Software Quality Framework for Continuous Integration

A Thesis

Submitted in partial fulfillment of the requirements for the degree of Masters of Science in Software Engineering at the College of Computer and Information Science at Prince Sultan University

By:
Saba Talal Hamdan

August 2015
THESIS APPROVAL

Name: SABA TALAL J. HAMDAN
Degree: Master in Science (MSc)
Title of Thesis: Software Quality Framework for Continuous Integration

Examiner Committee:

Dr Suad Alromani
Senior Supervisor/Chair
Assistant Professor
Prince Sultan University

Dr Mohammad Tounsi
Internal Examiner
Associate Professor
Prince Sultan University

Dr Iman Almomani
Internal Examiner
Associate Professor
Prince Sultan University

Date Approved: 24 August, 2015
Dedication

To my mother Madawi Abdullal

To my father Talal Hamdan

To my sisters Reem, Maha, Rasha and Asma’a

To my brother Mohammed

With sincere love and appreciation.
Acknowledgment

First and foremost, all praises and thanks be to Allah for helping me in the completion of this thesis.
I would like to express my deepest gratitude and appreciation to my committee chair Dr. Suad AlRamouni. I thank her for her continuous support and encouragement, as without her guidance and persistent help, this thesis would not have been possible.

I would also like to thank the faculty members of the CCIS department and specially Professor Dr. Ajantha Dahanyake, the MSE program coordinator and my committee members, Dr. Iman AlMomani and Dr. Mohamad Tounsi.

My deepest gratitude and many thanks to Dr. Areej AlWabil for her guidance, I really appreciate her useful tips and suggestions to improve my work.

My greatest thanks and appreciation are extended to my father for his unlimited support and encouragement through every step.

Last but not least, my sincere thanks and appreciation are to my family, friends and everyone who gave me a hand during my studies specially my friends Nahlah AlMashouq, Fahda AlAmmar and Arwa AlBoalayan.
Abstract

This research combines two main areas, the first one is software quality and the second is the agile practice of software continuous integration. Software quality has been an important topic since the beginning of the software development and production. Many researches in the literature have discussed how the quality of software is a critical factor to its success. Because software became an important part of almost every task in our daily life, having high quality software that meets the users’ expectations is important.

Software integration is a stage in every software development lifecycle, it is defined as the process to assemble the software components and produce a single product. It has been shown that software integration and integration testing can make more than 40% of the overall project cost, so it is important that they are done efficiently and easily to be able to manage the involved risks.

A software engineering practice called continuous integration (CI) was introduced by Kent Beck and Ron Jeffries to mitigate the risks of software integration, enhance its process and improve its quality. In this thesis, the principles of CI are identified and applied to a case study in order to analyze their impact on the software development process quality factors.

The overall results of this research showed an improvement in the quality factors of the software development process, such as the development time and the bug fixing time, as opposed to software components being integrated in a traditional way. Continuous integration of software components also increased the customer satisfaction level by allowing software providers to deliver parts of the software in less time.

The contributions of this thesis in the domains of software quality and software integration are twofold; introduce the application of the continuous integration practices to a project case study and examine their impact on the software development process quality factors.

Keywords
Continuous Integration, Software Quality Framework, ISO, Agile, Extreme Programming, Software Development
ملخص:

إطار عمل لجودة البرمجيات عند ربطها باستمرارية

تتطرق هذه الدراسة إلى موضوعين مختلفين، الموضوع الأول هو جودة البرمجيات، والموضوع الثاني هو الممارسات المتعلقة بربط البرمجيات باستمرارية. لطالما كان موضوع جودة البرمجيات من أهم المواضيع منذ بداية تطويرها وانتاجها. كما أنه يوجد العديد من الدراسات والأبحاث السابقة المتعلقة بهذا المجال لما له من أهمية ودور حاسم لإنجاح البرامج. ولأن البرمجيات أصبحت جزءاً مهماً من معظم الأعمال اليومية فإنه من الضروري أن تكون ذات جودة عالية تلبية لمستوى توقعات المستخدمين.

يعتبر ربط البرمجيات مرحلة من دورة حياة تطوير أي برمجية، ويرجع على أنه عملية تدمج مكونات البرمجية و التي تم تطويرها على حدة، لتكوين منتج برمجي موحد. يذكر أن هذه المرحلة ومرحلة اختبار نجاحها تستغرقان ما يقارب 40% من إجمالي زمن إتمام تطوير البرمجية لذلك فإنه من المهم الحصول على كفاءة وسهولة أداء هاتان المرحلةن وإدارة المخاطر المتعلقة بهما بشكل جيد.

لتحسين عملية ربط البرمجيات وجودتها وإدارة المخاطر المتعلقة بها، قام كينت بيك ورون جيفري بطرح ممارسات الربط باستمرارية والتي تفترض ان يتم ربط عناصر البرمجيات باستمرارية طوال فترة التطوير وان لا يترك لها بعد انتهائها.

في هذه الرسالة، تم دراسة التغيير الحاصل في مواصفات جودة عملية تطوير البرمجيات عندما يتم ربط العناصر المكونة لها باستمرارية مقارنة بربط عناصر البرمجية تقليديا بعد انتهاء مراحل التنفيذ بالكامل. وعند مقارنة النتائج في الحالتين، تم التوصل إلى أن الربط باستمرارية يحسن بعض مواصفات جودة عملية تطوير البرمجيات مثل الوقت المستغرق في التطوير، ووقت المستمر لتصحيح الأخطاء البرمجية ومستوى رضا المستخدمين حيث أن الربط باستمرارية آتى بإصالت البرمجية إلى المستخدمين في وقت أقرب.

ضمت هذه الرسالة إلى مجال جودة البرمجيات وربطها جزآن أساسيان: طرح عملي لممارسات الربط باستمرارية ودراسة أثرها على مواصفات جودة عملية تطوير البرمجيات.

مفاتيح البحث:

الربط باستمرارية، إطار عمل جودة البرمجيات، أيزو، البرمجة الرشيقة، البرمجة القصوى، تطوير البرمجيات.
3.6.2 Applying Continuous Integration ................................................................. 35
  3.6.2.1 Current Development Process .............................................................. 36
  3.6.2.2 The Process After Applying CI Practices ............................................. 37
3.7 Quality Attributes ...................................................................................... 39
3.8 The Framework ......................................................................................... 43

CHAPTER 4: Result and Evaluation ................................................................. 45
  4.1 Introduction ............................................................................................... 46
  4.2 Changes in Quality Factors .................................................................... 46
  4.3 Resulting Software Quality Attributes Measures .................................... 48
  4.4 Thesis Result Summary .......................................................................... 53
  4.5 Guidelines and Best Practices ................................................................ 54

CHAPTER 5: Conclusion .................................................................................. 55
  5.1 Introduction ............................................................................................... 56
  5.2 Thesis Summary ...................................................................................... 56
  5.3 Contribution to software Engineering .................................................... 57
  5.4 Future Research Directions ................................................................... 57

References and Appendixes ........................................................................... 59
References ....................................................................................................... 60
Appendix A: ...................................................................................................... 65
  Case study approval letter ............................................................................ 65
Appendix B: Sample pseudo-code of the implemented requirements .......... 66
Appendix C: Sample screenshots of the test cases and test execution result .. 67
Appendix D: Testing results ........................................................................... 68
Appendix E: Related published paper ............................................................. 69
List of Figures

Figure 1 – Software Development Lifecycle Phases......................................................... 2
Figure 2 – The differences between using waterfall and agile ........................................ 7
Figure 3 - Phases of conducting the literature review .................................................... 14
Figure 4 - Continuous Integration Model ........................................................................ 17
Figure 5 - ISO 9126 model............................................................................................... 19
Figure 6 - Dromey quality model..................................................................................... 21
Figure 7 - Development process before applying continuous integration ...................... 37
Figure 8 - Development process after applying continuous integration ......................... 38
Figure 9 - Quality attributes of the development process ................................................. 40
Figure 10 - Software Quality Framework for Continuous Integration ......................... 47
Figure 11 - The cost of change in traditional SDLCs......................................................... 52
Figure 12 - The cost of change in agile SDLC ................................................................. 52
List of Tables

