, E. B. Swanson, and G. E. Tompkins, "Characteristics of application software maintenance," *Communications of the ACM*, vol. 21, no. 6, pp. 466–471, 1978.

[7] K. P. B. Webster, K. M. de Oliveira, and N. Anquetil, "A risk taxonomy proposal for software maintenance," in *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pp. 453–461, IEEE, 2005.

[8] R. N. Charette, K. M. Adams, and M. B. White, "Managing risk in software maintenance," *IEEE Softw.*, vol. 14, pp. 43–50, May 1997.

[9] J. Choudhari and U. Suman, "Iterative maintenance life cycle using extreme programming," in *Advances in Recent Technologies in Communication and Computing (ARTCom), 2010 International Conference on*, pp. 401–403, Oct 2010.

[10] R. Yongchang, L. Zhongjing, X. Tao, and C. Xiaoji, "Software maintenance process model and contrastive analysis," in *Information Management, Innovation Management and Industrial Engineering (ICIII), 2011 International Conference on*, vol. 3, pp. 169–172, Nov 2011.

[11] S. Nerur, R. Mahapatra, and G. Mangalaraj, "Challenges of migrating to agile methodologies," *Commun. ACM*, vol. 48, pp. 72–78, May 2005.

[12] D. P. Truex, R. Baskerville, and H. Klein, "Growing systems in emergent organizations," *Communications of the ACM*, vol. 42, no. 8, pp. 117–123, 1999.

[13] M. Kajko-Mattsson and J. Nyfjord, "A model of agile evolution and maintenance process," in *System Sciences, 2009. HICSS'09. 42nd Hawaii International Conference on*, pp. 1–10, IEEE, 2009.

[14] W. Reyes, R. Smith, and B. Fraunholz, "Agile approaches to software maintenance: an exploratory study of practitioner views," in *Managing worldwide operations and communications with information technology: 2007 Information Resources Management Association International Conference, Vancouver, British Columbia, Canada May 19-23, 2007*, pp. 265–269, IGI Publishing, 2007.

[15] G. Prakash, "Achieving agility in adaptive and perfective software maintenance.," in *CSMR*, pp. 61–62, 2010.

[16] D. Knippers, "Agile software development and maintainability," 2011.

[17] G. K. Hanssen, A. F. Yamashita, R. Conradi, and L. Moonen, "Maintenance and agile development: Challenges, opportunities and future directions," in *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pp. 487–490, IEEE, 2009.

[18] L. T. Heeager and J. Rose, "Optimising agile development practices for the maintenance operation: nine heuristics," *Empirical Software Engineering*, pp. 1–23, 2014.

[19] K. Sureshchandra and J. Shrinivasavadhani, "Moving from waterfall to agile," in *Agile, 2008. AGILE'08. Conference*, pp. 97–101, IEEE, 2008.

[20] L. Rising and N. S. Janoff, "The scrum software development process for small teams," *IEEE software*, no. 4, pp. 26–32, 2000.

[21] J. Ågerfalk, B. Fitzgerald, and O. P. In, "Flexible and distributed software processes: old petunias in new bowls," in *Communications of the ACM*, Citeseer, 2006.

[22] J. Erickson, K. Lyytinen, and K. Siau, "Agile modeling, agile software development, and extreme programming: the state of research," *Journal of database Management*, vol. 16, no. 4, p. 88, 2005.

[23] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas, "Manifesto for agile software development," 2001.

[24] P. Abrahamsson, *Agile Software Development Methods: Review and Analysis (VTT publications)*. 2002.

[25] J. Li, N. B. Moe, and T. Dybå, "Transition from a plan-driven process to scrum: a longitudinal case study on software quality," in *Proceedings of the 2010 ACM-IEEE international symposium on empirical software engineering and measurement*, p. 13, ACM, 2010.

[26] S. Jalali and C. Wohlin, "Agile practices in global software engineering-a systematic map," in *Global Software Engineering (ICGSE), 2010 5th IEEE International Conference on*, pp. 45–54, IEEE, 2010.

[27] D. J. Reifer, "How good are agile methods?," *Software, IEEE*, vol. 19, no. 4, pp. 16–18, 2002.

[28] L. Vijayasarathy and D. Turk, "Agile software development: A survey of early adopters," *Journal of Information Technology Management*, vol. 19, no. 2, pp. 1–8, 2008.

[29] S. Sukumaran and A. Sreenivas, "Identifying test conditions for software maintenance," in *Software Maintenance and Reengineering, 2005. CSMR 2005. Ninth European Conference on*, pp. 304–313, IEEE, 2005.

[30] B. P. Lientz and E. B. Swanson, "Software maintenance management," 1980.

[31] K. Bennett and V. Rajlich, "A new perspective on software evolution: the staged model," *submitted for publication to IEEE*, 1999.

[32] Iso, "International Standard - ISO/IEC 14764 IEEE Std 14764-2006," *ISO/IEC 14764:2006 (E) IEEE Std 14764-2006 Revision of IEEE Std 1219-1998)*, pp. 1–46, 2006.

[33] I. Heitlager, T. Kuipers, and J. Visser, "A practical model for measuring maintainability," in *Quality of Information and Communications Technology, 2007. QUATIC 2007. 6th International Conference on the*, pp. 30–39, IEEE, 2007.

[34] K. Aggarwal, Y. Singh, P. Chandra, and M. Puri, "Measurement of software maintainability using a fuzzy model," *Journal of Computer Science*, vol. 1, no. 4, p. 538, 2005.

[35] V. Nguyen, B. Boehm, and P. Danphitsanuphan, "A controlled experiment in assessing and estimating software maintenance tasks," *Inf. Softw. Technol.*, vol. 53, pp. 682–691, June 2011.

[36] I. Buchmann, S. Frischbier, and D. Putz, "Towards an estimation model for software maintenance costs," in *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*, pp. 313–316, March 2011.

[37] D. I. Sjøberg, J. E. Hannay, O. Hansen, V. B. Kampenes, A. Karahasanovic, N.-K. Liborg, and A. C. Rekdal, "A survey of controlled experiments in software engineering," *Software Engineering, IEEE Transactions on*, vol. 31, no. 9, pp. 733–753, 2005.

[38] H. Svensson and M. Host, "Introducing an agile process in a software maintenance and evolution organization," in *Software Maintenance and Reengineering, 2005. CSMR 2005. Ninth European Conference on*, pp. 256–264, March 2005.

[39] N. Jain, "Offshore agile maintenance.," in *Agile*, pp. 327–336, 2006.

[40] C. Poole and J. W. Huisman, "Using extreme programming in a maintenance environment," *IEEE Software*, vol. 18, no. 6, pp. 42–50, 2001.

[41] S. Shaw, "Using agile practices in a maintenance environment," *Intelliware Development Inc*, 2007.

[42] J. Thomas, "Introducing agile development practices from the middle," in *Engineering of Computer Based Systems, 2008. ECBS 2008. 15th Annual IEEE International Conference and Workshop on the*, pp. 401–407, IEEE, 2008.

[43] M. Polo, M. Piattini, and F. Ruiz, "Using a qualitative research method for building a software maintenance methodology," *Software: Practice and Experience*, vol. 32, no. 13, pp. 1239–1260, 2002.

[44] B. Patwardhan, "An overview of devops," *CSI Communications*, p. 14, 2012.

[45] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical software engineering*, vol. 14, no. 2, pp. 131–164, 2009.

[46] J. Rowley and F. Slack, "Conducting a literature review," *Management Research News*, vol. 27, no. 6, pp. 31–39, 2004.

[47] R. K. Yin, *Case study research: Design and methods*. Sage publications, 2013.

[48] C. B. Seaman, "Qualitative methods in empirical studies of software engineering," *Software Engineering, IEEE Transactions on*, vol. 25, no. 4, pp. 557–572, 1999.

[49] B. Kitchenham and S. L. Pfleeger, "Principles of survey research: part 5: populations and samples," *ACM SIGSOFT Software Engineering Notes*, vol. 27, no. 5, pp. 17–20, 2002.