Table 1 - Collected references number per data source ...................................................... 16
Table 2 - CMM model levels .................................................................................................. 22
Table 3 - Description of the agile practices ......................................................................... 26
Table 4 - Measures of quality attribute .............................................................................. 40
Table 5 – Framework’s metrics of measuring quality ......................................................... 41
Table 6 - Quality before and after applying continuous integration .................................... 46
Table 7– Summary of the result of measuring each quality attribute: .................................. 53
## Abbreviations and Terminologies

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Build</td>
<td>A constructed pre-release version of a program</td>
</tr>
<tr>
<td>CI</td>
<td>Continuous Integration</td>
</tr>
<tr>
<td>CVS</td>
<td>Concurrent Versions System</td>
</tr>
<tr>
<td>SDLC</td>
<td>Software Development Lifecycle</td>
</tr>
<tr>
<td>SVN</td>
<td>Subversion – A software versioning and revision system, used to maintain current and historical versions of software files</td>
</tr>
<tr>
<td>VCS</td>
<td>Version Control System</td>
</tr>
<tr>
<td>XP</td>
<td>eXtreme Programming – lightweight software development methodology that responds to changing customers and intend to improve software quality</td>
</tr>
</tbody>
</table>
CHAPTER 1: INTRODUCTION
1.1 Introduction

Software integration is the practice of linking together subsystems or components of software to produce a single unified product [36]. It is a phase of the software development life cycle where different parts of the software are put together after they are implemented (See figure 1).

![Software Development Lifecycle Phases](image)

Figure 1 – Software Development Lifecycle Phases

Software integration can fairly be an easy practice when one developer implements all parts of the software. But when the system is large and more than one developer is working on it, integrating their components together becomes drastically complex [11],[19],[37]. This complexity of integration is known in the software industry as the “integration hell” [1][11][12][15][37]. It is a problem that happens typically when integration is left to a later stage of the development life cycle, after all components are fully implemented and are ready to be put together for testing. The practice of Continuously Integrating software was introduced to overcome this problem [12][17].

Continuous Integration was first published in 1998 by Kent Beck and Ron Jeffries. It is a software development practice that was proposed as one of the main twelve primary Extreme Programming (XP) practices [6][19]. It basically means that each developer integrates his/her work continuously (at least once a day). This practice assures that small parts are added immediately once they are implemented and are ready to be a part of the system, before they become large and complex and before their integration process becomes lengthy and complicated [4][17]. This practice reduces the cost and time of retesting the features of the software every time it is rebuilt [11].

Applying the practice of Continuous Integration requires automating the test cases every time part of the software is added and making sure they cover the whole system
including the newly added parts[8][12]. It also requires that the software gets tested and built automatically and the testing result is sent immediately to the responsible developer as soon as possible (mainly in the next working day). This helps in identifying bugs and issues while the developer’s knowledge about his/her codes is still fresh [8][11][37].

Continuous integration does not only solve the problem of “integration hell”, but it also has many other benefits. As a result of integrating continuously, defects and interface mismatches will be found earlier so that they are fixed in time. It is much easier to detect and solve the causes of defects when they were created few hours ago than if they were created weeks or months ago! [5][17]. Also, finding those defects in an early stage of the software development life cycle makes fixing them less costly and requires less effort compared to when they are found at a later stage [26].

Continuous integration also has a very important impact on the quality of software. It affects the software development processes quality thus impacting the quality of the resulting product [15]. Different studies in the literature focus on identifying and categorizing software quality attributes. One of the most important papers on this topic is given in [22], where the author proposed a classification of software quality attributes based on which phase of the software development life cycle they should be mapped to the design. The author gives his proposed classification on the key goals that each software development phase aims to achieve.

1.2 Goal and Motivation

In this thesis, my main focus is to identify and compare quality attributes of software that has an advantage of applying the practice of integrating software components continuously against the software that was being integrated traditionally at a later stage of the software development life cycle.
The main outcome of this thesis is a software quality framework for continuous integration, which provides guidelines, recommendations and best practices for software producers, software developers, software testers and quality assurance teams.

Because of the large shift of software producers towards the agile practices of software development[4], the need of having a quality framework has immerged. The proposed framework in this thesis can benefit analysts, developers, testers and project managers.

In this chapter, the background material will be presented to give an appropriate perspective for the work in this thesis. The chapter contains brief overviews about software quality, software development lifecycles, agile and extreme programming, software integration and continuous integration. The practices of continuous integration will also be presented in this chapter and finally a brief about the relationship between software quality and the continuous integration of software.

1.3 Background

This section includes the necessary background material to give an appropriate perspective for the work done in this thesis.

First, it discusses the quality and quality assurance areas. Then it includes definitions about the software development lifecycles and specifically agile. Finally it includes a section about the integration phase of the software lifecycle and the different methods of integrating the software parts and how it affects the software development process quality.

1.3.1 Quality

It is defined as a matter of products and services whose measurable characteristics satisfy a fixed specifications defined beforehand. The term quality also means meeting customer needs and expectations[27]. As defined by Oxford dictionary, the word quality means the degree of something’s excellence or the standard of something as measured against other things of a similar kind.
In software engineering, software quality is related to two sets of factors. The first is the functional, which means how the software conforms or complies with the functional requirements and specifications defined by “what” the user wants the software to perform. The second one is the non-functional requirements, which is defined by “how” the user wants the software to perform. Examples of non-functional requirements are the software availability, reliability, efficiency, and usability etc. According to [13] and [25] software quality, besides confronting to the above two set of factors, also means freedom from deficiencies.

For agile software development, different meaning of quality assurance was defined by [33] as the development of software that can respond to change whenever the customer requires it to change.

Software quality assurance is a very important part of software projects. The success or failure of software projects depend mainly on how well it fulfills the quality requirements of the software and meet its stakeholders’ expectations [18][50].

1.3.2 Software Development Lifecycle (SDLC)

Software development processes to produce software systems are known as the software development lifecycle SDLC. Software development lifecycles usually consist of many phases. Traditional SDLCs are usually plan-driven. The most known and basic SDLC is the waterfall. It is consisted of six main phases where it starts with analysis, design, implementation, integration, testing and finally software delivery [46].

The problem with traditional, plan-driven SDLCs is that the validation and verification is left to a stage after all previous stages are finished and done. This makes finding that the software developed is not as required too late! And makes the found defects difficult to fix. Finding defects on a later stage of the development lifecycle requires going back to the phase that introduced the defect. The longer the phase has been done the more difficult it is to go back to it and do the necessary changes [24][26].
Many researches proposed improving ideas to the traditional software development lifecycle, mainly to overcome the late discovery of defects and to make fixing them less costly [9][26]. The most important idea is to use agile software development methodology instead of the traditional software implementation methodology.

1.3.3 Agile and Extreme Programming

Agile software development was introduced in the end of the 20th century [31]. It is a development practice that is able to deal with unstable, changing and volatile requirements [32]. Agile practices are built upon four main techniques 1) simple planning, 2) short iterations, 3) early release, 4) frequent customer feedback [31].

It is well known that producing high quality software in short period of time is the goal of applying agile practices. The main point that differentiates agile compared to other development processes is that it easily accepts business changes during any phase of the software development [32]. This flexibility and adaptability of the agile methods makes them meet the complexity of modern software [29][31].

The agile development process includes a variety of agile methodologies that can be followed such as Scrum, Crystal and Extreme Programming.

Extreme programming is the most widely used method among the agile development methods [31]. It was first introduced in 1997 based on twelve primary practices. [6] The most special feature about extreme programming is that it is concerned about the actual programming practices. This makes it efficient, low risk and flexible [31][48]. Figure 2 shows the differences in the process of software development when following the waterfall methodology and when using the agile methodology.

As seen in figure 2, agile process is more flexible in terms of accepting users’ changes and receiving them informally through stories. Phases of implementation, testing and integration in agile SDLCs are iterative.
1.3.4 Integration

Software integration is the practice of linking together subsystems or components of software to produce a single unified system [36]. It is a part of any software development lifecycle as software is usually developed through different phases and by a team of engineers. Software integration is done in traditional software development lifecycle as an independent step in a later phase, after the implementation of software is completed.
1.3.5 Continuous Integration CI

Continuous Integration is a software development practice that was proposed in 1997 as one of the main twelve primary Extreme Programming (XP) practices [2][6][32][50], namely,

- The planning game
- Small release
- Metaphor
- Simple design
- Testing
- Refactoring
- Pair programming
- Collective ownership
- Continuous integration
- 40-hours week
- On-site customer
- Coding standards

It means basically that each developer integrates his/her work continuously (at least once a day). This practice assures that small parts are added immediately once they are implemented and are ready to be a part of the system, before they become complicated [17].