[50] G. Coleman and R. O'Connor, "Using grounded theory to understand software process improvement: A study of irish software product companies," *Inf. Softw. Technol.*, vol. 49, pp. 654–667, June 2007.

[51] S. Sarker, F. Lau, and S. Sahay, "Using an adapted grounded theory approach for inductive theory building about virtual team development," *SIGMIS Database*, vol. 32, pp. 38–56, Dec. 2000.

[52] S. Qureshi, M. Liu, and D. Vogel, "A grounded theory analysis of e-collaboration effects for distributed project management," in *System Sciences, 2005. HICSS'05. Proceedings of the 38th Annual Hawaii International Conference on*, pp. 264c–264c, IEEE, 2005.

[53] B. H. Hansen and K. Kautz, "Grounded theory applied-studying information systems development methodologies in practice," in *System Sciences, 2005. HICSS'05. Proceedings of the 38th Annual Hawaii International Conference on*, pp. 264b–264b, IEEE, 2005.

[54] L. Silva and J. Backhouse, *Becoming part of the furniture: the institutionalization of information systems*. Springer, 1997.

[55] S. A. LaRocco, "A grounded theory study of socializing men into nursing," *The Journal of Men's Studies*, vol. 15, no. 2, pp. 120–129, 2008.

[56] J. W. Creswell, *Research design: Qualitative, quantitative, and mixed methods approaches*. Sage publications, 2013.

[57] D. Badampudi, "Factors Affecting Efficiency of Agile Planning: A Case Study," Master's thesis, Blekinge Institute of Technology, Sweden, 2012.

[58] D. Wallerstorfer, *Improving Maintainability with Scrum: How Scrum Affects Code Maintainability*. AV Akademikerverlag, 2012.

[59] A. Von Mayrhauser *et al.*, "Program comprehension during software maintenance and evolution," *Computer*, vol. 28, no. 8, pp. 44–55, 1995.

[60] J. Choudhari and U. Suman, "An empirical evaluation of iterative maintenance life cycle using xp," *ACM SIGSOFT Software Engineering Notes*, vol. 40, no. 2, pp. 1–14, 2015.

[61] G. Ernberg, "Scrums effects on software maintainability and usability," 2015.

[62] A. Van Deursen, "Program comprehension risks and opportunities in extreme programming," in *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*, pp. 176–185, IEEE, 2001.

[63] A. Law and R. Charron, "Effects of agile practices on social factors," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, pp. 1–5, ACM, 2005.

[64] D. M. Rafi, K. R. K. Moses, K. Petersen, and M. V. Mäntylä, "Benefits and limitations of automated software testing: Systematic literature review and practitioner survey," in *Proceedings of the 7th International Workshop on Automation of Software Test*, pp. 36–42, IEEE Press, 2012.

[65] S. Chopra, "Implementing agile in old technology projects," in *Reliability, Infocom Technologies and Optimization (ICRITO)(Trends and Future Directions), 2014 3rd International Conference on*, pp. 1–4, IEEE, 2014.

[66] M. Skoglund and P. Runeson, "A case study on regression test suite maintenance in system evolution," in *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pp. 438–442, IEEE, 2004.

[67] R. Borg and M. Kropp, "Automated acceptance test refactoring," in *Proceedings of the 4th Workshop on Refactoring Tools*, pp. 15–21, ACM, 2011.

[68] A. Capiluppi, J. Fernandez-Ramil, J. Higman, H. C. Sharp, and N. Smith, "An empirical study of the evolution of an agile-developed software system," in *Proceedings of the 29th international conference on Software Engineering*, pp. 511–518, IEEE Computer Society, 2007.

Chapter 8

Appendix

Appendix 1: Invitation letter for participation in Case Study

Greetings,
This is Gopi Krishna Devulapally.

Short Introduction:

I am pursuing my masters in software engineering, at Blekinge Institute of Technology, Sweden. I am into my final semester, where I need to conduct a thesis as a part of my course. In this consent I have requested Hrushikesh, VP HR, Capgemini, to conduct it at Capgemini, India. So he has requested Dalal Mayank Associate Director Quality, to assist and mentor my thesis at Capgemini to closure.

Research Objective:

This study aims to conduct a case study at Capgemini, specifically to study Applicability of Agile in Software Maintenance. Further, aims and objectives of this master thesis are presented as follows.

- Identifying the practices of agile which facilitates maintenance process in an offshore team..
- Identify the advantages and disadvantages of adopting agile in an offshore team dealing with maintenance activities.
- Identifying the impact of adopting agile in an offshore team dealing with maintenance activities, in terms of productivity.

Invitation:
Along with my mentor at Capgemini, we have appraoched Hemanshu Ghia to

provide us information about the projects which are using Agile as their delivery model, since the units of analysis for this case study are projects (which have adopted Agile) at Capgemini. From the information provided by Hemanshu about different projects, I kindly request you to participate in the interviews which are planned as a part of Case Study.

Note:

Anonymity of the interview participant is fully preserved.

Regards
Gopi Krishna Devulapally

Appendix 2: Close-ended Questionnaire for First Round Interviews

Rate the usage of agile practices in your project:

Requirement and Design

Agile Practice	Not Used	Low Usage	Medium Usage	High Usage
Product Backlog				
User Stories				
Planning Game				
Task Prioritization				
System Metaphor				

Implementation:

Agile Practice	Not Used	Low Usage	Medium Usage	High Usage
Coding Standards				
Test Driven Development				
Pair-Programming				
Refactoring				
Collective Code Ownership				
Daily Builds				
Continuous Integration				
Code Reviews				
Bug Tracking				

Small Releases				

Testing:

Agile Practice	Not Used	Low Usage	Medium Usage	High Usage
Unit Testing				
Automated Testing				
Acceptance Test				

Process:

Agile Practice	Not Used	Low Usage	Medium Usage	High Usage
Short Iteration				
Release Planning				
Iteration Planning				
Task Board				
Daily Stand-up Meeting				
Velocity				
Demo				
Retrospective				

Team Organization:

Agile Practice	Not Used	Low Usage	Medium Usage	High Usage
Small Team				
Co-located Team				
On-Site Customer				
Scrum Master				

Appendix 3: Open-ended Questionnaire for Second Round Interviews

Documentation

1.Which Technical documents in focus of software maintainability are prepared/used in the project
2.As it is SCRUM, does reduced number of documents affect software maintainability?
3.What is the intensity of usage for those documents? (High, Medium and Low)
4.How often revisions are done to the documents which support software maintainability?
5.Do you think those documents are overhead or does it helps you in tracing the decisions made in the project in regards to software maintainability?

Metrics

1.What role does metrics (which focus maintainability) play in your project?
2.What is the main content of reports which are submitted to the client?
3.What is the review mechanism which takes place in each phase until the project reaches testing phase?
4.How do you track the status of the reviews after the completion of review in each phase?

Software Architecture and its evolution

1.What is process followed in designing or making a decision on architecture? (probe: dependencies among user stories is considered?)
2.What deliverable the process is expected to produce? And who were audience for that deliverable? And how it is shared among them?
3.How often you review architecture?
4.What is the role of architect on software architecture? 5.Key considerations in reviewing the architecture (any checklist is used) or process followed in reviewing it.
6.Who has the authority to make decisions on design? And who reviews and signs it off?
7.Is Software Maintainability considered as quality attribute before designing architecture of the system, i.e. does your design decisions is only dependent on what is to be delivered or it also considers some quality attributes along with delivery of features?
8.What techniques are used for designing the architecture which focuses software

maintainability?
9.Do you practice any impact-analysis for the CR's and defect correcting actions in a sprint for the whole architecture of the system?
10.How the decisions on architecture are shared among the team?

Coding

1.Is code review practiced?
2.Is any process involved in code review?
3.Do you consider code maintainability criteria for Code review?
4.Is refactoring Practiced?
5.On what basis do you refactor the code? (Any checklist), if practiced?
6.Does the time constraint of sprint effect the code quality?
7.Any practice which helps in supporting software maintainability?
8.Does time constraint of sprint effect integration testing?
9.Does the concept of collective code ownership exist?
10.Effort involved in short-gun surgeries (High, medium or low)

Testing

1.Do you practice Test Driven Development concept?
2.How do you test the effect of changes over the system?
3.What are the different activities in relation to testing help in software maintainability?
4.How do you evaluate the effectiveness of unit testing?
5.How do you track Bugs?