Applying the practice of Continuous Integration requires that each time a new part is added to the system, automatic test cases are created and added to cover the whole system including the newly added parts. It also requires that the software gets tested and built automatically and the developer whose codes are added receives an immediate feedback about the newly integrated codes[5][8][11]. Any feedback at this stage will be considered by the developer as soon as possible. This helps in identifying bugs and issues while the developer’s knowledge about his/her codes are still fresh [40].

In order for developers to benefit from implementing the practice of CI, they should change their typical day-to-day software development habits. CI requires each individual to commit code frequently, no to commit broken code, fix broken builds immediately, write automated tests, all written tests and inspections must pass, run private builds and avoid getting broken code [8][23]. Each one of these is explained in the following section.
1.3.5.1 Developers’ Continuous Integration practices

In this section, the primary practices of continuous integration will be presented and explained. After applying those practices, the organization is considered to be using continuous integration in its software development process.

1. Commit code frequently
This is the center of continuous integration. Developers should not wait for more than one day to commit their code to the shared code repository. To make this practice easier, developers may do one of the following: 1- make only small changes instead of changing multiple parts all at once. 2- commit after each task assuming tasks are broken up to parts that can be finished in few hours [28][37][39].

2. Do not commit broken code
Broken code is a code that contains any type of failure when it is included in a CI build [34]. Before committing changes to the shared code repository, developers should test their code and build it privately on their machines. They should not commit any code until it passes all tests and inspections.

3. Fix broken builds immediately
A broken build may be an error in compilation, in the database or in the deployment. It is anything that prevents the build from reporting success [15]. The responsible developer should fix such things immediately and the task of fixing a broken build should be the top priority in the project.

4. Write automated developer tests
Tests must be automated in order for them to run in a CI system. It also should cover the whole source code [50].

5. All tests and inspections must pass
For a build to pass, 100% of the automated tests must pass successfully. This is the most important CI criterion regarding the software quality. To assure that the whole source code have a corresponding test case, there are coverage tools that run as a part of the integration build to assist pinpointing source code that does not. There are also other tools which are used to run automated inspectors for checking the general coding and design standards [37][39].
6. Run private builds
CI tools allow developers to have a copy of the software from the shared code repository locally in their workstations. This provides them with the ability to have a recent integration build locally before integrating it to the main integration build server to insure it does not fail [15][37][39].

7. Avoid getting broken code
CI tools help in reporting any broken code once it occurs. Sending feedback to the developers immediately is one of the most important features of CI. All developers are updated with status of the code all the time, if the code is broken, they should not check it out from the shared code repository [40]. Once the feedback is received that the code is broken, the developer responsible for it should be already working on fixing it. Otherwise the benefit of CI’s immediate feedback is lost [11][39][40].

1.3.6 Quality and CI

The Agile software development methods include many practices that have quality assurance potential. Agile practices in general provide many benefits to the quality of software such as:
- Providing better understanding between the developers of the system and the users by using simple “story” of how the system works instead of representing the software requirements formally
- Enabling better visibility to the software project by involving the customers of the project [44].
- Reducing risk by often validating and verifying [44]
- Reducing repetitive work by providing automating parts of the development cycle [16].

1.4 Problem Statement

This research tackles the area of improving the software quality by changing the software development practice of integrating the software parts or subsystems. In the traditional software development lifecycles, integrating software parts is left to a later stage after the
completion of the software implementation. This delay of the integration makes it a complex and lengthy process [17].

Instead of getting into that complexity, software integration becomes continuous as a part of the software implementation phase. This thesis compares between the practices in both cases and the impact of the change on the software development process quality.

1.5 Research Question and Objectives

This study analyzes the current integration process that is done in an organization traditionally and compares it to the integration process when done continuously.

After examining the integration practices, the resulted software will be examined to compare the quality characteristics in both cases and answer the following question:

“How are software development quality attributes impacted by continuous integration?”

The research objectives are:

- To identify the software quality attributes (based on ISO categorization)
- To study the “software development processes” quality factors
- To identify which software development process quality factors are impacted by Continuous Integration
- Test software development process quality factors of CI against traditional process of software integration

1.6 Scope of the Thesis

The objective of this thesis is to examine and study how Continuous Integration affects software quality factors, and then to come up with a framework to define those quality attributes.

As continuous integration is considered a software development practice, the study will focus on the quality criteria at the “development process” which includes all phases of the software development.
The study will include identifying which software quality factors are impacted by continuous integration; this will be defined by comparing it against the software quality factors when traditionally integrating software components. The main focus will be on the behavioral aspects resulted by the software continuous integration as opposed to its technical configuration and setup.

1.7 Thesis Organization

The rest of this thesis is organized as follows: Chapter 2 is a review of the current researches on software quality and continuous integration. It also contains a review of some software quality models. Chapter 3 covers the methodology implemented to apply the experiment done for the purpose of this thesis. Chapter 4 includes the results of actually applying the practices of continuous integration. Chapter 5 covers the conclusion of this work and the future directions.
CHAPTER 2: Related Works
2.1 Introduction

This chapter explores related previous studies conducted in the areas of continuous integration and software quality. The chapter is comprised of two parts, the first one reviews studies on software quality and continuous integration and the second part reviews some of the current software quality frameworks. Before discussing the software engineering topics, an overview of the systematic literature review process, which was conducted for this thesis is presented.

2.1.1 Systematic Literature Review

A systematic literature review (SLR) is a methodology for collecting literature on specific topic in software engineering and conducting evidence-based synthesis of software engineering concepts [51]. This methodology was followed in order to conduct this chapter. The general phases of the methodology were tailored to suit the purpose of this thesis and are presented in figure 3.

This approach facilitates three key phases in software engineering reviews: identification of relevant literature, technologies and best practices, conducting the synthesis, and reporting the results.

It can be observed that although few review studies were identified in which continuous integration in software engineering projects were discussed (Open source[19]
there were none in which continuous integration in the context of financial services for government applications are discussed. The literature search process for this MSc thesis focused on examining peer-reviewed published literature and articles in the domain of software process and integration that were published between 2000 and 2014.

Preferred candidate articles are those which satisfied the inclusion criteria of: 1-conducted through case studies or experiments 2- tackle both the subject of continuous integration and the subject of software quality 3- include the collection and analysis of previous experiences 4- papers with successful methods and best practices. The search methodology also examined the white papers and technical reports which cover the topics of agile software development methodologies and practices, extreme programming, software quality frameworks and software development processes and software quality.

A comprehensive search was conducted using several digital libraries of computing databases, some specific conference proceedings of science and engineering conferences in the past two decades, and Software engineering Journal papers. The selected computing databases are known in software engineering research and practice, to include many of the studies related to Continuous Integration and has been used by similar literature reviews in the software process literature.

The digital libraries included ACM, IEEE and Science Direct. Search terms were used to identify the literature on continuous integration in the digital libraries. The search terms ranged from simple single-word terms such as integration to the more complex Boolean phrases such as “continuous integration” and “quality”. As noted by [51], different syntax considerations were taken into account in the formulation of these search terms in the SLR approach. Table 1 shows the distribution of the collected published literature on the topic listed per source of literature.
Table 1 - Collected references number per data source

<table>
<thead>
<tr>
<th>Resource</th>
<th>Number of Publications</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACM Digital Library</td>
<td>portal.acm.org/dl.cfm</td>
</tr>
<tr>
<td>IEEE</td>
<td><a href="http://ieeexplore.ieee.org">http://ieeexplore.ieee.org</a></td>
</tr>
<tr>
<td>Science Direct</td>
<td><a href="http://www.sciencedirect.com">www.sciencedirect.com</a></td>
</tr>
<tr>
<td>Other (Scholar)</td>
<td>Scholar.google.com</td>
</tr>
<tr>
<td>Total</td>
<td></td>
</tr>
</tbody>
</table>

Broad thematic areas in which continuous integration has been applied in the past three decades range from robotics [30], to open source projects [49] and basic science disciplines such as software systems for chemistry [7].

Software Engineering journals in which research on continuous integration has been reported include Journal of Systems and Software, Information and Software Technology, and Nuclear Data Sheets.