Process Improvement

1.In the focus of Software maintainability, how do you plan process improvements?
2.If process improvements take place, what observations do you conduct to baseline the software maintainability of the system?
3.How the quality of the software is planned?
4.Which artifacts of the project are used during process improvement (focus software maintainability)

Social Factors

1.As the teams are self-organized, does losing resources (maintainers) affect the maintenance related work?
2.How do you quantify the knowledge in the team?
3.What is the truck number for the team?
4.Is there any difference in the productivity measurement for maintenance activity and development activity?
5.Do teams use the concept of dedicated maintainers?
6.Does resources (maintainers) availability is affected by development activities?

Planning

1. Explain release plan of maintenance activities for your project?
2. What meetings are held for those release planning? What do you call them?
3. Who are the stakeholders for those meetings?
4. Explain how you plan iterations or sprints in your project for maintenance activities?
5. Who are the stakeholders for planning iteration or sprints?
6. Do have any change management for your team?

Iteration

1. What comes after planning?
2. Explain different things which are done during the iteration?
3. Who creates tasks?
4. How do you start implementing or correcting bugs which are assigned as tasks?
5. How do start implementing new feature or enhancement?
6. Who decides the complexity of the task?
7. Do you have any say on the complexity of the task?
8. How does your estimation vary based on complexity of the task?
9. What is the process for signing of acceptance test for bug fixing and implementing feature?
10. How unimplemented tasks are dealt?

Appendix 4: Validation Survey Invitation

Hello Everyone

Greetings from Karlskrona,

I am getting in touch to kindly request your assistance with a research project. I will be looking at how different teams are affected by adopting Agile Practices during Software Maintenance. So I kindly request you to participate in a validation Survey (A Survey whose purpose is to generalize the results of a study).

Full anonymity of Survey participant is preserved.

Link to Survey:

http://www.survio.com/survey/d/N6H8J3G6B9M8Q4V0P

Please do share it with your colleagues and friends, if you think they are eligible/correct participants for the survey. Your response matters

Thanking you and awaiting your response for the Survey

With Regards

Gopi Krishna Devulapally

Appendix 5: Validation Survey

Validation Survey

Section 1: Background

Name of the Organization	

	Small	Medium	Large
Size of your organization			

	<3Years	>3Years and <5 Years	>5Years
Experience in Software Industry?			
Number of Maintenance Projects you have worked?			

	Corrective	Perfective	Adaptive	Preventive
What type of Maintenance Project you have worked in?				

Section 2: Rate upon Advantages

Question	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Improved Productivity					
Improved team morale					
Merging of bug fixes rapidly					
Improved categorization/prioritization of Tasks					
Cross-training amongst many modules of the code base					
Improved Estimates					
Improved Test suites					
Improves Maintainability Index					
Increase Customer Satisfaction levels					
Improved Code Quality					
Stable design					
Improved Comprehensibility					

Section 3: Rate upon Disadvantages

Question	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Leads to short-gun surgery					
Lacks Comprehensive Documentation					
Increase in entanglements					
Maintenance of Acceptance test suites					
Maintenance of automated test suites					
Requires experienced/competent personnel					
Comprehensibility issues among new developers					
Time wastage due to Standups					