2.2 Continuous Integration and Software Quality

Among the previous researches in the literature, there are many interesting works, which relate in a way or another to the topic of this thesis. In this chapter, others’ contributions will be reviewed.

In [22] the aim of the author was to clarify the term “software quality” and the concept of “good quality” as they might differ based on the different phases of the software development lifecycle.

The author proposed a classification of software quality attributes defined by the ISO 9126 standard of software quality. This classification is based on the phase of software development lifecycle that the quality attribute should be mapped to.
The categorizations in that research will be used in this thesis. As software continuous integration is a practice in the software development phase, the work of the author will immensely help in determining which quality attributes are the focus of the research.

In [39] the author discusses the history of software components integration and how it improved over the years and then provided his recommendations to the current development practices and presented a framework for introducing software continuous integration. His framework includes continuous integration practices, team composition, continuous integration monitoring practices and seven other components. Figure 4 shows the proposed part of software components continuous integration practices proposed by the author.

![Continuous Integration Model](image)

Figure 4 - Continuous Integration Model [46]

In [42] the author answers three primary questions about continuous integration. First, what are the benefits of continuous integration? Second, what are the most beneficial features of continuous integration? And finally, how can a system be improved by using continuous integration?
The research of [42] was carried out in an organization that never used continuous software integration before but was willing to implement this development process. The author introduced the practices of continuous software integration to the organization and presented his findings. The result of the study shows how the organization became more successful and how its performance has improved after applying software continuous integration.

In [23] the main purpose of the article is to explore and explain how the practices of agile development, continuous integration and test-driven development provide high quality and flexibility to software projects. The authors explained the types of projects that benefit the most from continuous integration and the characteristics of the teams that need to apply continuous integration practices.

They also discussed the relationship between the use of continuous integration and the quality management of projects. The article also includes a summary of the benefits of continuous integration practice from a technical perspective, as well as from business perspective.

A study [35] that tackled the same area was published in 2011 to provide a software quality framework for testing mobile applications specifically. The process of the study went in a similar approach as this work by first analyzing the software quality attributes related to mobile applications. And then defining the mobile applications’ development practices and map each practice to the attributes and then finally developing a testing framework for mobile applications that enhances their resulting quality.

The study [47] was conducted to show the lack of a software quality framework whose characteristics incorporate agility. The aim of the authors was to emphasize on the need of an easy to follow quality framework that can work with all agile software development process and can help small agile teams. The authors have recommended some
characteristics for the agile software quality framework and as continuous integration is part of the agile practices, those recommendations were taken into consideration in this work.

2.3 Software Quality Models

There are different software quality models that were developed. Some methods are related to specific kinds of software and some are general. Below are some examples of quality models that are explored from the literature.

2.3.1 ISO 9126 Software Quality Model

The ISO 9126 software quality model is one of the most used in the software industry. This model classifies the quality attributes into four classes: process quality, internal quality, external quality and in-use quality [8].

The model has six important quality classes named: efficiency, usability, reliability, functionality, portability and maintainability.

Figure 5 represents the basic classes of the model:

![Figure 5 - ISO 9126 model [8]](image)
2.3.2 McCall Quality Model

This quality model has three perspectives to identify the software system quality: Product revision, product transition and product operation [33].

- Product revision means the ability to undergo changes. It includes maintainability, flexibility and testability.
- Transition of the product means the adaptability of the software to new environments. This includes portability, reusability and interoperability.
- Product operation is the correctness, reliability, efficiency, integrity and usability.

2.3.3 Boehm Quality Model

This model was developed to address the contemporary shortcomings of models that evaluate the software quality quantitatively. Thus, it attempts to qualitatively define software quality by given set of metrics and attributes [9].

The model structures the quality factors hierarchically into high-level characteristics (representing the basic evaluation requirements), intermediate level characteristics (the expected qualities from the system) and primitive characteristics (providing the foundation for defining quality metrics).

2.3.4 Dromey Quality Model

This model focuses on defining the relationship between the quality attributes and the product attributes. It maps the correctness of the product to the functionality and reliability, the internal properties to the maintainability, efficiency and reliability, the contextual and descriptive properties to the maintainability, reusability, portability and reliability [14].

Figure 4 shows a clearer view of the model:
This model evaluates each software system differently, thus the dynamic process for modeling is required. The main idea is to have a model that is broad enough to work for different systems.

2.3.5 Capability Maturity Models

Those models were introduced by the Carnegie Mellon Software Engineering Institute (SEI). They mainly address the issues of software from a process prospective[38].

The models categorize the software organizations into five levels according to their maturity level. Table 2 shows a summary of what key process the organization should apply in order to reach each level:
Table 2 - CMM model levels [38]

<table>
<thead>
<tr>
<th>Level</th>
<th>Focus</th>
<th>Key Process Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial Level 1</td>
<td>Heroes</td>
<td>None</td>
</tr>
<tr>
<td>Repeatable Level 2</td>
<td>Project management</td>
<td>- Requirements management</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Software project planning</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Software project planning and oversight</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Software subcontract management</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Software quality assurance</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Software configuration management</td>
</tr>
<tr>
<td>Defined Level 3</td>
<td>Engineering process</td>
<td>- Organization process focus</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Organization process definition</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Peer review</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Training program</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Intergroup coordination</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Software product engineering</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Integrated software management</td>
</tr>
<tr>
<td>Managed Level 4</td>
<td>Product and process quality</td>
<td>- Software quality management</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Quantitative process management</td>
</tr>
<tr>
<td>Optimizing Level 5</td>
<td>Continuous improvement</td>
<td>- Process change management</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Technology change management</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Defect prevention</td>
</tr>
</tbody>
</table>
CHAPTER 3: Methodology
3.1 Introduction

In this chapter, the framework methodology is presented. First, the objectives and benefits of defining quality frameworks are identified. The second part shows the relationship between the software quality and the practices of the agile software development. After that, the continuous integration practices are detailed. And finally in the chapter, the case study is described and the quality framework is represented.

3.2 Research Method

This research is conducted based on the meta-analysis of data collected about the results and findings of previous researches in the same research area. A sample project will be developed to aid the analysis of the implemented continuous integration practices resulting in the framework of this thesis. The data will be used to understand the relationship between software quality and applying continuous integration practices. Then, the outcome will be a quality framework for continuous integration.

Quality frameworks or quality models are the processes and criteria that have to be met in an asset. They consist of three main components; inputs, control factors and outputs [38]. Framework inputs include activities, processes, skills, infrastructure, materials and people that an organization deals with to achieve results. Control factors are the management controls that determine the results and affect them. These factors can change the results even if the input factors remain the same. Finally, the outputs which are the quality results of the function of the previous two components.

For the quality framework in this thesis, the input is represented in the continuous integration practices listed in this chapter below. The control factors of the framework are represented in the quality attributes affected by those practices. And finally, the output is represented in the measured degree that the quality attributes are affected by the practices. The output is presented next in the “Result and Evaluation” chapter.
3.3 Benefits and Objectives of Defining Quality Frameworks

In the section below, some points are listed to explain the importance of defining software quality frameworks in general. The next section describes what objectives can be met by having quality frameworks.

3.3.1 The Benefits of Defining a Quality Framework

- Having a defined quality framework allows organizations to use it for assuring and measuring the quality of their software.
- Defining quality frameworks lead to develop and implement common standards and also share good practices among team members and organizations.
- Reduces quality reporting burden among developers of the software.
- Quality frameworks provide quality assurance methods and tools that can be used to monitor and evaluate the quality of software outputs across software development processes and over time.
- It also facilitates the practical implementation of software in organizations specialized in producing software.

3.3.2 The Objectives of Defining Quality Frameworks

The objective of defining a software quality framework for continuous integration is to establish an organized system of methods and tools that can guide individuals in the software producing organization to ensure the quality of the software.

The following are the main quality assurance aspects covered by software quality frameworks:

- Documentation, ensuring that the quality requirements of the software are documented explicitly,
- Standardization of process and development methods, this should be clearly defined and known to the whole software development team members.
- The continuous monitoring, measuring and assessment of correctly implementing quality processes.
3.4 The Relationship Between the Practices of Agile Development and Quality

Before defining the quality attributes, table 3 shows the general impact of each agile software development practice on the software development process.

<table>
<thead>
<tr>
<th>Agile Practice</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>On-site customer</td>
<td>In Agile, the customer is highly involved in the development of the software. Unlike the waterfall model, customers are welcomed to actively refine and correct the software requirement throughout the whole development process.</td>
</tr>
<tr>
<td>Pair programming</td>
<td>This means that two programmers continuously work together on the same code at the same time. This practice can highly improve the software design quality and reduce defects. It serves as an ongoing code review and continuous code inspection.</td>
</tr>
<tr>
<td>Refactoring</td>
<td>This is a disciplined technique for restructuring an existing code by modifying its internal structure without altering its external behavior. The main practice is a series of small behavior preserving transformation. Each transformation is called a ‘refactoring’ although it does a little change but a sequence can produce a significant restructuring. Because each refactoring is small, it reduces the possibility of something to go wrong and does not interrupt the system from functioning during and after each refactoring.</td>
</tr>
</tbody>
</table>
Continuous integration

It means that the team needs to keep the system fully integrated all the time and not only integrates the code once or twice. This practice reduces the time spent on searching and investigating bugs; it also allows the early detection of problems.

Small release

Agile releases are completed in 1 to 2 months maximum. This is very helpful in reducing customer related risks. The customer can monitor the project and provide feedback. Also the customer might find out quality issues at the very beginning and manage the risks after that.

Acceptance testing

This test is carried out after passing all the unit test cases. The difference between acceptance testing in waterfall model and acceptance testing in agile is that in agile it occurs more earlier and more frequently while in waterfall it is only carried out once after the system is submitted to the customer.

Early customer feedback

This is the most valuable characteristic of agile development. As the software releases are short and moved quickly to next development phases, this enables the development team to get the customer feedback early and take any necessary action immediately.

Test driven development

This practice is very important for continuous integration because one test case might run multiple times daily. Automatic testing is faster than manual and less costly, it also consumes fewer resources. However, automated test needs more effort to be designed implemented and maintained.

3.5 Continuous Integration Practices

As the focus in this thesis is continuous integration, this section will detail the main principles of this practice: Continuous integration is not only about automating test or test-driven development, those are parts of the continuous integration process but there are other practices to account for the complete continuous integration picture.
Below is a list of the main continuous integration principles:

- **Using a Versioning System to Manage the Developed Software Source Code**

  Having a versioning control system has been a basic standard for software development teams for the last decades; it keeps all software assets in a centralized location. The purpose of having a versioning control system (VCS) is to manage the changes that are made to the code of the software project. It provides a complete audit trail and history of the code.

  Software projects involve storing too much files, which may be modified concurrently by several people. VCSs provide many features to the developers, for example it allows them to move back and forth between different versions of code using only a single command.

  This prevents the mess caused by the traditional way of versioning: having countless folders with different dates and versions of the code.

  Also, unlike the traditional versioning way, VCSs allow multiple people to work on an item at the same time because the build server monitors the code to determine the changes done to it and then later merges the changes together.

- **Automating the Build of the Developed System**

  This means the ability to trigger and build a clean version of the software through a single action or command. The built version is composed automatically from raw components. Build automation also allows different teams to build their own systems solely from the code that is fetched from their own versioning control repository.

  Having an automated build procedure can save a lot of time on the long term. Instead of the build taking 30-minute to hours it can be reduced to only several seconds.
The same pattern that performs the automatic build task can be used in the production, staging or development environment.

- **Fast Build**
  
  Because build should happen frequently, the process should not take more than 10 minutes or else people will not commit as frequently. The developers should be encouraged to have their code build and check the test results faster.

  The team can choose their trigger that is set for automatic building, for example the trigger can be: “on every merge to the development branch” this should be taken into consideration because the speed of building will also depend on it.

- **Automating the Deployment of the Developed System**
  
  Automatic deployment is the process of consistently pushing software to different environments when a trigger occurs. This also enables receiving the resulting status whether the software has been successfully deployed on the other environment or not.

  Deployment automation can save the development team as well as the clients a significant amount of time. Clients do not have to be offline during the deployment of the software on their system.

  With an automated system also, human errors can be prevented thus mitigating such errors and saving time and effort. Contingency plan of the software is also one of the features of automating the deployment; it provides the ability to rollback to the previous working version as if nothing happened.

- **Versioning of Test Scripts and Configuration Files**
  
  Just like the versioning of the software source code, test groups and configuration files need to be audited and traced as they are being changed and edited frequently. They are kept in a repository in the versioning control system in a way that teams can go back to a previous version.
• **Self-testing Builds**

  There are many different methods for software testing; some of them may involve automated testing like unit test and interface test. Self-testing builds are the next step. Once the team can have tests that can be automated, they can execute those tests all at the same time while performing a build.

  The issue in having those tests is that the task of creating them is not easy, but once they are created they will save the teams huge efforts[32]. The more good tests and the more tests are performed, the greater the awareness of the status of the software being developed. Having self-testing builds differentiates between software development companies being reactive or proactive.

• **Test in a Replica of the Production Environment**

  It is an important practice for a successful agile development to have a testing environment that is similar to the production environment. Teams are expecting their perfectly working software to work as well when released to its users.

  Making sure that the testing environment is as close as possible to production environment enables the build server to build for the same standards. Both environments should have the same version of operating system, libraries and patches, etc.

• **Commit Code Frequently**

  Committing code means creating a new version of the software in the versioning control system. This action creates a point in the history of the code base that developer can jump over to it and view the changes made to reach this point in the software.

  An important note is that not every commit to the VCS is a release to the production. In fact, thousands of commits may compose a single release of the
software to the production. The more frequent the code is committed to the VCS the easier it is for developers to remove, add and merge their work with other developers work. It also makes the developers more comfortable as if something went wrong they could easily start over from when the previous commit was done without worrying because only small amount of changes were made since then.

- **Consolidate Code Daily**
  
  This practice involves building the mainline on the integration server so that it does not become a complicated task later. Consolidation of the code is usually done at the end of the day, as it is better when all the developed features are merged and compiled. Some teams prefer to do it only when a feature is completed.

Some of the benefits of consolidating the code are:

1- It helps developers when they want to merge the changes they have done to the same piece of code. If two developers are trying to change the exact line of code at the same time, it helps them to find out about this situation early so they can decide on an action.

2- It gives the developers visibility on the code that another developer is writing to enhance the process of code review.

3- It helps in tracking and monitoring the performance of the software when in production through the self-testing automated builds.

  It is important to note that when the code changes, the test cases need to be changed accordingly.

- **Availability of Builds**
  
  The builds availability is to ensure that the developers can access them whenever a manual test may be needed. Although automated tests are created and used, there might be some issues that a computer can’t detect.
• **Fast Access of Test Results**

The result of testing should be available to any stakeholder who can determine if the tests have passed or failed such as the developers, managers, users and clients.

Also, the presentation of the test results should be shown in a way that is understandable by its intended person. There are many available tools to support this; they show a view for every user some for determining what is wrong, some for where and others for how to fix it.

**3.6 Project Case Study**

As an example of a project that applied the agile development practices (including continuous integration) I chose a running system that is developed at my workplace. [check appendix A for approval]

When the project first started, it was being developed traditionally, following the waterfall life cycle for several years. But as the industry is shifting to agile and after it has proven its success and suitability with modern projects, the company decided to cope with the change and shift all its processes to agile as well [28][45]. The shifting wasn’t easy and it was done through multiple stages. Not all agile practices were introduced at once; a plan was conducted to prioritize the introduction of the practices to the project. Different studies were found to introduce continuous integration practice differently, such as[1][28][37][39][45]. When a practice is to be introduced, it is first communicated to the owners of the project. The owners have to accept and approve the provided justification of the need for the change or improvement.

Continuous integration is still not among the applied practices in the project but the feasibility studies of applying it were conducted. A continuous integration environment was set by the software development team to show its benefit to the overall processes in the software development lifecycle of the project.
The rest of this chapter includes a description of the software system that the study has been applied to, how it was applied and the differences in the processes before and after applying CI then finally the affected quality attributes in the framework.

3.6.1 Software System Overview

The software project in this study is a financial system that provides payment services. The project has started more than ten years ago and the first working release of it was introduced in October 2004. The system is being developed and run by an information technology company but it is owned by another entity (referred to in this thesis as the project owner). The system follows a service-oriented architecture. Through the years, new services were being added to the system to cope with the new needs and requirements of its users.