Appendix 6: Codes Overview

Code-ID	Position	Parent code	Code	All coded segments	Activated co...	Author	Documents
112	1		Software Maintenance	0	0	gopikrishna	0
84	2	Software Main...	Advantage	0	0	GOPI	0
93	3	Software Main...	Bug fixes	6	0	GOPI	6
94	4	Software Main...	Categorization/Prioritisation	6	0	GOPI	6
109	5	Software Main...	Change request	1	0	gopikrishna	1
100	6	Software Main...	Code Quality	8	0	GOPI	8
95	7	Software Main...	Cross-training	8	0	GOPI	8
99	8	Software Main...	Customer Satisfaction levels	7	0	GOPI	7
96	9	Software Main...	Estimates	6	0	GOPI	6
111	10	Software Main...	Maintainability Index	5	0	gopikrishna	5
101	11	Software Main...	Modifiability	8	0	GOPI	8
91	12	Software Main...	Productivity	6	0	GOPI	6
102	13	Software Main...	Stable Design	7	0	GOPI	7
92	14	Software Main...	Team Morale	9	0	GOPI	9
97	15	Software Main...	Test suites	8	0	GOPI	8
110	16	Software Main...	Trust	3	0	gopikrishna	3
108	17	Software Main...	Unimplemented work	1	0	gopikrishna	1
85	18	Software Main...	Disadvantage	0	0	GOPI	0
106	19	Software Main...	Standups	6	0	gopikrishna	6
105	20	Software Main...	Customer Involvement	3	0	gopikrishna	3
104	21	Software Main...	New developers	6	0	gopikrishna	6
103	22	Software Main...	Automated Test Suites	8	0	gopikrishna	8
90	23	Software Main...	Entanglements	5	0	GOPI	5
89	24	Software Main...	Implicit requirements	1	0	GOPI	1
88	25	Software Main...	Customer Feedback	7	0	GOPI	7
87	26	Software Main...	Comprehensive Documentation	8	0	GOPI	8
86	27	Software Main...	Short Gun Surgery	8	0	GOPI	8
52	28	Software Main...	Abstraction of Architecture	1	0	GOPI	1
57	29	Software Main...	Dependency	2	0	GOPI	2
67	30	Software Main...	Experience	6	0	GOPI	6
77	31	Software Main...	UAT suites	8	0	GOPI	8