When new needs or services emerge, the owners of the project communicate this to the teams working on it. After that, there are several steps to go through before the new requirements get added to the system. The steps are as follow:

1- The business analysis team starts having requirement elicitation sessions with the owners in order to fully understand their needs. The work of the business analysis team takes about three months until they have the new requirements well documented with the possible solutions for the implementation.

2- The business analysts’ documents are then given to the software architects to analyze the most suitable solution among the provided ones for applying the new changes. Many meetings and justifications take place during this phase. Those meetings usually include the business analysis team, the solution architects, the software development manager and sometimes the owners. The result of this phase is usually the software architecture document.

3- The design documents along with the proposed software architecture are finalized and given to the software development team to start the actual implementation. New services usually consist of several requirements. Requirements are grouped together to compose a release of the project.
Release parameters:

In this project, the number of requirements in one release usually falls between 10 and 50 but it can be less or more. For a release to be planned, designed, implemented and then published to users, it usually takes 6 months to a year based on several factors such as the number of requirements, availability of resources and the type of release whether it is an emergency or not. It is the responsibility of the project manager to design the plan of the project phases. Each team manager can give his/her estimation of cost and time to help the project manager in getting as realistic plan as possible.

Considered requirements:

For the purpose of this study, the last published release was used as an experiment. The latest release included replacing an internal part of the system that has been implemented long time ago with a newer developed system. Four requirements of the release were considered for continuous integration by the software development team. The new requirements in this project usually get prioritized and grouped into releases based on too many factors such as the severity, emergency level, age of the requirement etc. In the case of this study, the latest requirements were chosen because of their recency, applicability and suitable size for this experiment. [Sample code algorithm included in appendix B]

Team organization

The software development team consists of the software development manager, four senior software engineers and two juniors.

The initial steps of setting the continuous integration environment included hardware related activities and software related activities. The software development team mainly has to understand the tools required for integrating continuously and what changes should be done to their daily tasks. On the other side, preparing the test cases [sample included in appendix C] and implementing the automatic testing, building and deployment scripts was by the help of another team that is expert with writing Unix scripts to be used on the servers. The build and deployment manager along with other teams were mainly involved in preparing the hardware part of the continuous integration environment. The hardware included a continuous integration server and also included setting up new development and
testing environments to be redundant copies of the production environment. Until the time that this study was made, the continuous integration environment was not fully set for this project because of feasibility and cost matters, only the minimum setup of CI was there to continue with the feasibility study.

The team members involved in the process were given training on an application lifecycle management (ALM) tool of CollabNet called TeamForge. TeamForge is used in agile development environments in order to have a collaborative architecture that manages all the activities in the process of any software project development.

**My role in case study**

To implement the software for the case study, I worked along with two senior developers. I carried out most of the implementation with their guidance and their help in the parts where more than one developer should be involved to get more accurate results.

**3.6.2 Applying Continuous Integration**

The first part of this section explains how the practices of the current software development process are done, and then how those practices are changed when applying continuous integration. The second part summarizes the difference in the environment set up of hardware and software and in the roles of the team members after applying continuous integration.
3.6.2.1 Current Development Process

The current development process goes through multiple stages starting from receiving the new requirements until it is implemented and ready to be deployed on the production environment. The steps below summarizes the description of the process:

1- Software development manager receives the release requirements from the business analysis and software architecture teams and the estimated time to implement those requirements from the project manager
2- The development manager assigns requirements to the software development team members to implement
3- Each member will check out the latest version of software from the CVS to make sure their changes are made to the most recent release of the software
4- Developers will start to implement changes to the release, this task takes several weeks or months based on the number of requirements assigned to them
5- After completing the whole release, the team members will test their work individually and then integrate it manually with other member’s works.
6- After that, when the release passes all tests, the development team will check their codes in the CVS and inform the software development manager with the changed parts of the software
7- The software development manager will ask the build and deployment manager to take the following required actions:
   a- Check out the updated software from the CVS
   b- Create a build with the new changes
   c- Deploy the build into the integration testing environment
8- The integration testing team will be informed about the new added requirements to the software and then will receive a notification that the new changes were deployed to their environment and by now they can start testing
9- A report about the status of the software will be prepared by the testing team, the bugs will be reported back to the software development manager and there will be bug fixing estimated time for the software development team members
10- The process will be repeated until the integration testing team reports no bugs in the release
11- Once the release passes the integration test, the build and deployment manager will again deploy the release into the quality testing environment
12- The quality of the software will be measured automatically and if the reported quality confronts to the service-level agreement (SLA) it will finally be deployed to the production environment

The steps are represented in figure 7:

![Figure 7 - Development process before applying continuous integration](image)

3.6.2.2 The Process After Applying CI Practices

1- The software requirements will be given directly to the software development team members and not through their manager because of the flat organizational structure in the agile software development teams
2- Each developer will take some parts of the requirements and mark them under his/her responsibilities then will start by creating the test cases of those requirements
3- After completing the test cases, each team member will check out the latest code version from the SVN and will start implementing his/her parts.

4- Every few hours at the same day or at least once a day, the developers should commit whatever parts of code they have completed to the SVN.

5- The server will then create a build with the committed changes with minimal human involvement.

6- After that, the build will be deployed automatically to the testing environment which is a replica of the production environment and the created test cases will be run automatically.

7- Once the test results are out, the developers will be informed about the status of their work and then act accordingly.

The steps are represented in figure 8:

Figure 8 - Development process after applying continuous integration
The below list summarizes the necessary changes that have to be done in order to apply continuous integration practices:

- Continuous integration server should be added to execute the integration and build scripts every time a code check-in is made
- Build scripts should be implemented to automate the build process instead of performing it manually
- Test cases are completely automated and can be run every time a change is made
- Testing environment is very similar (if not copies) of the production environment
- Once a function is ready and passed the quality tests it can be put into production without having to wait for other functions
- When rolling back a function, no need to rollback the whole release
- The build and deployment are done automatically with no or minimum involvement of the build and deployment manager
- The test cases are created by the developers and performed automatically without the involvement of the testing team

3.7 Quality Attributes

In this section, the quality attributes of the framework will be presented and the metrics to measure each attribute are then defined. The last part of this section contains the framework of this study.

To identify which quality attributes are affected by the continuous integration of software, the ISO standard definition of attributes was used. In [22], quality attributes were categorized into groups by their relation to the software development lifecycle phase. As continuous integration is considered a development process, the quality attributes under that group are the focus of this thesis.

Quality attributes of the development process are divided into three sub-groups: process quality, input quality and cost model. The process quality includes change
management, documentation quality, test quality and time management which in turn includes the development time, time to market and affordability. Figure 9 is a clearer representation of the development process quality attributes:

![Figure 9 - Quality attributes of the development process](image_url)

Table 4 shows how each related development quality attributes is measured.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Measure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time to develop</td>
<td>Time to finish implementing a feature and have it ready to be tested</td>
</tr>
<tr>
<td>Introduced bugs</td>
<td>The number of bugs in the new code or in affected old parts of the code</td>
</tr>
</tbody>
</table>
Time to deliver | Time to finish implementing and testing a feature and have it ready to be used by the user
---|---
Test quality | Tests should cover all features and detect most (if not all) bugs
Documentation | All the work done by the developers should be included in the document
Change management | Changes to the requirements should be acceptable and does not cost too much time and effort
Cost model | It is represented by the cost of preparing the new setup compared to the amount that will be saved after the applying the new changes.

Metrics are defined as standards of measurement and have been used in the information technology industry to indicate a method of measuring the quality attributes of a particular activity within a project.

Different projects may use different metrics based on the type of the project and the quality attributes to measure.

Table 5 shows what metrics are used in the project of this thesis to measure each of the quality attributes in the software development process.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Factor</th>
<th>Metric</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time to develop</td>
<td>Project owner satisfaction</td>
<td>The time when all features will be done?</td>
</tr>
<tr>
<td>Introduced bugs</td>
<td>Defect density</td>
<td>Number of defects compared to the number of modified LOC</td>
</tr>
<tr>
<td>Parameter</td>
<td>Definition</td>
<td>Comparison</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Time to deliver</td>
<td>Release completion time / number of requirements in the release Vs. The average time of completing a requirement</td>
<td>Comparing the delivery time of the software as a whole vs. the delivery time when done feature by feature to the users</td>
</tr>
<tr>
<td>Test quality</td>
<td>Test effectiveness = Number of defects found divided by number of test cases executed.</td>
<td>Time spent on creating test / time spent on performing test</td>
</tr>
<tr>
<td>Documentation</td>
<td>Document structure and updatability</td>
<td>Document format / maintainability of the document over time</td>
</tr>
<tr>
<td>Change management</td>
<td>Number of accepted change requests in a month / rework “regression” caused by the changes</td>
<td>Acceptance of changes / cost of change</td>
</tr>
<tr>
<td>Cost model</td>
<td>- The cost to get the continuous integration environment</td>
<td>- The cost of something compared to its benefit</td>
</tr>
<tr>
<td></td>
<td>- Effort – work on necessary phases without having to go from the beginning of the process</td>
<td>- Resources / number of times the test was repeated</td>
</tr>
<tr>
<td></td>
<td>- Getting fast feedback</td>
<td></td>
</tr>
</tbody>
</table>
3.8 The Framework

In this section, the software quality framework for continuous integration is presented. The framework consists of four main parts. The first one is applying the continuous integration practices to the software development process. The second one is defining the software quality attributes. The third one is to define the metrics of measuring each software quality attribute. And the fourth and last part is to measure the actual quality attribute.
Figure 10 summarizes the steps in the proposed framework:

Apply CI practices to the software development process:
- Commit code frequently
- Use a versioning system to manage the source code
- Automate the build
- Fast build of the code
- Automate the deployment
- Versioning of scripts and configuration files
- Self-testing builds
- Use similar development, testing and production environments
- Availability of builds
- Fast access to test results

Define quality attributes:
- Time to develop
- Number of introduced bugs
- Time to deliver
- Test quality
- Document quality
- Change management
- Cost model

Define quality metric for each attribute:
- Time to develop - feature implemented and ready to be tested
- Number of introduced bugs - in the new code or affected old code
- Time to deliver - feature tested and ready to be used
- Test quality - test coverage and bugs detection
- Document quality - complete and updated
- Change management - time and effort to apply changes
- Cost model - the cost of the new setup vs. the benefit of it

Measure each attribute

Figure 10 - Software Quality Framework for Continuous Integration
CHAPTER 4: Result and Evaluation
4.1 Introduction

In this chapter, the differences between the metrics of each quality factor are provided, first, theoretically and then the measured results of the software quality attributes in the framework are presented. Finally a comparison of the software quality between the results before and after applying the continuous integration practices is provided.

The recommendations, guidelines and best practices of this study are based on the results presented in this chapter.

4.2 Changes in Quality Factors

Table 6 contains each one of the framework quality attributes and how it differs when the software is integrated traditionally and when integrated continuously.

Table 6 - Quality before and after applying continuous integration

<table>
<thead>
<tr>
<th>Comparison criteria</th>
<th>Before continuous integration</th>
<th>After continuous integration</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Time to develop</strong></td>
<td>Developers work on the whole release requirements</td>
<td>Developers work on a single requirement</td>
</tr>
<tr>
<td><strong>Introduced bugs</strong></td>
<td>As testing is done only after finishing a large part of code, the number of bugs is greater than when testing is done after implementing only small parts of code</td>
<td>Every feature is tested and fixed after it is completed and it does not have to wait until the whole release is finished</td>
</tr>
<tr>
<td><strong>Time to deliver</strong></td>
<td>After finishing the release and testing it</td>
<td>After finishing a single requirement and testing it</td>
</tr>
<tr>
<td><strong>Test quality</strong></td>
<td>Testing is done to the releases as a whole</td>
<td>Testing a feature is done once it is integrated</td>
</tr>
</tbody>
</table>
| **Testing environment is not the same as the production environment** | Small parts are tested individually and testing results are sent immediately back to the developers  
Testing environment is almost exactly the same as the production environment |

| **Documentation** | The release is documented properly  
The release specification is documented before starting the work  
And the development work is documented in detail after finishing | The requirements are documented  
But there is a minimal amount of documents by the developers  
The automatic tools generate statistics and data about the developed features based on the requirements, the testing results and the generated bugs etc. |

| **Change management** | Change is only accepted after a long process and approvals  
Changes are introduced through a whole new patch or release | Change is accepted at any time only by adding new requirements by the owner to the business analysis and software development teams |

| **Cost model** | The time and effort of manually doing the software build and integration | The cost of getting continuous integration server and tools |
4.3 Resulting Software Quality Attributes Measures

The **time of development** factor is measured for the project by how much the system user is satisfied about the time it took to develop the features. Before continuous integration, the development time is the time when the project is in the implementation phase from its beginning until it ends within the software development team. With continuous integration, the development phase also includes testing, build and deployment of the product. Once the developed part is deployed to the testing environment without issues, the development phase is completed for that requirement.

The plan to complete the selected release was to take 7 months. Out of the planned 7 months, 6 weeks was the estimation by the project manager and the software development manager to complete the release implementation, which is equal to 30 days.

When implementing the same requirements and continuously integrating parts of the code, the team spent 26 days to complete this. The most difficult thing that the team may face is to get familiar with the daily integrations. Members of the team had to integrate even their incomplete parts of code without getting integration issues at the end of every working day. Some modifications had to be made to the code after it was committed but this decreased and the team will get used to it as the work proceeds.

The **introduced defects** to the software are measured by the number of new bugs compared to the number of lines of code added. In this case it is assumed that the same code was implemented before and after thus the same number of lines of code (180 lines) were implemented in both cases.

This metric will be measured by considering two things. First, in the case of traditional integration, the defects will not be discovered until the development phase is done and the defects will not be reported to the developers until the testing phase is done.
After that, an initial investigation will be conducted to find out which part of the code is most likely causing the defect and assign the fix of the defect to the developer who wrote that code. The assigned developer then starts to investigate and fix the defect. This whole process takes several weeks. For the same scenario in the case of continuous integration, defects will be reported directly to the developer who wrote the code immediately after the developer checks in the code to the CVS. The developer will start fixing the defect after that without waiting for the defects to be assigned to him/her and also without waiting for other teams to report. As checking in the code to the CVS is done on daily bases, most of the unit-testing defects will be fixed at the same day or maximum the day after. After finishing the release in this case study, two bugs were reported five months after the end of the implementation phase. The estimated time to fix those bugs is 12 days for the two developers who implemented the affected part of code. In the case of continuous integration, the investigation time is deducted and the estimate time for those bugs to be fixed will be 2 days.

The second thing to consider when measuring this metric is the fact that developers will take less time in understanding the cause of the defect when it is reported within the next twenty-four hours of the implementation, unlike the traditional way of integration where the defects are reported after weeks or months of implementing the code.

The **time to delivery** quality is measured by the following metric: release completion time divided by the number of requirements in the release that is for the project when it is traditionally integrated. To measure it in a project that is being continuously integrated, the average time of delivering a requirement will be considered.

Based on the type of the project of this study, every new release is requested to add three or more services to the system. The fact that the system is service oriented made the customer more satisfied when receiving every service immediately once it is done, instead of having to wait for all the other services in the release to be completed.

Based on the estimation of the project, assuming that there was no delay, the release of four services took 7 months thus every service takes 7 weeks to be completed. The resulted
difference between the traditional integration way and the continuous integration in this metric is that the customer will be able to use a service every 7 weeks instead of having to wait to receiving all the services after 7 months!

There are different metrics to measure the test quality attribute of software. One of them is the test effectiveness, which equals the number of found defects divided by the number of executed test cases. In continuous integration, developers are responsible for creating the test cases before starting the implementation.

As reported by the testing team, each requirement should at least have two test cases but it also depends on the complexity of the requirement. In the case of the selected requirements of this study, there were 16 test cases. It took the team five days to create them and only two days to execute them. [appendix C and D contain screenshots of parts of the created test cases and their execution results]

The time spent on creating the test cases is another metric to measure the test quality. In continuous integration, the developers create the test cases before starting the implementation to follow a test driven development (TDD) process. This process results in a shorter implementation time [10][35]. As mentioned above, when creating the test cases by the testing team it took them five days while in the case of continuous integration the time of creating the test cases is included in the 26 days of the implementation phase.

The metric of measuring the software documentation quality is usually about how much does the documents cover about the system and how well formatted it is. As known about agile software development methodology it has less importance for documentation than the other traditional software development methodologies [6][41]. For this project, the system documentation was not much affected. The same templates of documentation were used in both cases when using or not using continuous integration.