Appendix 7: Screen-shots from MAXQDA

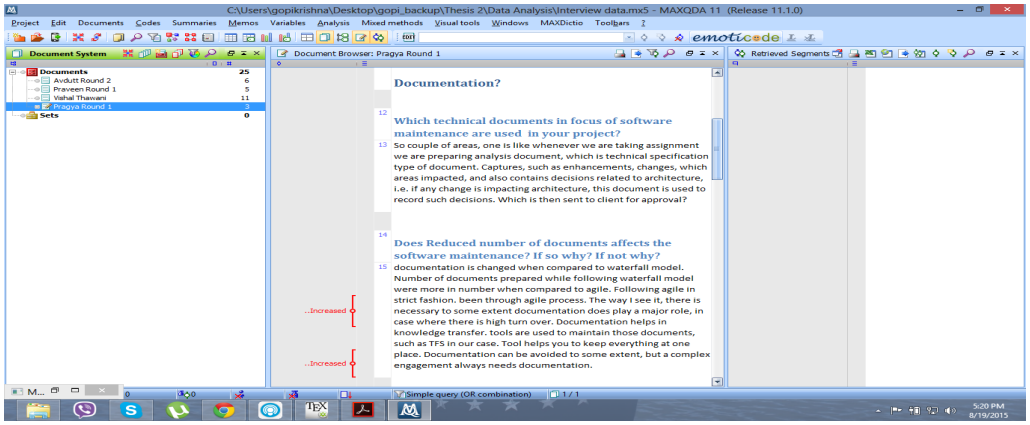


Figure 8.1: Screen Shot for Imported Document in MAXQDA

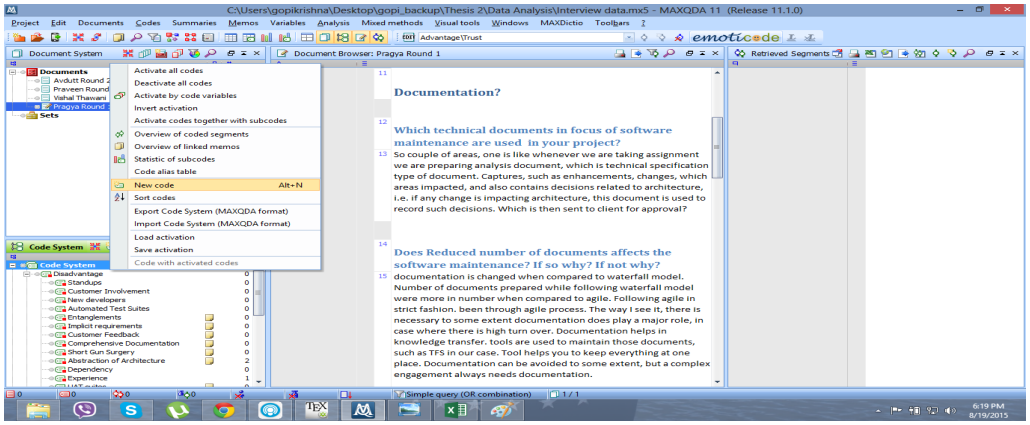


Figure 8.2: Screen Shot for adding code to code system in MAXQDA

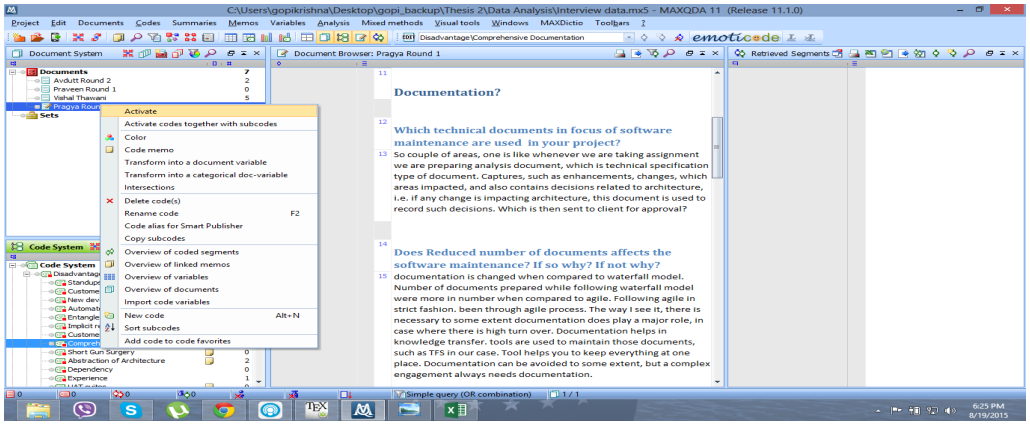


Figure 8.3: Screen Shot for Activating Code from Code System in MAXQDA

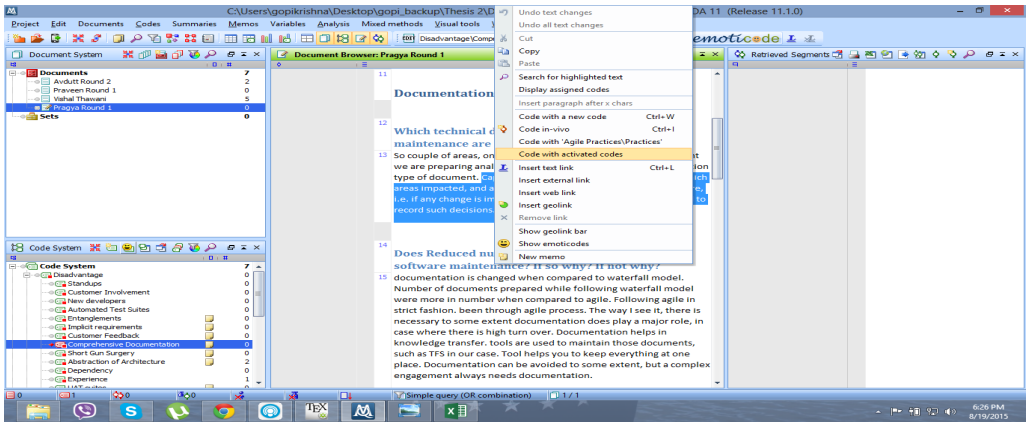


Figure 8.4: Screen Shot for attaching code to text in transcript