The most important benefit is that in continuous integration, the software documentation is done continuously as new features are implemented. When the documents
are immediately updated and reflects the actual progress, it improves the software visibility to the stakeholders.

For the project in this case study, the owners request to have detailed documents from every team working on the project so for the owners to accept the shit to continuous integration, the documentation quality should not be reduced.

The **change management** can be the most important quality factor when it comes to agile software development compared to traditional software development. The metric to measure the quality of change is equal to the number of accepted change requests divided by the number of closed requests in a month * 100%.

**Cost** of projects is usually measured in terms of effort and time. Some kinds of projects also consider the spent cost of the used tools, software and hardware in the process of developing the project. To apply continuous integration, the cost of adding the CI server to the development environment should be considered as well as the cost of getting a software development lifecycle management tool. Also, there is the cost of replicating the environments to be similar to the production environment.

Although this will cost the organization, but it will save time, effort and man-days in the future. The cost of having independent testing, build and deployment employees will be saved as testing will be done by the development team and the software build and deployment will be automated.
As represented in the figure 11, the cost of change if using traditional software development process is acceptable early in the lifecycle of the project, when the team is still in the phase of gathering requirements. As the project proceeds, the cost of change becomes drastically high and infeasible.

During the agile development process, requirements are being collected continuously throughout the lifecycle of the software this makes it easy to adopt changes at anytime.[29]

As shown in figure 12, the cost of change is linear and stable almost during the whole process. The metric to measure the quality of change is equal to the number of accepted change requests divided by the number of closed requests in a month * 100%.
4.4 Thesis Result Summary

This section represents the quality measures of the software development process attributes with and without continuous integrations.

Table 7 – Summary of the result of measuring each quality attribute:

<table>
<thead>
<tr>
<th>Quality attribute</th>
<th>Measure without CI</th>
<th>Measure after CI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time to develop</td>
<td>30 days</td>
<td>26 days</td>
</tr>
<tr>
<td>Introduced bugs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of bugs</td>
<td>2 bugs</td>
<td>2 bugs</td>
</tr>
<tr>
<td>Fixing time</td>
<td>12 days</td>
<td>2 days</td>
</tr>
<tr>
<td>Time to deliver</td>
<td>The whole release in 7 months</td>
<td>One service every 7 weeks</td>
</tr>
<tr>
<td>Test quality</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of test cases</td>
<td>16 test cases</td>
<td>16 test cases</td>
</tr>
<tr>
<td>Creation time</td>
<td>5 days</td>
<td>0</td>
</tr>
<tr>
<td>Execution time</td>
<td>2 days</td>
<td>0</td>
</tr>
<tr>
<td>Documentation</td>
<td>Detailed and high quality</td>
<td>Remain the same</td>
</tr>
<tr>
<td>Change management</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Cost model</td>
<td>Build and deployment manager</td>
<td>CI server</td>
</tr>
<tr>
<td></td>
<td>Testing team</td>
<td>New ALM</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Hardware upgrade</td>
</tr>
<tr>
<td>Total days</td>
<td>49</td>
<td>28</td>
</tr>
</tbody>
</table>
4.5 Guidelines and Best Practices

For agile software producers whom are not applying the practices of continuous integration it is recommended that they start implementing it as a part of their software quality improvement plan. The process of applying the continuous integration practices can be quite easy once the suitable tools are found. After setting up the continuous integration system, it is good if the teams start using its features gradually as experienced. Overtime, the way of practicing continuous integration will improve and the team members will get more comfortable with applying them.

Shifting the traditional software integration to continuous integration has shown an improvement to the overall software quality. In addition to the benefits to software quality itself, continuous integration has also affect the development processes of the software testers and developers. Test cases are created by the developers before the code implementation and are run every time code integration is made. This gives the developers clearer view of the implemented software and saves them a lot of the bug investigation time.

It is important that the development team members commit to the practices of continuous integration for the organization to benefit from applying it. There were some difficulties in the beginning with some team members, specially those whom are used to the traditional daily practices of software development for a long time. But after getting to understand how the continuous integration works, the flow will improve.

Continuous integration also eases the tasks of project managers. It gives them higher visibility so that they can make more accurate estimations regarding the system development time, effort, needed number of resources etc.
CHAPTER 5: Conclusion
5.1 Introduction

This chapter consists of three sections. The first section summarizes the work of this thesis. The second section explains the contribution of this thesis to the field of software engineering. The third and last part includes some recommendations and future research directions.

5.2 Thesis Summary

Software producers are shifting their development practices towards continuous integration as it showed a significant improvement to the overall software quality. Many of the risks involved in the software integration process were mitigated as a result of integrating parts of the developed software continuously once it is ready.

By checking in the developed code to the version control system, defects are detected earlier and an action is taken immediately rather than discovering detects in a later stage of the software development lifecycle.

Continuous integration helped in measuring the health of the software while still being developed because tests are automated as a part of the integration process. When the health status of the software is measured continuously, corrective actions are efficiently taken before the defects are accumulated.

When a defect is found after a part of the software is integrated to a tested development environment, it limits the scope of investigating the cause of the defect only to the newly added part. This helps find and fix the defect more efficiently and in less time.

This work represents a comparison of the software quality attributes before and after implementing the continuous integration practices. The developed software for this thesis was a part of a system that follows the service oriented architecture. Finally, the outcome of this work is a software quality framework that includes recommendations to the software
producers whether to apply continuous integration practices and benefit from improving the software quality or not.

5.3 Contribution to software Engineering

The result of this thesis includes a framework to improve the quality of agile software systems being integrated continuously. Adding research to the area of agile programming and continuous integration copes with the shift of the worlds’ software developers towards it. Researches that prove and describe the benefits gained by the continuous integration of software can encourage and motivate software producers to apply agile practices and improve the quality of both their processes and products.

This research provides a practical experience for an organization that is currently in the process of shifting from waterfall software development to agile software development. It explains particularly the introduction of the continuous integration practices to the development process.

The lessons learned can be useful for any software producing related person such as analysts, developers, testers, project managers and clients.

5.4 Future Research Directions

In this research, the focus was only on the continuous integration practices of agile software development. To continue with this research in the future, other agile practices may be considered such as pair programming and combine the affected quality attributes in one quality framework.

An addition to this research, more activities that occur during the software development lifecycle might also be studied to understand their impact on the overall software quality factors. An example of those activities is including the affect of accepted change requests to the resulted software quality framework.
It is also good to consider applying a similar study to a complete release and to identify the changes to the quality attributes.

Further work may be added about the difficulties and challenges in introducing continuous integration to a current development process. It is not easy for developers (specially experts) to change their daily activities and to adapt new practices.
References and Appendixes
References


Appendix A:  
Case study approval letter

To whom it may concern,

I, Saba Hamdan, a Software Engineering Master student at Prince Sultan University, am kindly seeking your approval to use the software and hardware tools in the development environment to analyze the software development practices data for the purpose of conducting my Master’s thesis.

The thesis is under the title of “Software Quality Framework for Continuous Integration,” supervised by Professor Suad Alramouni.

All the data and information collected will not impose any obligation of confidentiality and will only be used strictly for the purpose of this study.

If you find this request acceptable, please sign below.

Your kind consent is appreciated.

With Regards,

Saba Hamdan
Jr. Software Engineer

Approved By:

Abdulaziz AlAfsaleq
Operations Director
Appendix B:

Sample pseudo-code of the implemented requirements
Manipulated pseudocode of the original implemented code:

The requirements selected in this thesis are a part of the services of the latest release. The release was created in order to add a new functionality and to improve the previous calculation method. The below is a general algorithm representation.

```pseudo
//replace old code component
import necessary libraries
remove connections to old code
remove links to old code

{ implement enhanced function
  new more efficient formulas of calculation
}

add connections to new code
add links to new code

//add new payment service
import necessary libraries

{ implement new payment function
  function rules are implemented according to the new requirements
}
```
Appendix C:

Sample screenshots of the created test cases and test execution result

Test cases:

After implementing the code, test cases are created and run automatically to verify that the code conforms to the requirements. The below image contains a sample of three test cases created for the requirements selected in this study.
Appendix D:

Testing results:
After completing the test cases, they are run on the code and a report is automatically created indicating the status of each test case. The screenshot below contains a sample of four test cases results.
Appendix E:
Related published paper: