Walden University

COLLEGE OF MANAGEMENT AND TECHNOLOGY

This is to certify that the doctoral dissertation by

Ajay Gawali

has been found to be complete and satisfactory in all respects,
and that any and all revisions required by
the review committee have been made.

Review Committee
Dr. Raghu Korrapati, Committee Chairperson,
Applied Management and Decision Sciences Faculty

Dr. Reza Hamzaee, Committee Member,
Applied Management and Decision Sciences Faculty

Dr. Walter McCollum, University Reviewer
Applied Management and Decision Sciences Faculty

Chief Academic Officer

Eric Riedel, Ph.D.

Walden University
2012

Abstract

Impact of Agile Software Development Model on Software Maintainability

by

Ajay R. Gawali

M.Sc. Physics, University of Pune, India, 1991

B.Sc. Electronics, University of Pune, India, 1989

Dissertation Submitted in Partial Fulfillment

of the Requirements for the Degree of

Doctor of Philosophy

Applied Management and Decision Sciences:

Information Systems Management

Walden University

May 2012

Impact of Agile Software Development Model on Software Maintainability

by

Ajay R. Gawali

Dissertation Submitted in Partial Fulfillment

of the Requirements for the Degree of

Doctor of Philosophy

Applied Management and Decision Sciences:

Information Systems Management

Walden University

May 2012

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

Abstract

Software maintenance and support costs account for up to 60% of the overall software life cycle cost and often burdens tightly budgeted information technology (IT) organizations. Agile software development approach delivers business value early, but implications on software maintainability are still unknown. The purpose of this quantitative study was to better comprehend the impact of the Agile development approach on software maintainability. The study drew on resource dependence and Lehman's software evolution theories. The research questions for the study examined (a) the impact of the Agile software development model on software maintainability and its components- software analyzability, changeability, stability, and testability; and (b) the extent of the influence of the Agile approach, characterized by test-driven development, refactoring, continuous integration (CI) on software maintainability and its components in U.S.-based IT organization. Software source code data from a specific project were collected from 61 software iterations of software system developed using the Agile model. The quantitative study employed the analytic hierarchy process and multiple regression technique for data analysis. Results of this study revealed that the test-driven development explained 17% of variation, whereas refactoring accounted for 11% of variation in software maintainability. The CI factor was found to be statistically insignificant. This study contributes to positive social change by ascertaining the impact of Agile approach on software maintainability, further benefiting the Agile project management, Agile advocates, IT operation management, Agile practitioners, businesses adapting Agile development approach, and by contributing to the possible reduction in software maintenance efforts and cost due to improved software maintainability.

Dedication

Kudos to my wonderful friends and peers at Walden University for their inspirational words and encouragement throughout this journey! My spirited school and college teachers, electronic multimeter gift from my father, wise sponsorship of my grandfather for 80286 based PC, friendship of Suresh Nair, Amma's divinely hug, Dadaji's addictive wisdom, Babaji's blissful eyes, Krishna's comforting smile, fruits of our vegetable garden, many known, and unknown critics are some of the greatest beads and moments in the garland that I owe to this milestone.

I dedicate this study to my family. Thanks to my parents, brothers, sisters, and most importantly my wife, Jayashree for her forbearance and kind support. Especially, I dedicate this work to my children, Om and Soham who gave up some of our quality time while I pursued this research. I am with you again!

Acknowledgements

The completion of a dissertation is not a solitary endeavor. It is accomplished with patient support and guidance from many great souls. My sincere thanks to committee chairperson, Dr. Raghu Korrapati provided me with insightful feedback and expertise for all aspects of this research study. I also thank Dr. Reza Hamzaee for his invaluable advice on research design, and Dr. Walter McCollum for his critical review of the dissertation! Their contribution was significant to achieve the intended level of quality in this study.

Many thanks to my colleagues at Intel Corporation - Mike O'Hair, Andy Jeffrey, Miguel Oviedo Bonilla, Jay Turpin, Greg Clark, Hassan Vishwa for providing active feedback throughout this study. Importantly, thanks to my managers Bruce Epling, Gary Nielsen, and Robert Benavidez for their outstanding and instrumental support throughout this quest! Intel is THE great place to work!

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction to the Study

Introduction

Software development and maintenance are two distinctive yet very well connected phases within the software evolution and life cycle. According to Martin and Osborn (1983), information technology (IT) software maintenance-related costs and efforts within IT organizations across the globe are expensive and repetitive in nature. Maintenance activities are labor intensive and entail programming as well as nonprogramming tasks (Bendifallah & Scacchi, 1987). More maintenance implicates more maintenance efforts for programmers (Hoffer et al., 2008). Furthermore, these activities are often significantly involved and are expensive IT fucntions (Arthur, 1988; Swanson & Beath 1998). Specifically, adaptive and perfective software maintenance activities require 75% effort, whereas corrective maintenance tasks consume about 21% of the maintenance efforts (Bennett & Rajlich, 2000). In addition, 85 to 90% of IT budgets go to legacy system operation and maintenance (Erlikh, 2000).

Although quality is one of the goals within development projects, software developers' efforts are often directed to deliver software on time and within budget. This may often lead to the deployment of software with poor maintainability. The goal of software maintainability must be established during the requirements, software design, and development stages (Kyte, 2011; Pigoski, 1997). Recently, the value of the Agile software development model in delivering software early has been widely acknowledged, but shortcomings of this modern development approach have also been recognized.

Current Agile-driven development literature (Abrahamson et al., 2002; Cockburn, 2001; Fowler & Highsmith, 2001; Highsmith, 2004) is primarily focused on Agile's success in attaining early business value delivery, and there is a lack of the empirical literature that examines the Agile development model's impact on the software maintainability within the postdeployment phase of the software life cycle. Agile-driven software development projects are increasingly reported with success, meeting the time constraint–however, its maintainability assessment is still an open challenge for IT management.

IT organizations often inadequately comprehend the maintenance implications of modern development technologies (Sneed, 1995). When embracing innovation in software development methodologies, IT management needs to examine the implications across the complete software life cycle beyond development. There are few studies conducted in the area of methodological impact on software maintenance (Banker, Davis, Slaughter, 1989) prior to the emergence of the Agile model. Software process innovations are targeted to improve development productivity when the majority of the software life cycle costs are postimplementation (Swanson & Beath 1989). A report from the Royal Academy of Engineering (2004) pointed out that the absence of a link between the project and organization's key strategic priorities is the most common cause of project failure. Software maintenance quality and maintenance cost are two of the key priorities of IT management, and these priorities can not be set outside of a software development project.

Strengthening the Agile values prescribed by Beck and Cockburn (2007), Information Technology Infrastructure Library (ITIL) practitioners recognize an immediate issue with attempting to design services from the operations standpoint (Johnson et al., 2003). The issues are with the importance of the process being diminished, the possible high frequency of code change, contracts becoming less important, and change management becoming a burden or bureaucratic hurdle (Highsmith, 2002). Another issue is that the comprehensive operational management support required for the completed service cannot be determined until all of the incremental iterations are executed. Johnson and Higgins (2007) argued that none of the Agile methods makes it easy for operational management to align appropriate process interfaces. Johnson and Higgins suggested cyclic assessment of changed demand of IT infrastructure, including prediction of the final level of required application software support. Furthermore, on the operation side, the ITIL framework is too generic and offers a broad overview of the processes. For instance, ITIL change management processes do not transfer easily to software development processes (Johnson & Higgins, 2007). With the lack of specific procedural guidance and alignment for both Agile developers and software maintenance management, today's IT management is continually challenged to develop and maintain a quality software product.

The Agile model advocates and emphasizes close collaboration efforts among the stakeholders. Johnson and Higgins (2007) posited that the software developers and IT operational managers are aware that IT services provided to customers are dependent on

both parties understanding the customer requirement, understanding mutual constraints, and working together to deliver the best possible IT software and services. With the same understanding, Johnson and Higgins (2007) further highlighted the need for including software maintenance management in the design and development stages to ensure that legacy services are correctly assessed and decommissioned. With the rapid changes in technologies, supportability, maintainability, and associated maintenance cost of the software application could change over time.

Some studies have attributed the increase in software maintenance to the increase in software complexity. The culpable factors for higher software maintenance spans from poor design, unskilled resources, lack of adequate documentation, older technologies, lack of proper change management controls, and poor software development project governance (Deklava, 1992; Osborn, 1989). For instance, Kemerer and Slaughter (1999) have argued that many of the problems in software maintenance are caused by a lack of knowledge of the maintenance process and of the cause-effect relationship between software practices and software maintainability.

Gilb's (1988) evolutionary development method of short release is the original key practice of extreme programming (XP), which is known as *incremental development* and allows software systems to evolve and refine based on the feedback from each release. Cohen et al. (2003) argued that software maintenance is really the normal state of an XP software development project wherein the project evolves over time because of frequent iterations. Stafford (2003) also viewed the first iteration as the initial release

and all following iterations as the *maintenance stage* of the development cycle. Another practice of on-site customer is now referred to as *real customer involvement*, and the development team expects the inclusion of customer's knowledge of the real environment into the development iterations (Keenan & Bustard, 2006). Understanding the implications of these and other key Agile practices as a part of the Agile Software Development Model or ASDM on software maintainability is an important research need that is discussed and examined through this study.

Concerns related to the higher cost involved in software maintenance have led Kremer and Slaughter (1999) among others (Basil et al, 1996; Belady & Lehman, 1976; Gefen & Schneberger,1996; Tamai & Torimitsu, 1992; Yuen, 1985), to study software evolution and development function in relation to software maintenance. These studies have allowed researchers to test and analyze software maintenance efforts and internal maintainability related factors over the software life cycle. Based on this research, the software maintenance phase and development model render itself fit for a study of the impact assessment of Agile model on software maintainability.

Organizations are increasingly under pressure to adapt their business processes in response to relentless technological, organizational, political, and other changes (Davenport & Perez-Guardado, 1999). Agile development methods allow the business organizations to respond to these changes in the business domain by delivering the critical software functionalities early in the development project cycles (Augustine, 2002; Beck, 2002;). The Agile method, with its top variants–XP and Scrum–is garnering strong

advocacy from the software development communities, practitioners in mainstream business organizations from various sizes, and, most importantly, from business management for the embedded agility in its practices.

Resolving the problem of higher software maintenance cost and effort is important because it will aid in understanding software maintenance behavior within software development projects driven by the Agile development model, a model that is gaining maturity and being used widely for its core orientation of faster delivery of a working software application. According to Betz (2007), for systems with unclear and user-facing functional requirements, the Agile methods are the least risky approach to ensure the business value delivery. Agile model in the software development domain offers a set of advantages that have gained it acceptance among the IT practitioners, but it also presents concomitant challenges and peculiar risks to application maintenance management within the IT operation organization. Agile development methods have been implemented in the last 10 years, with the actual incubation process starting in the 1990s in the minds of some software development practitioners who had the clear objective to decrease overall software development time by quickly creating working and reliable software. Everything in software changes–including the requirement, design, business, technology, team, and its team members–and hence the concern should be an ability to cope with change (Beck, 2005). How this software maintenance and support organization translates this enforced change into the stature of adaptable organization as a whole is the key IT management challenge.

Agile methodologies are characterized by attributes of iterative development, constant customer feedback, and well-structured yet flexible teams. Although the objective of the Agile development approach to leverage rapid core business competitiveness is well-disseminated, complete implication assessment of this modern development approach still needs to occur in Agile development, as well as in IT software application maintainability domains to eliminate aleatoric alignment. The use of Agile methods potentially influences key tactical and strategic IT operation effectiveness indicators within application service and support organization. Scant literature exists today that investigates the links between the Agile development approach such as XP and IT software maintenance, support, and IT operation effectiveness, especially within application service and maintenance management domain. The core tenets, practices, and values of Agile have yet to scale and evolve to align with an IT software maintenance and operation management framework. This study offers tutelage for the software maintenance, operation management, and software development organizations and establishes a coterminous framework that leverages software development, maintenance, service, and support management and its cost effectiveness within IT organizations.

Background of the Problem

Software maintenance costs are a significant expenditure, with some organizations allocating 60-80% of their IT budget to maintenance activities (Kaplan, 2002). There are enormous cost and effort pertaining to software maintenance spent across IT organizations while adapting to the Agile software development model to cope

up with the business changes. Within this context, the Agile model's implication on software maintainability has received renewed attention. Beck (2002) also suggested that Agile methodologies founded on Agile principles have received great acceptance because of the increasingly changing business landscape and the pressure on IT organizations to respond to these changes. Huo et al. (2004) found in their analysis multifold outcomes related to Agile methods:

1. Practices followed within agile driven development possess quality assurance (QA) abilities; some are practiced inside the development phase and others are supportive in nature.
2. Agile QA practice occurrence is higher in Agile-driven development projects when compared to waterfall-driven development projects or initiatives.
3. Agile QA practices are followed in very early stages of development.

The understanding of software maintenance relationship with Agile development methodologies will help in developing effective development and maintenance management strategy to control and possibly decrease the software maintenance cost while retaining the Agile model's benefits. The study of software maintainability as a function of a software development model has lagged behind other studies that have highlighted the Agile model's success within development projects' boundaries because of a lack of empirical software maintainability related data. Caun et al. (2010) pointed out that Agile model adoption is constrained by project size, type, project resource

experience with Agile, and availability of knowledgeable and committed customers (Erickson et al., 2005; Fitzgerald et al., 2006).

Software evolution and maintenance is closely coupled with software development process rather than the mere outcome of the development function. Ruiz et al. (2003) presented the ontology for the management of software maintenance projects and highlighted that in recent years researchers have focused their attention on looking for techniques that help to increase the efficiency of the Software Maintenance Process (SPM). One way to improve software maintenance quality and decrease maintenance costs is to reuse previous information and knowledge with effective retrieval (Loof, 1997). It remains unclear how the developmental approach related practices collectively influence the very essential attribute of software quality–software maintainability.

The core issue is that software maintainability is not often a major consideration during software application design, development, and implementation, as indicated in previous studies (Bendifallah & Scacchi, 1987; Schneidewind, 1987). Other studies from Lee (1998) and Balci (2003) uphold the need of controlled design and maintenance process early in the software life cycle to reduce the software maintenance costs. This puts adequate emphasis on the need for optimal alignment between the software development model and the software quality attributes that could provide data-backed intervention strategies to aid IT management. This further can leverage organizations for better software life cycle management as well as end-to-end software evolution process governance. The causal link will further help Agile-driven development organization, IT

management, and Agile project managers, who often fall short in envisioning postdevelopment technical debt, which is evident only in subsequent maintenance release cycles. The low maintainability of software could eventually have adverse implications for the stable, reliable, and efficient business operation, and in turn, on the economy.

Statement of the Problem

The problem addressed in this study concerned the software applications that result into higher software maintenance cost due to lower maintainability at the end of the software development project caused by non-optimal alignment between Agile practices and software maintainability objectives. It is important to control software maintenance cost through the development of quality software systems for business organizational productivity, operational stability, and to promote social and economic progress. When embracing innovation in software development methodologies such as Agile, IT management must examine the implications or impact of the development model across the complete software life cycle that spans well beyond the development phase.

The software development approach influences the software quality including software maintainability, a key software quality attribute. The specific problem, however, is that implications of the Agile development model on software maintainability and its subcharacteristics are still unknown. This void further continues to create misalignment between Agile approach and maintainability objectives and cost IT organizations later in the software application support cycle. When software application development cost is just a down payment (Kyte, 2011), higher software maintainability is

a long-term success factor within software development projects for a software product, argued Moser et al. (2007). Kendall et al. (2010) concluded that the selection of software development methodology is a key decision that should be integrated within the strategic process prior to embarking on a new software development project.

In sum, this study focused on the issue of addressing the alignment between Agile practices and software maintainability objectives. This issue is created due to lack of knowledge about the influence of the Agile model on software maintainability. This study tested the hypotheses that ASDM influences the software maintainability and its subcharacteristics: software changeability, testability, analyzability, and stability.

Nature of Study

The nature of this quantitative study involved assessing the software maintainability of the Agile driven software development system to uncover ASDM's impact on software maintainability characteristics. This study evaluated the impact of the ASDM-related independent variables characterized by TDD, Refactoring, and CI on SM in terms of SA, SC SS, and ST.

SM and subvariables (SM: SA, SC, SS, ST) = f (TDD, REFR, CI).

The above equation is revisited in Chapter 3 with additional detail in conjunction with the Analytic Hierarchy Process (AHP). The final maintainability was calculated as a sum of the weighted dependent variables. The multiple regression analysis was conducted to test the stated hypotheses. Leedy and Ormrod (2005) advocated quasi-experimental research design for researchers recording measures on dependent variables

for a single group before and after the treatment of the independent variable. In further support, Creswell's (2003) model for quasi-experimental design is shown below depicting the single group (software system in this case) with a treatment of TDD, REFR, and CI as independent variables characterizing ASDM in this case. Software analyzability, changeability, stability, and testability were measured during each Agile iteration during which ASDM is used, based on existing software source code data. The nature of the study thus involved measurement of software maintainability and its subcharacteristics after each iteration driven by ASDM, and hence, this research design is a hybrid approach of quasi-experimental and the time series experimental approach.

Group A O – X – O –X – O – X – O – X – O –X – O – X – O

Figure 2. Single group interrupted Time Series Design, Creswell (2003, p.169) adapted for this study that intend to measure the impact throughout the Agile development project.

This study also embodies the cause-and-effect intent when assessing the Agile model's implication on software maintainability. The research data for this study was retrieved from a single multinational technology manufacturing organization. The purposive data set of the software systems was drawn from its IT organization's source code repository that used ASDM in its software development and evolution phase using either XP and or Scrum methodologies. The need of this nonprobabilistic and judgemental selection was rooted in the research question that is specifically directed towards uncovering the impact of ASDM on software maintainability characteristics. Specifically, source code at each Agile iteration completion during the software

development project driven by ASDM was analyzed to assess the development model's impact on software maintainability and the four subcharacteristics.

Mendes et al. (2009) examined software maintainability measures and found that the most commonly used software maintainability metric was an ordinal scale metric based on expert view as one of many software quality attributes. However, these subjective perspectives-based measures alone are not the right fit for this study. Rather, this study integrates internal quality view on software product quality (Heitlager, 2007) utilizing software/source code properties or attributes and relevant measures listed for each dependent variable.

Research Questions

The purpose of this empirical study was to investigate the impact of the Agile software development model on software maintainability. In congurence with this purpose, this quantitative study was guided by the key research question: How does ASDM impact the software maintainability? In this study, ASDM is operationalized using Test driven devlopment (TDD), Refactording (REFR), and Continuous Integration (CI) as key independent variables. The study thus attempted to answer five emergent subquestions:

RQ1. How does ASDM that is characterized by TDD, REFR, and CI, impact software analyzability (SA)?

RQ2. How does the ASDM that is characterized by TDD, REFR, and CI, impact software changeability (SM)?

RQ3. How does the ASDM that is characterized by TDD, REFR, and CI, impact software stability (SS)?

RQ4. How does the ASDM that is characterized by TDD, REFR, and CI, impact software testabiliy (ST)?

Lastly, the final research question was also examined to strengthen the outcome of this study by providing additional inferences through examination of impact on resultant maintainability characteristic that is a function of SA, SC, SS, and ST.

RQ5. How does the ASDM that is characterized by TDD, REFR, and CI, impact resultant weighted software maintainability (SM)?

Hypotheses

To answer the above research questions, 15 pairs of null hypotheses and alternative hypotheses were tested in this study.

To answer RQ1, the first three pairs of hypotheses were proposed and tested.

$H_0$1: TDD has no influence on the Software Analyzability.

H_A1: TDD has a positive influence on the Software Analyzability.

$H_0$2: Refactoring has no influence on the Software Analyzability.

H_A2: Refactoring has a positive influence on the Software Analyzability.

$H_0$3: CI has no influence on the Software Analyzability.

H_A3: CI has a positive influence on the Software Analyzability.

To answer RQ2, the next three pairs of hypotheses were proposed and tested.

$H_0$4: TDD has no influence on the Software Changeability.

H_A4: TDD has a positive influence on the Software Changeability.

H_05: Refactoring has no influence on the Software Changeability.

H_A5: Refactoring has a positive influence on the Software Changeability.

H_06: CI has no influence on the Software Changeability.

H_A6: CI has a positive influence on the Software Changeability.

To answer RQ3, three pairs of hypotheses were proposed and tested.

H_07: TDD has no influence on the Software Stability.

H_A7: TDD has a positive influence on the Software Stability.

H_08: Refactoring has no influence on the Software Stability.

H_A8: Refactoring has a positive influence on the Software Stability.

H_09: CI has no influence on the Software Stability.

H_A9: CI has a positive influence on the Software Stability.

To answer RQ4, three pairs of hypotheses were proposed and tested.

H_010: TDD has no influence on the Software Testability.

H_A10: TDD has a positive influence on the Software Testability.

H_011: Refactoring has no influence on the Software Testability.

H_A11: Refactoring has a positive influence on the Software Testability.

H_012: CI has no influence on the Software Testability.

H_A12: CI has a positive influence on the Software Testability.

Lastly, to answer RQ5, three pairs of hypotheses were proposed and tested for the resultant Maintainability that was derived from sum of weighted SA, SC, SS, and ST.

$H_0$13: TDD has no influence on the resultant Software Maintainability.

H_A13: TDD has a positive influence on the resultant Software Maintainability.

$H_0$14: Refactoring has no influence on the resultant Software Maintainability.

H_A14: Refactoring has a positive influence on the resultant Software Maintainability.

$H_0$15: CI has no influence on the on the resultant Software Maintainability.

H_A15: CI has a positive influence on the resultant Software Maintainability.

Variables of Study

Dependent Variables

Besides software maintainability as a main dependent variable, the dependent subvariables to measure software maintainability are SA, SC, SS, and software testability ST.

Table 1

Software Maintainability Subcharacteristics per ISO/IEC 9000/9126 Software Quality Standard

Y	Maintainability subvariables	Variable Abbreviation
Y1	Software Analyzability	SA
Y2	Software Changeability	SC
Y3	Software Stability	SS
Y4	Software Testability	ST

The four dependent subvariables were generated primarily from software maintainability related literature including ISO/IEC 9126-1 (2003) software quality standard and other researchers' studies (Chen & Huang, 2008; Hegedu et al., 2010; Kanellopoulos et al., 2008; Kanellopoulos et al., 2010) that integrated software maintainability specific variables in their studies. I used the following independent variables to examine the impact on the dependent variables in this study.

Independent Variables

The main independent variable that drives software maintainability in this study is the ASDM. Three key variables that operationalized the ASDM model or main independent variable in this paper were identified as TDD, REFR, and CI.

Table 2

List of Independent Variables that Operationalize Agile Software Development Model (ASDM)

	ASDM variables	Variable Abbreviation
X1	Test Driven Development	TDD
X2	Refactoring	REFR
X3	Continuous Integration	CI

The pair-programming variable, although significant in Agile projects, was not included in the study due to its stable usage within the examined software development project. Furthermore, all the data were extracted from an iteratively developed software

system that followed all key Agile model-driven XP practices listed in the Agile literature review.

TDD (X1): Test-driven development is one of the key characteristics of ASDM approach in which unit tests are written before writing the actual code (Beck, 2003). The creation of the test and coding follows testing until they pass. The gradual test-code cycle is followed until the user functionality is delivered that may span one or more iteration. The developers thus create a good amount of test code that can be measured, indicating whether ASDM followed TDD during the development. This study measured the TDD in the percent of test classes created in every Agile iteration during the ASDM driven development project.

Refactoring (X2): ASDM is also characterized by periodic refactoring of the code throughout the iterations. The intent of refactoring is to drive the simplicity in the design and make the code more manageable in an incremental manner. Developers take steps to improve the software program structure without actually changing its external behavior (Fowler et al., 1999) during the development iterations; however, it may not take priority over the user requirement-specific programming tasks and efforts. The consequences of not doing refactoring—and, specifically, its impact on overall software complexity—are well recognized in the industry and management. Refactoring efforts are evident when programmers spend their efforts simplifying large and complex classes or methods during the iteration resulting in reduction of the count of complex classes. McCabe Cyclomatic

Complexity (McCabe, 1976) was used in this study to measure the complexity of the software classes at the end of every Agile iteration.

Continuous Integration (X3): With incremental code development, an Agile software development team essentially introduces new bugs in the code. Each iteration continuously changes the software code. Continuous integration technique is a unique practice that allows the Agile software development team to build and integrate the software code on an ad-hoc and continuous basis whenever the changes are incurred to the software during the development phase itself. This practice enforces the quality of the code and reveals the issues resulting from continuous development. The count of integrated builds was used in this study to measure the extent of CI being used during every iteration. This study conducted an analysis based on the existing empirical data, that is, the software systems developed using XP or Scrum, which are the most widely accepted Agile models today (Charette, 2004) in the real field setting of an IT organization within a technology manufacturing corporation.

Purpose of the Study

The purpose of this investigation was to draw attention to the Agile development model's influence on software maintainability characterized by its analyzability, changeability, stability, testability within software development project while comprehending the long-term impact on software life cycle. The intent was not to declare current Agile methodologies within IT companies as inadequate or derisory when assessing the resulting software maintainability as a key long-term software quality

attribute. Nor was it to declare Agile development methodologies as inefficient for their intended software development goals. The intent was to provide quantifiable insight into how software maintainability is influenced by the Agile development model as a result of software development project within IT organizations.

Software maintenance researchers consider software maintenance cost, both direct and indirect, as a leakage to business organization's profitability (Kunstar & Havlice, 2008). The purpose of this study was to gain insight to build upon prior research, specifically Moser et al. (2007), Chen and Huang (2009), Kanellopoulos et al. (2010), and Sindhgatta et al. (2010), which were guided by software evolution and maintenance laws originally postulated by Lehman and Belady (1974). These scholars have analyzed software maintainability within the software life cycle and software maintainability as a function of software development approach. In extending these key researchers' efforts, this study provided a much-needed acuity by focusing on the ASDM and a crucial attribute of software quality–software maintainability.

It was hoped that this research would connotatively stimulate the significance in the increasingly important, but mostly debatable, undervalued, and overlooked maintainability side of the software evolution life cycle driven by the Agile model. Understanding software maintainability-specific dynamics within software systems approached with ASDM did contribute to sharpen the Agile approach for enhancing maintainability and yield a cost-effective software maintenance life cycle.

Theoretical Framework

The research questions in this study were answered mainly within the tenets of laws of software evolution originally developed by Lehman and Belady (1974) and their revisions over the last 30 years (Lehman, 1991, 1996), with supportive perspectives from open system theory and resource dependency theory (Pfeffer & Salancik, 1978). These supportive theories provide the fulcrum to multiple stakeholders including IT management, software development, and maintenance resources to design, deploy, and sustain the software efficaciously that participate at various stages of the software life cycle.

The research question in the current study enlisted software maintainability-related four subcharacteristics or factors as dependent variables and three Agile model attributes as independent variables, as highlighted in Figure.1. Note that resultant software maintainability or indexed Y was regressed for X1 (TDD), X2 (Refactoring), and X3 (Continuous Integration) at the end, besides regressing all separate Ys on all X variables. The indexed Y is essentially a sum of weighted analyzability, weighted changeability, weighted stability, and weighted testability. This is because software maintainability is a function of its analyzability, changeability, stability, and testability. Agile development teams, IT, and system maintenance management will benefit from this integrative assessment approach to comprehend the impact of the Agile approach on individual attributes of maintainability as well as on cumulative resultant maintainability.

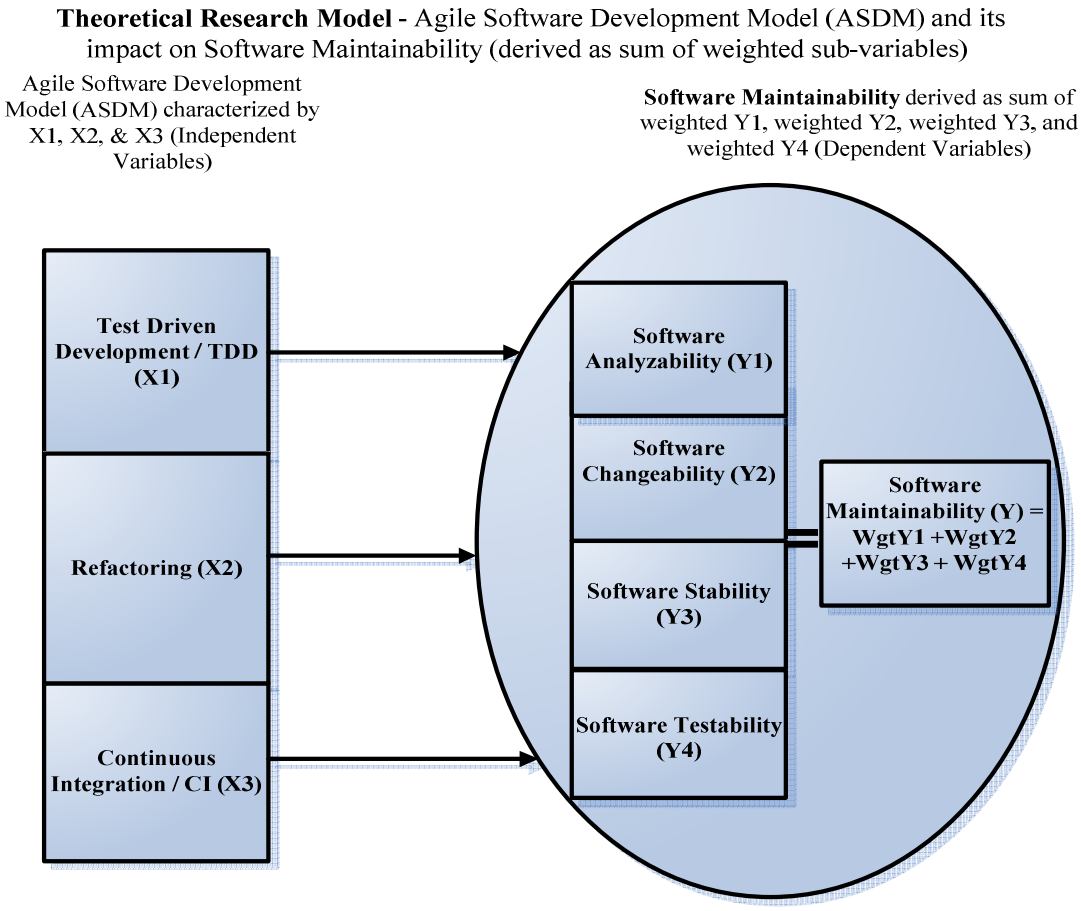


Figure 1. The conceptual framework – Agile model and its impact on software maintainability and its sub-characteristics

Definition of Terms

This section highlights the key terms that are used in this study with added information and clarity.

Adaptive maintenance: Changes made to the software system to evolve its functionality to changing business needs or technologies (Kunstar and Havlice, 2008).

Agile Software Development Model: The Agile model is a unique software development approach that is adaptive and agile in nature, characterized by iterative and nonlinear development pattern. The Agile model follows blend of development practices essentially derived from Agile manifesto throughout the development cycle as tabulated in appendix A.

Analytic Hierarchy Process: Analytic Hierarchy Process is a multicriteria decision-making process originally proposed by Saaty (1980).

Analyzability: Software capability to allow identification for parts, which should be modified. ISO 9126 defined it as attributes of software that bear on the effort needed for diagnosis of deficiencies or causes of failures, or for identification of parts to be modified.

Corrective maintenance: Changes made to software system to repair flaws in its design, coding, or implementation (Kunstar and Havlice, 2008).

Complexity: Structural characteristics of software that represent how the objects within programs are interrelated within object oriented programming (McCabe & Watson, 2994).

Coupling: This characteristic of software represents physical connections on how tightly the objects are related within object oriented programming (Chidamber, Darcy, Kemerer (1998).

Duplication: It is the percent of all code that repeats more than once (in equal block of at least 6 lines; Heitlegar et al., 2007).

E-Type system: Software that is proprietary and not open standard and is widely used in the real world.

Information Technology Infrastructure Library: Information Technology Infrastructure Library is an IT management framework or approach to IT service management offering the set of best practices (ITIL, 2002).

Iteration: Repetitive and small cycle of development within an Agile development project (Beck, 2000).

Object Oriented (OO): Software development paradigm or approach that refers to an 'object' as a main entity within the program structure.

Perfective maintenance: Changes made to the software system to add new features or to improve performance (Kunstar and Havlice, 2008).

Preventive maintenance: Changes made to the software system to avoid possible future problems (Kunstar and Havlice, 2008).

Refactoring: It is a development practice used within the Agile development project to restructure the code without changing its external behavior (Fowler et al., 2002). Refactoring is a technique that essentially simplifies the software code through several restructuring techniques throughout the development iteration or sprint cycle.

Software maintainability: The ease with which a software system or component can be understood, modified to correct faults, improve performance or other attributes, or adapt to a changed environment (IEEE 1990).

Stability: Software capability of the software product to avoid unexpected effects from modifications of the software. ISO 9126 defined it as attributes of software that bear on the risk of unexpected effect of modifications.

Testability: Software capability of the software product to enable modified software to be validated. ISO 9126 defined it as attributes of software that bear on the effort needed for validating the modified software.

Waterfall: Software development model that is a sequential development approach with linear phases–mainly analysis, design, and implementation.

XP: Extreme Programming: Agile development technique that is one of the leading Agile software development practices.

Assumptions

The assumptions in this study were driven by the Lehman's (1997) software evolution theory and specifically by underlying laws of continuing change, increasing complexity, self-regulation, conservation of familiarity, and law of declining quality. The continuing system change theory asserts that the software will and needs to continue to adapt and change throughout its life cycle to retain its usability. Agility in business requirements (Hirzalla, 2010), integration of new technology and tools within existing software system (Ruparelia, 2010), and continued corrections to stay functional and operational strengthen and support this assumption. The theory of declining quality formulated in 1996 by Lehman and Belady essentially complements the continuing change theory and asserts that the software will decline in its quality if it fails to adapt

and integrate changes (adaptive, corrective, perfective, preventive), therefore leading to poor maintainability over time. The other assumptions of this study are related to the increasing complexity, which states that software complexity grows with its evolution unless the work is done to reduce or maintain it, with eventual slow growth of a software system (Grubb & Takang, 2003).

Additionally, this study also assumed that the evaluators are experienced Agile practitioners and are professional in their expert judgment when using AHP to elicit the data from their work experience with one or more Agile-driven software development and maintenance projects.

Limitations

The limitations in this study were that it was conducted based on existing data–specifically, the software system that was developed using ASDM within an IT organization of a U.S.-based semiconductor manufacturing corporation. The study examined a software system that was developed using XP and Scrum methodologies. Therefore, the results obtained may not be generalized among non-Agile software development projects. The other limitation of the study is that it focused specifically on the software maintainability characteristics tree and its four specific subcharacteristics, and it did not address impact on other software quality characteristics such as portability or reliability. Additionally, events such as IT software development organizational changes, development resource turnover, and overall IT organizational readiness to Agile and software support functions are not normalized and not incorporated into the study.

Finally, the data used was based on an XP-driven software development project that used an object oriented paradigm, so the results may not be directly applied to other development approaches or model variants that are being practiced within IT organizations. This study's focus was limited to studying four maintainability-specific internal subcharacteristics as shown in Table 3, and no other software quality characteristics to maintain clear focus surrounding the research questions.

Table 3

Internal Software Quality Characteristics Examined in this Study

	Internal Quality Characteristics	
	Software	**Software**
Software Portability	**Maintainability**	**Reliability**
Adaptability	**Analyzability**	Maturity
Installability	**Changeability**	Fault Tolerance
Co-existence	**Stability**	Recoverability
Replacability	**Testability**	

Note. Bold items represent internal software quality characteristics examined in this study.

Software development projects are expensive and demand the business deliverable to be accomplished within allocated resources, including budget and acceptable time. With the given project constraints and valuable organizational resource allocation, IT development projects that are specifically driven to assess the impact and

overall efficacy of a development methodology such as Agile driven methods (XP, Scrum) are practically difficult to experimentally study within operationally active IT organizations.

The high cost associated for the experimental setup to assess the impact of identified Agile-specific factors on software maintainability and application support functions, forces the research design to be based on post facto or already existing data. Prolonged engagement of the expensive and busy resources for a longer time may not be practical within the business organization for possible experimental research setting as highlighted earlier.

Scope and Delimitations

The scope was to establish an empirical link between the Agile-driven software development model adopted in the development stage and the software maintainability attributes of the developed software. Despite the fact that the study addressed tenets of the Agile software development model, its functional characteristics, and how the Agile model impacted the increase or decrease in software maintainability, this study did not attempt to develop a causal model between Agile attributes and maintainability-related attributes. This study was an extension of existing Agile model and software maintainability-related studies and how this model is a development approach in specific impacted software maintainability and its internal attributes or subcharacteristics.

Another delimitation of the study was that the analysis did not attempt to highlight the implications on software maintainability within any specific work locations,

organizations, geographies, or countries. Furthermore, this study did not attempt to show the merits or demerits of one Agile model attribute over another. Researchers reported that the software language and technology selection (Briand et al., 1997), software maintenance processes (Grubb & Takang, 2003), software age (Banker,1992; Oman, 1992), development team stability, experience, and skills (Pigoski,1997) are some of the factors that influence the maintainability. This study, however, did not link these intervening variables that fall outside of the core Agile development approach model to software maintainability. It is important to realize that software development and or improvement related to the project's budgetary concerns could force organizations to retain software with higher maintainability despite realizing the need to improve the software maintainability. In some cases, a software maintainability improvement project may even not get prioritized for several reasons within the IT program and project management domains. This study did not analyze the possibility of losing the customer base due to poor satisfaction or inadequate usage of the software systems due to high maintainability nor forecast maintainability. The study didn't assessed actual financial implications to IT management due to possible increases in software maintainability.

Finally, this study did not track the software source code over the entire software life cycle, but only assessed maintainability for a specific set of revisions during the Agile development project based on the availability of data and specific selection criteria outlined in Chapter 3. This delimitation arose from the nature of the data collected from

the specific IT organization within the identified corporation at specific times during the software life cycle, as bound by the quasi-experimental research design of this study.

Significance of Study

The Agile-driven software development phase is a high-leverage point in the software life cycle in terms of delivering intended business value early. The knowledge about software maintainability within the Agile driven software development context will contribute to better software application design, qualitative integration, and better software maintainability during the postdeployment cycle. For instance, software stability within the ASDM context is important for IT management as it also indicates the level of understanding of the development team about the software design, domain, and possible efforts that may be needed later in the Agile development iterations (Olague, 2006). Unstably designed software could provide needed cues to the Agile project management and development team to realign the focus on the development team's adherence to one or more Agile practices, including the possible alteration in the development approach or process itself.

This study was significant given that Agile is touted as the preferred software development model of today's dynamic business world and IT organizations but with minimum understanding of its implications on software maintainability. This study benefited Agile practitioners and software development organizations integrating the Agile development model within their development ecosystem to coalesce Agile practice areas with software maintainability objectives proactively. The findings added value to

software maintenance organizations by contributing specific changes required within software maintenance domain supportive to the Agile-driven software development model targeting proactive control and reduction of software maintenance cost over software life cycle. Lastly, the IT service delivery and supportability risks may be mitigated with software maintainability embedded in the Agile development model.

In earlier studies, Lee (1998) and Balci (2003) suggested that controlled design and implementation considerations during the development phase could influence software maintenance efforts and therefore the cost involved in postdevelopment phase–often known as the *sustaining and support cycle*. Most of the studies, however, are from the pre-Agile development model era, which further necessitates reexamining the underlying concerns within the Agile development model's context that accentuates the software maintenance puzzle. Minsky (1995) argued that this increase in software maintenance has a negative impact on IT efficiency and business value attainment given tight IT budgets. Software organizations often operate in a highly dynamic market environment under tight time and cost constraints (Cugola & Ghezzi, 1998).

The types of software maintenance, adaptive and perfective software maintenance, represent about 75% of maintenance cost, whereas 21% of the cost goes toward corrective maintenance (Bennett & Rajlich, 2000). Typically 70% of software development budgets, according to Bennett (1990), are spent on software maintenance, and total maintenance constitutes anywhere between 40% and 90% of the total life cycle costs. Furthermore, these higher maintenance costs, including ineffective resource

allocation, are one of the major detriments to the productive IT development projects. An earlier study conducted by Carver (1988) examined the levels of complexity changes that occur during the software development phase and reported the finding that an increase in software complexity does impact the maintenance difficulty.

As a necessary outcome, this study provided a practical perspective for software engineering scholars, Agile practitioners, software maintenance organizations, and IT management to align Agile practices with maintainability considerations in the design and development phases. This study's advocacy of assessing software development model's implications on software maintainability in turn could benefit the business and IT organizations deploying the software applications in multifold ways to improve quality and control maintenance cost. Increasing reliance on software applications at all levels of organizations, societies, individuals, and economies further necessitated the critical assessment of software evolution within ASDM-driven software life cycle. Studying the impact of this modern development approach on software maintainability was the very first step toward this goal.

The results of this study could help IT organizations make better decisions about the optimal alignment of the Agile model and attain better software maintainability. The results could also be further used to leverage the Agile model within the cloud and virtual environment deployment, which is important to IT management and the Agile advocates for whom software maintainability is one of the key success indicators and considerations when attaining Agility through ASDM.

Need for the Study

When debating the long-term life cycle impact of Agile methodologies, Majko-Mattson et al. (2006) led intriguing discussions among Agile practitioners as well as researchers. Agile software development methodologies integrate several key practices that are intended to drive quality software development. Agile practices such as pair programming and continuous integration assist the code quality improvement (Lindvall et al., 2004; Schwaber & Beedle, 2002). Furthermore, a direct result of this software quality improvement should also be evident in the reduction in the software defects (Auvinen et al, 2005; Ileva et al.,2004; Laymann et al, 2004; ; Lindvall et al. 2004; Schatz & Abdelshafi, 2005). Researchers (Lawrence & Yslas, 2006; Schatz & Abdelshafi, 2005) also argued that the Agile model adoption itself is a challenge in large (Larrson, 2003), complex, and distributed (Cohn & Ford, 2003; Mistra et al., 2006) projects and organizational settings.

These mixed findings necessitated the need for the further evaluation of the impact of the Agile approach on software quality within bigger project as well as within large IT organization. Formal code reviews (Auvinen et al., 2005), survey-based studies (Benefield, 2008; Rico, 2008; Vilkki, 2009) analyzing quality perception within organizations that adopted ASDM, and number of defects analysis (Ileva et al. 2004; Schatz & Abdelshafi, 2005) were conducted recently since Agile adoption. However, there was no empirical study examining the implications or impact of the Agile model on

software maintainability and its subcharacteristics, the core internal software quality attributes. Few studies that had attempted to connect Agile software development methodologies to software quality tenets, and no quantitative study exists that specifically examines the impact of ASDM on the four key software maintainability characteristics. As an example, researchers (Hulse et al., 1999; Jain et al., 2008; Madison, 2010) argued that software architectural and design considerations early in the development stage could potentially improve maintainability. Standard and simplified design approach reduces the software complexity and, in turn, software analyzability—one of the key subcharacteristics that determine software maintainability (Jensen et al., 2008).

In a correlation study, Misra et al. (2008) also studied Agile adoption-related factors and their relationship to software quality issues and the project's success. Researchers, including Sindhgatta et al. (2010), also studied software maintenance through defect analysis in the postsoftware deployment phase, but they did not study the impact of the Agile-driven software development approach on internal quality characteristics related to maintainability based on source code property assessment. This void further underscored the need for this study.

This study was further needed to comprehend the Agile model's impact on long-term software sustainability efforts due to the resulting maintainability from the strategic IT management perspective. The software development model plays a critical role in shaping and defining the software life cycle and its postdeployment behavior. Without knowing how a development model influences the software maintainability,

analyzability, changeability, testability, and its stability, IT management could potentially risk spending precious organizational resources on the wrong priorities.

Within this integrative theoretical framework, the implication of ASDM as an early business value delivery approach (*Agile Manifesto*, 2001) is significant to comprehend within software maintainability realm. Software maintenance researchers consider software maintenance cost, both direct and indirect, as a leakage to the business organization's profitability (Kunstar & Havlice, 2008). Furthermore, IT organizations may fail to reap the true benefits of potent characteristics of the Agile development model and deploy the software maintenance resources for more productive IT organizational needs. The Agile software development model is one of the important determinants within the software quality chain that impacts the software maintenance and associated service domains; it is imperative for IT management to drive a closer examination of the model's traits. The rise in software maintenance efforts and cost translates into low return on investment (ROI) for the business and organization it serves, which eventually impacts the satisfaction of internal and external stakeholders and customers. This study addressed the issues of software maintainability and lack of its adequate considerations during the development, as posited in several Agile and software maintenance literature, specifically when organizations are driven by the increasing software quality expectations and the strong need to deliver software early.

Social Change Implications

The implications of this study are rooted to its ability to assess the impact that explains the Agile development model's influence on software maintainability. The link between this software development approach, referred in this study as *ASDM,* and software maintainability characteristics could potentially help IT management as well as Agile practitioners to comprehend how the Agile model may be leveraged and aligned when considering software maintainability implications in the postdeployment phase. The study also has implications for reduction in software maintenance costs and therefore better productivity in a softer application management function. Being able to understand the software maintenance through the concepts of software evolution driven by ASDM could help mitigate the software quality concerns, thus preventing destabilization of the software application as well as underlying business functions that rely upon it. IT cost controls through better software maintainability, and project success rate improvement through delivery of quality software systems with a healthy life cycle, are socially significant implications of this study.

The implications of this study for IT software development and maintenance management relate to their ability to understand software life cycle specifically initiated with Agile driven development approach and its impact on software maintainability. This knowledge now could make it much easier for Agile project management and Agile practitioners to adjust, enforce, and influence the specific Agile practices and techniques. It will also allow project management to incorporate and adjust for risk within the

software application development, deployment, and postdeployment phases for better software maintainability. At the same time, understanding the impact of the Agile model on software maintainability could help IT software application and maintenance management align maintenance resources and functions targeted specifically to Agile driven developed software systems. The IT software development and maintenance management could drive Agile practices deeper into the software life cycle with greater emphasis put on continuous delivery and automated deployments, which are getting significant attention within the cloud computing and virtual environments while managing software maintainability.

Chapter Summary and Organization of the Study

This chapter has highlighted the importance of understanding software maintainability as an essential quality outcome of the Agile driven software development approach while controlling for variables such as Agile project size as measured by a number of agile developers, Agile model implementation period measured in number of years, and software development paradigm filtered to the object-oriented approach. Although this study was quasi-experimental in nature and assessed existing data, controlling for the above variables during the software system selection step was important. As such, the control variables in this study facilitated the filtering of the data.

The impetus behind this study was the wider attention that the Agile software development methodologies has received to improve business agility by IT management. Organizations are extracting the business value early in the software development project

through adaption of frequent changes within the software, but comprehending the dynamics between maintainability characteristics of software and ASDM is becoming an equally propelling need as the attention continues to grow.

This chapter also provided the theoretical framework used to analyze and determine the impact to software maintainability. The analysis conducted in this study approached maintainability from a perspective similar to prior software maintenance function studies, though this study focused on the measurement of a key software quality attribute: software maintainability during the post-Agile development phase. The organization of this study logically followed the conceptual background and problem statements highlighted in this chapter.

The study proceeds as follows: Chapter 2 presents a review of relevant literature on Agile software development approach, software maintainability, underlying software evolution and maintenance theory, and key attributes of both these domains. Prior software maintainability and ASDM-approach related studies are discussed and their tenets compared and contrasted within the context of research question of this study.

Chapter 3 addresses the quantitative method used in this study, which included the research design and the data set used to analyze the implications of the Agile model on maintainability. The additional two chapters were developed after conducting research data collection and analysis. In Chapter 4, the data and empirical analysis results are presented, with the aim of answering the research questions posed in Chapter 1. Chapter 4 also includes statistical data that led to the rejection of the null hypotheses

put forth in the study. Finally, Chapter 5 summarizes the conclusions that can be derived from this study. This section contained a discussion of the study's limitations and implications for positive social change. The chapter also offers recommendations for IT management, Agile practitioners, and future research.

Chapter 2: Literature Review

The research reviewed in this study utilized the tenets of software evolution theory to explain the Agile software development approach and its continued gains over the other traditional development methodologies over the last decade within the context of software maintainability. Additionally, this review also focused on software maintainability–a critical success factor within the software development dynamics that conterminously assemble within the Agile driven development approach. In this literature review, the first step involved providing an overview of how the software development model literature has been researched with the emerging need of ASDM in recent years. Second, a review of literature that merged the tenets of ASDM within the context of Lehman's software evolution laws is presented with identification of key ASDM factors or characteristics. Next, a discussion follows on recent research studies pertaining to software maintainability and its subfactors, that is, analyzability, changeability, stability, and testability, focusing on its implicit and explicit links to ASDM.

Research Strategy

In conducting this literature review, five sources and databases were used: Advanced Computing Machines (ACM) Journals, IEEE Journals, IEEE conferences, Software Maintenance and Evolution Journal, ABI/INFORM Global, Dissertations & Theses at Walden University, Dissertations & Theses: Full Text, ProQuest Central, and Gartner Research Database. The initial selection of articles about the tenets of software

development methodologies, ASDM, software maintenance, software maintainability, and software maintenance evolution in specific is 5 or more years old. More studies that are contemporary related to software development methodologies (less than 5-years-old) were sought to provide the updated thinking and analysis based on Agile principles. The majority of studies used in this review are from 2005 to 2011. As Agile practitioners are also heavily disseminating though conferences, the literature review also spans many conference proceedings and practitioners' papers presented at these professional conferences, mostly retrieved from IEEE database.

The search terms were *software maintainability, software maintenance, Agile development methodologies, software change, Agile development model, software development, software maintenance cost, software agility, business agility, software life cycle, software changeability, software complexity, software testability, software analyzability, software stability, software evolution, IEEE definition for software maintenance, project success, software changes, critical software development success factors,* and *ISO/IEC 9126 2004.* The research also included the keywords *software maintainability*, *software quality*, and *Agile* in the following combinations: *quality in agile software development projects, software maintainability within agile, agile software development life cycle*. Some of the results were actual research studies whereas others were descriptive articles, case studies, and some were meta-analysis. One dissertation related to Agile project success factors was also located, but not included in the literature review to greater depth as it did not assess software maintainability as a success factor in

its variable list. Another dissertation related to software metrics and maintainability was included for its relevancy to software maintainability measures as well as relevant data analysis method used in this study.

This chapter followed the literature map as shown in Figure 3, and it guided the organization of relevant sections beginning with ASDM as an independent variable followed by four dependent variables specific to Software Maintainability, namely, Software Analyzability, Software Changeability, Software Stability, and Software Testability. Connectively, Lehman and Belady's (1978) Software evolution theory examined these two distinct yet related domains. The literature review took an integrative approach throughout this chapter.

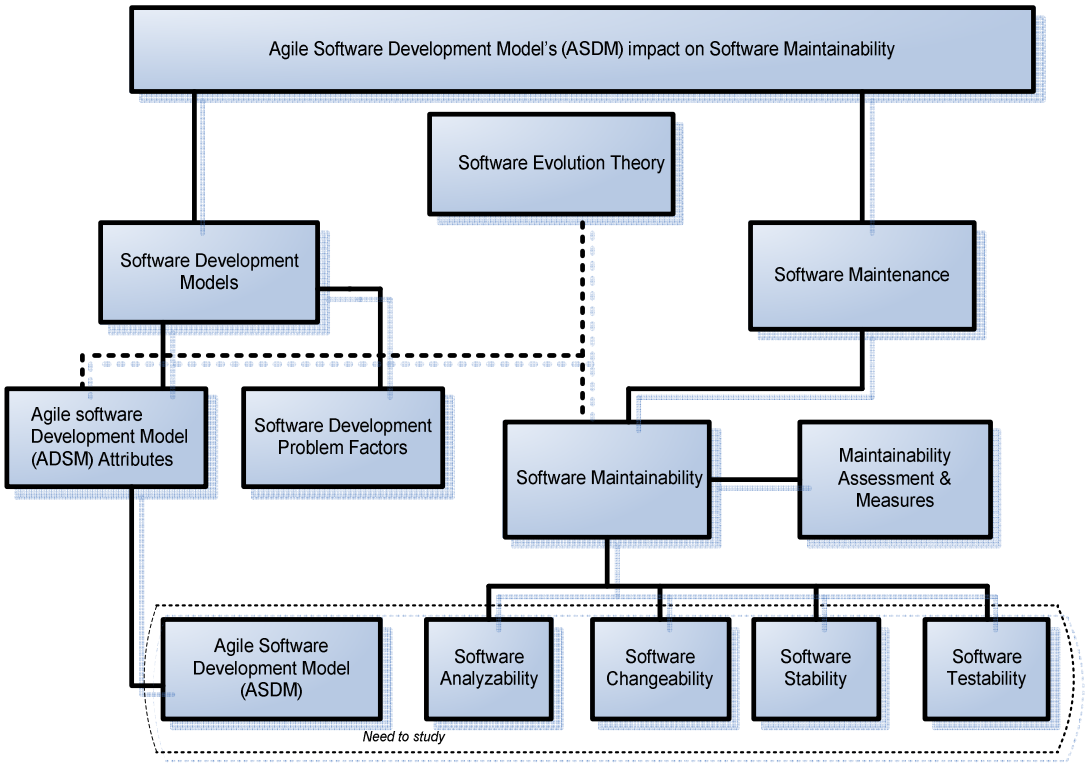


Figure 3. Literature map organization overview.

Agile Approach to Software Development–Maintainability Perspectives

The software life cycle begins with a specific development paradigm. Its multifaceted outcomes and impact on the interconnected entities, processes, and life cycle phases within software engineering domain are being studied widely within academia and research communities. Modern software development models such as Agile are not an exception. DeMarco (1978) contended that all the modern analysis methodologies are linked to improved system structure and importance of early development stages. These modern practices also enhance detail definition of systems functions, sequenced tasks and well defined intermediate results, and improved communication between users and analysts. Furthermore, all these aspects are consequently expected to improve the reliability and maintainability of a software system reducing the "burden of maintenance" (Dekleva, 1992, p. 355).

In an exploratory study conducted by Dekleva (1992), the focus was targeted to comprehend the influence of a few selected system development methodologies on maintenance time. The study examined four types of development approaches. These approaches included software engineering referred to as a *process-oriented methodology* (Martin & McClure, 1985), information engineering approach that is data oriented, prototyping referred as an *iterative approach*, and CASE engineering examined as automation oriented approach. Practically one or more combinations of these approaches are used during the system development, with each approach still unique in its own set of characteristics when compared to traditional SDLC approach. The major limitation of

this study was that it was the first study to investigate the relationship between the development methodology and maintenance when the modern systems development methodologies were just being introduced, but with limited usage and proliferation within industries in the 1990s.

Dekleva's (1992) study yielded no evidence of correlation between the usage of modern software development methodologies and the maintenance time and cost involved. The findings, however, showed that the changes in the time spent on maintenance activities with more reliable software resulted from the modern development methodologies, which needed less frequent repairs. The system size (complexity) was linked to higher maintenance cost, thus software developers could potentially offset the benefits attained due to greater system functionalities (higher complexity) by spending more on the maintenance and loss in productivity. Similarly, the number of users also influenced the maintenance time, suggesting that developers need to be considerate about the system with higher system usage base. System age and organizational stability also influenced the maintenance time.

Modern development methodologies apparently did not account for the reduction in total maintenance time (Dekleva, 1992). Furthermore, it was found in Dekleva's study that maintenance time actually increased after the system's deployment in the early years of sustaining. The methods, however, led to changes in maintenance time allocation with less time spent on emergency fixes, change evaluation requests, and implementation of mandatory changes including functional enhancements. Li et al. (2010) cautioned that

cost drivers of software maintenance effort might vary across organizations and development projects.

Changes within and outside of a software system's ecosystem are inevitable. Ravichandar et al. (2008) examined the ripple effect of needs change and found that capabilities-based design improved the change-tolerance of the system with volatile needs. In another study, Huo et al. (2004) attributed the acceptance of the Agile approach for its accommodation to dynamic requirements, close collaboration between developers and customers, and resulting early software delivery. Note that throughout the software life cycle stages, requirement gaps and related issues (Apfelbaum & Doyle, 1997; Dethomas & Anthony, 1987; Mogyorodi, 2001; Monkevich, 1999) were significant factors contributing to defects, therefore influencing the software maintenance. When compared to the waterfall model, Sommerville (2000) criticized the "ceremonious" procedural traits and inflexible approach in addressing changing requirements despite the model's success in small and large size development projects (p. 65). Today, Agile is seen as largely successful in smaller projects, but its ability to scale effectively within larger projects is still questionable. For instance, system metaphor practice within ASDM is equivalently used instead of formal architecture. Rosso (2006) contended that architecture designing and software development are simply the earlier activities within the software life cycle and that software evolution is inevitable given that business needs and hardware keeps changing.

Software maintainability related subcharacteristics; namely, analyzability, changeability, stability, and testability are benefited directly from qualitative software design and sound architectural thinking. Huo et al. (2004) mapped both these models as shown in Figure 4 and found that Agile methods include software quality-guarding practices within development stages as well as some practices in a supportive role.

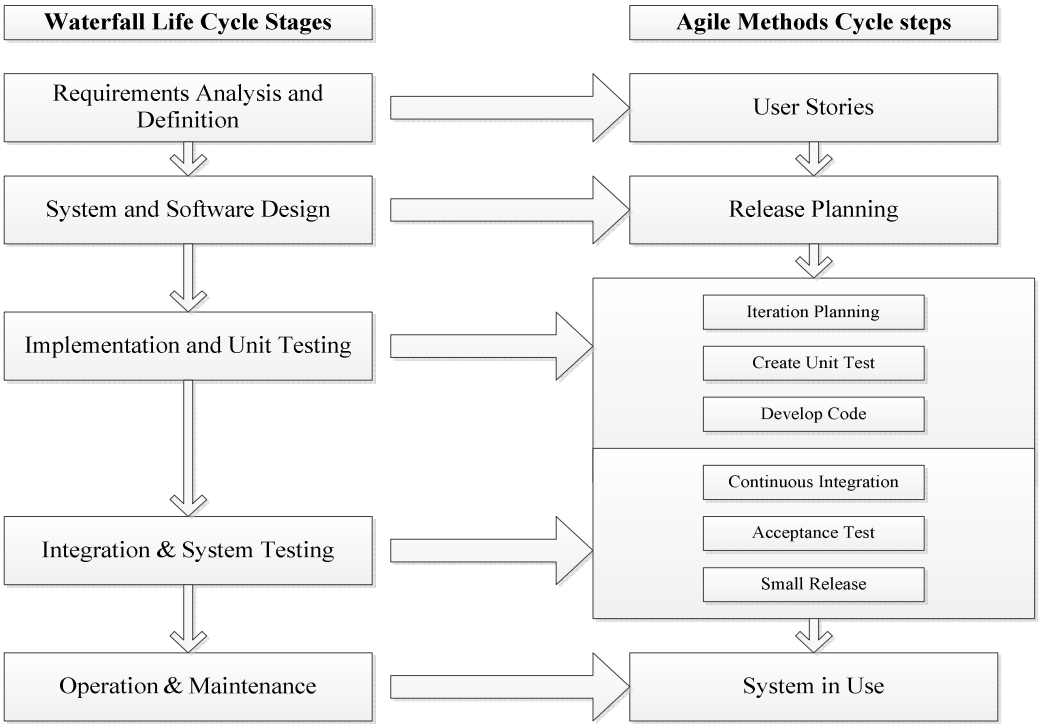


Figure.4. Comparison of Waterfall and Agile model-driven software life cycle stages (Huo et al., 2004).

Huo et al. (2004) also found that for frequency and availability of these quality assurance practices, such as continuous integration, the test-driven development is higher and early in the process stages. Software development projects are often expensive and demand well allocated resources, including adequate budget and time. With given project constraints and business value at stake, IT development projects specifically

driven to test the impact and overall efficacy of the Agile development methodology (such as XP and Scrum) in real-life settings are practically impossible.

Within iterative development cycles in Agile-driven projects, the development techniques or characteristics have very short feedback cycles (Beck, 2005). As an example, test-driven development (TDD) takes minutes to implement tests, leading to the code fulfilling that test. The continuous integration technique allows the programmers to change the code, recompile, and test its working in minutes on a regular basis throughout the day. The sprints in Scrum, or iteration in XP methodology, can be one to a few weeks, allowing for the working functionalities to be delivered ready for soliciting the feedback throughout this development cycle before moving to the next chunk of work. In the depiction presented in Figure 5, Ambler (2006) contended that ASDM practices allow the defect detection early compared to traditional development approach. As shown, the cost is lower when the frequent feedback cycles occur, or when the length of the feedback cycle is low. With the traditional development approach, for instance, the acceptance and system testing are open to the customers and testers only after months of development work. With longer feedback cycles, the changes potentially demand time and attribute to further delay in releasing the working software for its use. The cost to change the code based on early feedback during the development iterations is less as the code is still in its formation state. As the software code complexity grows with the development work progressing further, the accommodation of change, testing, and implementation of it is

still feasible within iterative Agile projects due to its characteristically change-supportive practices.

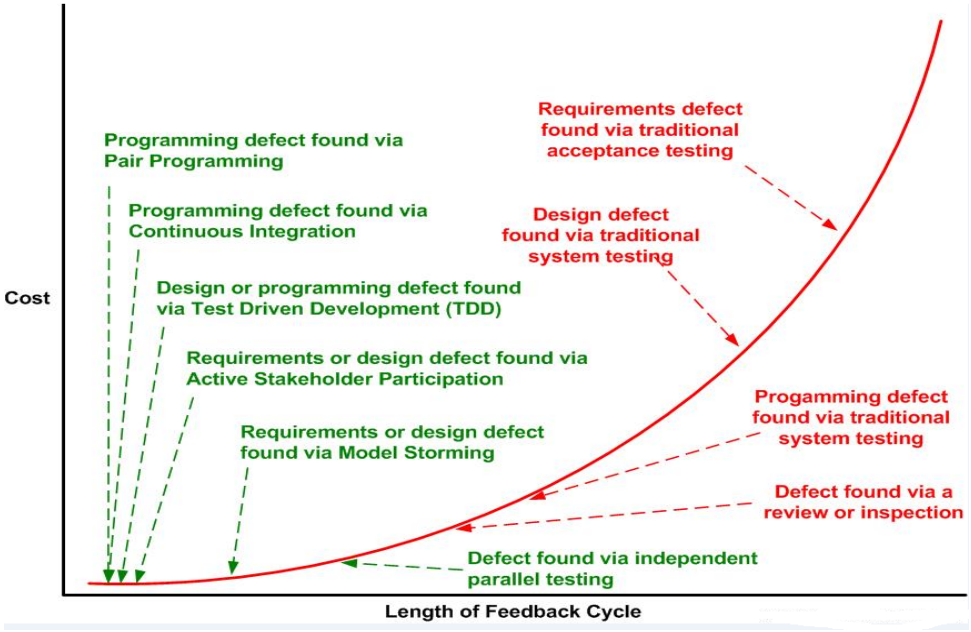


Figure 5: Cost of change curve (Ambler, 2006).

Pair-programming practice within Agile has an ability to detect and prevent the futuristic defect every minute when the developers develop the code daily throughout the iterations. As highlighted earlier, the continuous integration, TDD, and active customer participation throughout the iteration inhibits the defect buildup well ahead of the code deployment (Ambler, 2009; Beck, 2005). The cost to fix the defect found later in the software life cycle may involve regrouping of technical resources to comprehend the change implications as well as time investment to actually implement the change. The cost of actual software defect in terms of the impact to the business is often high and may lead to loss of the revenue for the business relying on the software application for its

operational functions. The feedback cycle, from the paired programmer, testing tool, and active customers in the Agile-driven development, in summary, is small and often minutes to hours (Ambler, 2009). The cost of change within the Agile development paradigm is essentially flat as seen on the left side of the curve in Figure 5. As marked in red, the software defect gets detected way later due to the longer feedback cycle that is often in months before the customer even can see the working software and begin testing.

Pahl (2004) posited that software's ability to adapt to its requirements and environment throughout its life cycle is critically vital for core business in continually evolving environment. With the Agile development model's absorption in the mainstream (Augustine, 2010) in recent years, examining its implication on software maintainability, a key quality attribute of the software life cycle in the post-deployment phase, is critical for IT management to govern end-to-end software life cycle cost effectively.

Next, the Agile approach to software development and evolution is based on a specific set of methodical practices, and thus the maintainability perspective cannot be completely comprehended without the knowledge of the Agile manifesto. The upcoming section sheds light on the fundamentals of the Agile manifesto and its underlying principles followed by key practices that characterize ASDM.

Agile Manifesto and Model's Key Characteristics

Cockburn (2006) highlighted the Agile manifesto that is based on four core principles as:

1. Individuals and interactions over processes and tools

2. Working software over comprehensive documentation

3. Customer collaboration over contract negotiation

4. Responding to change over following a plan. (p. 284)

These agile principles when further demystified, can be elaborated into classes: self-efficacy (*individuals*), collaborative teams (*interactions*), self-managed and focused team approach (*working software*), cross-functional team alignment (*customer collaboration*), and responsiveness and flexibility (*responding to change*).

Most traditional methodologies significantly put stress on linear processes, planned activities, tools, and documentation whereas "agile approaches place eighty percent of their emphasis on ecosystems-people, personalities, collaboration, conversations, and relationships" (Highsmith, 2002, p. 40). These Agile principles form the required meshwork to uphold the independent variable: the ASDM of this study. The Agile model or ASDM exhibits the following key characteristics that distinguish it from any other legacy, as well as other modern development models.

Some of the Agile practices are interdependent and hence prove effective when implemented and practiced collectively (Beck, 2000). However, organizations may or may not integrate all the practices in their software development projects, and will therefore yield varied levels of success (Conboy & Fitzgerald, 2010). The details of these key Agile characteristics are tabulated in Appendix A, which enlists the key practices in

each area. These key practices are referenced in subsequent discussions related to software maintenance and maintainability.

Agile model is characterized by several unique practices, such as iterative and test-driven development and multiple, short, and continuous releases (Beck, 2000). On the other hand, software stability and business continuity are essential requirements of the software maintenance and operation phases (Khan, 2010). Besides software stability, several other aspects of software maintainability such as software analyzability, testability, and changeability in turn influence the maintenance function and efforts. In summary, through these practices, the Agile model collectively shapes the development team's actions, which in turn may influence software quality characteristics including maintainability.

Software Development to Maintenance

Software development is intellectually complex activity (Vessey & Glass, 1998) and demands expertise in the areas of problem solving and constructing software (system and software discipline). Software maintenance, although a distinct phase of the life cycle, involves similar functions, but prioritizes different objectives with one of the critical goals being to bring the software back to working or updated state. The specific artifacts produced during a life cycle are the results of the methodology definition, processes, and management philosophy. Importantly, these outcomes can be used to understand and evaluate the success or failure of a methodology, as well as its impact and

value for the organization (Chiang & Mookerjee, 2004; Green, Hevner, & Collins, 2005; Rai, Lang, & Welker, 2002).

Software maintenance, support, and development contend (Gibson & Senn, 1989) for the precious IT resources–money and programming resource time. Organizations often segregate expert programmers and maintenance programmers primarily between business critical, complex software, and relatively less critical or less complex applications. However, complex maintenance issues are practically handled by competent programmers in the line of escalation and could consume substantial time.

It is estimated that there are more than 100 billion lines of code in production in the world, and as much as 80% of it is unstructured, patched, and badly documented (Vliet, 2008). It is a colossal task to maintain the software systems to satisfactory operational levels with timely error correction within SLA for each reported error. At the same time, these systems must be adapted to inevitable changing environments such as updated infrastructure, new functionalities, and increasing user needs.

Agile practices are gaining increased attention within IT organizations. According to Knipp et al. (2010), "Agile concepts have reached the tipping point, with rising rates of adoption in almost every industry" (p. 20). Chiang and Mookerjee (2004) emphasized the improvement in the development process and its relationship with software maintenance. They argued that the benefits of process improvement are not limited to accelerating the development work, but also to reducing the effort spent on corrective activities. Software maintainability is one of the key success factors for

software development, and software development process need to influence this aspect as much as possible, according to Moser et al. (2007). They contended that XP technically supports and enhances internal attributes of software maintainability through simple design, continuous refactoring, integration, and TDD approach.

Within the ASDM context, Murphy, Duggan, and Norton (2009) of Gartner Research warned that Agile development done incorrectly may result in strong initial productivity, but could quickly add to an increase in software maintenance and support costs, if key Agile practices are not put in place. They further contended that the "real productivity comes into play by finding defects sooner, focusing on what users really need, and reducing the amount of rework" (Murphy et al., 2009, p. 210). In the defense of ASDM, Beck (2000) argued that XP-driven development projects integrate software maintenance into the subsequent iterations. All the iterations after the initial release essentially fall into the maintenance stage of the development cycle, according to Stafford (2003). Iterative development allows developers to validate the software throughout the development cycle instead of conducting the validation at the end. It is also noteworthy that, at times, the software maintenance may be deluged with periodic but constant streams of changes through short release cycles to production. Ferreira and Cohen (2008) reported the impact of the iterative attribute of development, continuous integration technique, TDD, feedback, and collective ownership on stakeholders' satisfaction with the Agile development process and development results. Iteration, continuous integration, and collective ownership were found to have the strongest impact

on this satisfaction measure. Note that Agile may be seen as an approach that disgorges the functionalities or requirements too often into the production environment and hence cause of possible service interruptions, as illustrated in Figure 6.

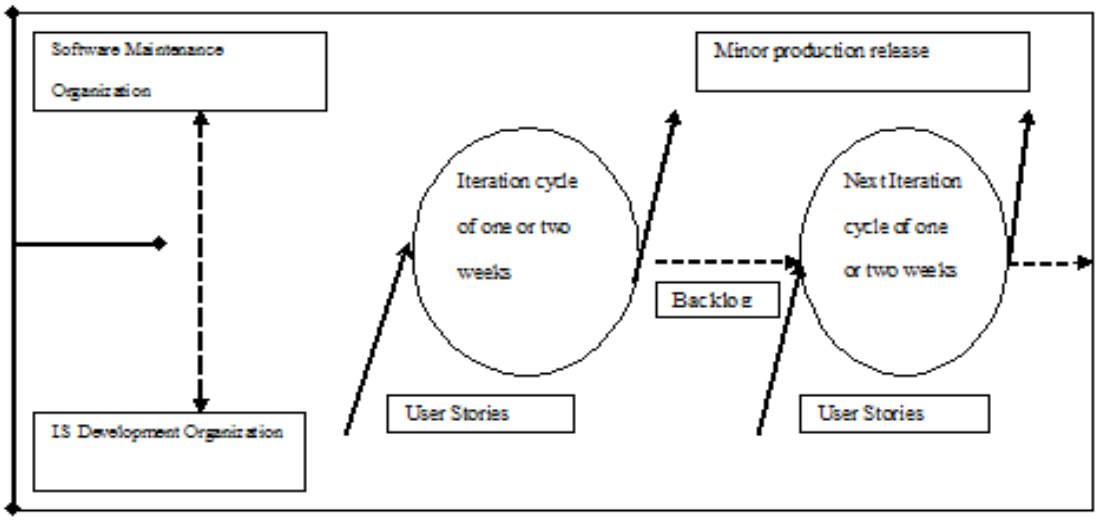


Figure 6: Operation/Maintenance and development interaction within iteration cycles.

Pair programming, one of the key Agile practices that is more prevalent in XP, yields more than just dissemination of knowledge across the team and error detection mechanism. It is an essential Agile trait that actively promotes continuous improvement in the current system as well as the futuristic application development efforts. Williams and Nosek (2000), in their study related to pair programming, found that this key Agile practice reduces the defect rate; furthermore, it does not impact the Agile development team's efficiency negatively. On the other hand, several researchers (Eberlein et al., 2002; Paetsch et al., 2003) proposed a basic form of requirements management within XP to address quality-related concern of the system.

TDD, another key Agile practice that rapidly vacillates between test case and actual code, enhances the ability of the development team to code incrementally, continually, and confidently without breaking the current state of the system. With every code addition and change backed by well written test or series of tests, software testability is also expected to be higher. Furthermore, TDD also serves as a foundation for refactoring, which is another key Agile practice that allows the Agile development to decouple the code and allow scalability of the application (Martin, 2003).

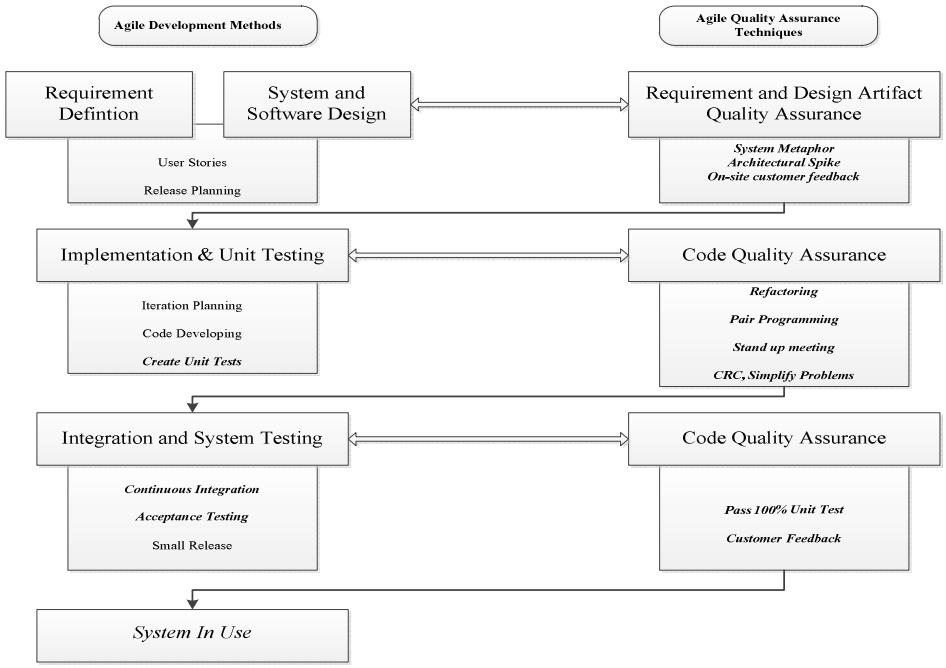


Figure 7. Agile method and software quality assurance practices (Huo et al., 2004).

Software refactoring is a restructuring technique to improve code quality (Fowler et al., 2002), especially extensibility, reusability, and maintainability without preserving its changing software code's external behaviors (Mens & Touwe, 2004). Refactoring practice from its inherent approach also appears to influence software stability and

changeability (Hegedus et al., 2010; Liu et al., 2007). Agile literature and practice suggested that TDD practice leads to an improved software design, and that, in turn, has a substantive influence on software maintainability, in specific software testability. Siniaalto and Abrahmsson (2007), in their comparative study, reported the impact of TDD on software design. They found that the impact of TDD on software design was not as obvious as expected; however, the test coverage was high in TDD adapted projects.

To strengthen the refactoring practice further within ASDM, additional Agile practice – the developer story was suggested by Jensen et al. (2008), and their field experiment's results showed that developer stories did affect architecture of the system as well as the developer's actual work and approach towards architectural issues. They further contended through explicated and visualized refactoring that developer stories increase the possibility of developing a practicable architecture through a series of intended options, through creating disciplined and recurring activities that: 1) facilitate sharing and embodying of knowledge about architectural issues, and 2) amplify visibility of refactoring efforts and this key practice for both customers and developers. (Jensen et al., 2008, p.183)

Simplicity in design practice ideally drives the iteration-specific user stories to completion status without adding unnecessary complexity in the design of the system. This agile practice takes an iterative approach and works only on a chunk of stories for applicable iteration. Continuous evolvement is thus at the core of this practice. The simple design objective also ensures that the code is less complex and easy to maintain.

The refactoring practice in Agile is a small and periodic transformation of the code that improves the system without altering its behavior in any way. Unit tests are run to ensure that the system continues to work without breaking after each refactoring cycle. Development teams do not need to wait either for iteration or release to complete, and refactoring can be done during iteration every hour or every day. With the only objectives being to improve the design, simplicity, and maintainability of the system code, refactoring is followed throughout the development phase (Martin, 2003).

Software maintainability, on the other hand, has been researched extensively in the past. Gibson and Senn (1989), in their experimental study, investigated the relationship between system structure and software maintainability and found that software structural differences impact the software performance. Additionally, improvement in the system structure decreases the time required to perform maintenance and reduces the frequency of ripple effect errors. Gison and Senn (1989) argued that maintainability of the system is evident in the time required to complete the change implementation (time), accuracy or correctness of the changes or modifications (accuracy), software programmer's confidence in the updates or changes made (confidence), and their own perceptions of the complexity of the software (perceptions). Maintainability is an attribute that is a combination of the software system and the maintenance team that performs the changes or maintains the software system (Hordijk & Wieringa, 2005).

The two critical software life cycle phases are distinctly clear, specifically when the software maintenance phase is consistently reported as being the most expensive, with 66% of the total software life cycle cost (Yip & Lam, 1994). Due to the cost of sustaining it, software maintenance is an expensive burden to businesses that is also referred as *technical debt* in modern, Agile development practices. Yip and Lam (1994) further contended that even structural improvements could add several advantages over the life cycle of actively maintained software system over complete redesign, which may not always be an option for IT management. Complex systems are difficult to comprehend and hence offer maintenance challenges, in addition to further corrective maintenance, in their life cycle (Gremillion, 1984; Lientz & Swanson, 1980; Vessey & Weber, 1983). In summary, system complexity is related to software maintainability.

For developers, according to Hotle and Norton (2009), Agile often is the most effective approach to maintenance and enhancement-specific software projects. Software maintainability and support specific considerations, however, are given short shrift when balancing the Agile principles within Agile-driven development project.

Software maintenance is defined as "the process of modifying a software system or underlying component after its delivery to fix faults, improves performance or other attributes, or adapt to a changed environment" (IEEE610, 1990, p. 92). The fundamental problem is that maintenance will continue to remain a big issue in the coming years. Because of the changes made to software, its structure degrades. Specific attention to preventive maintenance activities aimed at improving system structure is needed to fight

system entropy from time to time. Gibson and Senn (1989) further argued that the problem of maintenance is circular, i.e., software maintenance difficulty is due to its underlying complexity. It is because of this same complexity that more software maintenance is needed throughout the application life cycle.

A study conducted by Helms and Weiss (1985) found that a high amount of corrective and perfective maintenance was indicative of problems with the system development methodology. However, Agile development models were not matured fully and developed around this time frame; re-examining this assertion in the light of today's dynamic software development landscape is required to assess the implications of development approach on software maintainability.

In another study, Kan (2002) argued that the actual numbers of defects reported during the maintenance process are linked to the development process itself, and that fixing the defects with a quality fix at the earliest possible time is the priority action that can be taken during the maintenance phase. Additionally, reports estimated that regression testing could take as high as 80% of the overall testing budget and up to 50% of the cost of software maintenance (Harold, 2002). Kunstar and Havlice (2008) contended that the research in the future would aim to the improvement of maintenance, instead of trying to eliminate it given that the changes in the system and user requirements will be a constant part of the competitive and dynamic business organizations.

Within the development team, driven by Agile development methods, developers take on new quality improvement-related tasks, and testers get involved earlier in the project (Murphy, 2009). The cost of defect removal is lower than the cost of testing in maintenance phase after the software is deployed, according to Kan (2002). In summary, Software maintainability is not often a major consideration during software development and design phase (Bendifallah & Scacchi, 1987; Schneiderwind, 1987). A more controlled design and implementation process followed early in the software life cycle could reduce the software maintenance costs through improved software maintainability.

When studying key Agile specific characteristics, Mens and Tourw´e (2004) studied the refactoring practice within the larger context of software process and examined its impact on the development process. The study, however, did not provide any explicit links and impact assessment of refactoring to the software maintenance aspect. In another recent study, system dynamics (SD), a simulation model was proposed by Cao et al. (2010) that focused on "essential agile practices"–refactoring and pair programming (p. 3). Their results suggested that the cost of implementing the software changes increases in the later phase of development. Furthermore, the cost also increases with reduced productivity with delayed refactoring. Pair programming practice has been found to assist completion of more tasks within planned iterations at a lower cost than non-pair programming efforts. Although their model offered an integrated system approach, the model, however, is limited in its ability to analyze the impact of agile development practices on project deliverable, productivity, and cost. The current study,

on the other hand, stretches further to analyze the impact of the Agile model on software maintainability characteristics based on its direct influence on the source code properties.

While examining XP and its implication on maintainability, Moser et al. (2007), in their case study, reported that the XP method supported the development of easy to maintain code. They studied code properties, MI (maintainability index), moderate growth in coupling, and complexity metrics during development stages. A limitation of this single-case study was that they used the existing maintainability index (MI) measure as defined by Oman et al. (1994), which was originally derived in a non-XP environment. An important assumption of this study was that moderate growth would occur in the maintainability trend, but a decreasing trend over time should result in better maintainability characteristics.

Lastly, the model proposed in this paper may be used to detect the possible issues in development by noting the evolution of maintainability matrices. Within the Agile context, software maintainability state may be monitored using this model during development, in addition to an intervention technique, such as refactoring that can be deployed more often during the iteration. Hordijk and Wieringa (2005) also contended that uncovering the related software design decisions and factors is a critical step to drive the software maintenance cost down. In their study, software maintainability factors were discussed; however, those were only related to change efforts, efforts per unit, and volume of functional change size.

Kajko-Mattson and Nyfjord (2009) studied organizations that adapted ASDM (XP and Scrum) within the software evolution and maintenance cycle. They found that that agility was higher in the implementation phase compared to the pre-implementation phase, and concluded that the Agile aspect of the development approach was relevant in the context software evolution and maintenance. Figure 8 illustrates the typical XP-driven life cycle with maintenance phase that constitutes the planning phase, iterations to release phase, and productionizing phase, according to Kunz et al. (2008).

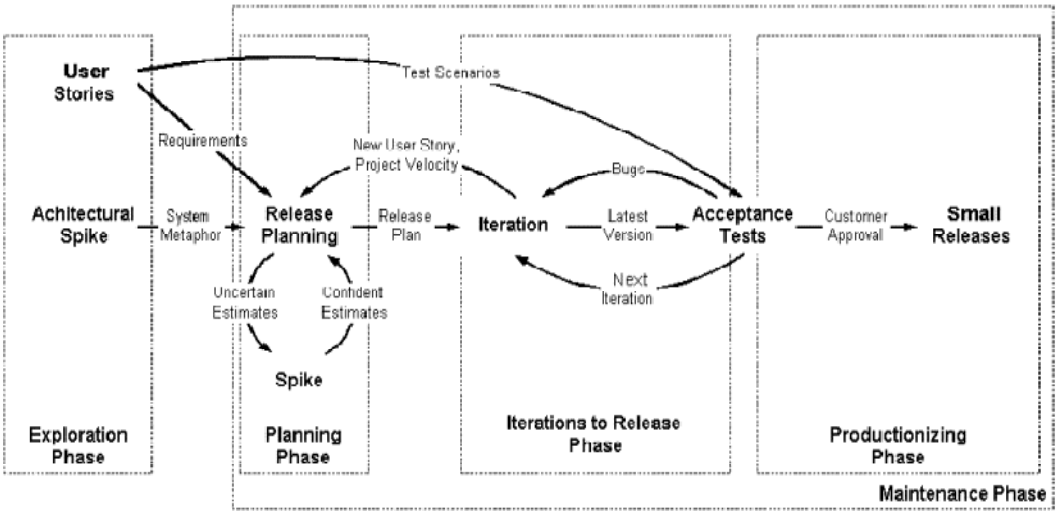


Figure 8. Illustration of eXtreme Programming Life Cycle (XP) by Kunz et al (2008)depicting maintenance phase.

Thus, as revealed in the ASDM related literature, software evolution and maintenance phases are integrated within ASDM, therefore triggering the subsequent need to investigate into software evolution theory and related literature. The next section examines literature pertaining to system theory, resource dependence theory, and software evolution theory within the context of the research question of this study that was probed towards the assessment of impact of ASDM on software maintainability.

System, Resource Dependence, and Software Evolution Theory

System theory is first discussed in order to interlock the theoretical foundation of this study in an upcoming section. According to French et al. (1985), organizations are open systems, and they continually interact with their environment. Furthermore, a system is constituted of two or more related or interdependent subsystems with clear and identifiable boundaries. An organization can be viewed as a goal-seeking open system that aims to create and distribute value (Sanchez & Heene, 2004). They further added that an organization is an open system of resource stocks and flows–a system in which resources flow into and out of the organization and where a "strategic logic" acts as "the operative rationale for achieving an organization's goals through coordinated deployments of resources" (Sanchez & Heene, 2004, p. 45). Software development and maintenance function adhere to open system theory whereby IT development and maintenance functions that may or may not be integrated within the same organization are committed to working on a common deliverable that impact the business value, customer satisfaction, and core business efficiency.

In general systems theory (GST), an organization is viewed as a complex system with boundaries that interface and allow input and output–simplified abstractions are often used within system theory. This system exists within a larger external environment that is constantly exerting pressure on its boundaries, an environment with which the organization must interact (Fossum, 1989). Furthermore, Fossum (1989) contended that systems theory recognizes that a change in one part of the system often creates change

throughout the system. Organizations respond to support the changes irrespective of the origin of the changes in various levels. According to Katz and Kahn's (1978) open systems model, structure first develops out of technical needs and later from internal integration pressures in combination with shifting demands from the environment. This observation applies to software development organizations that update their internal organization processes, resources, and structure to align with ASDM as well as software maintenance function that supports the core development function through software application evolution and maintenance. Note that originally, the Agile model is being introduced to support the rapid business changes (environment) triggering potentially several resultant business processes and software maintenance changes within IT organizations.

Secondly, resource dependence theory was fully developed by Pfeffer and Salancik (1978), and it is based on the core assumption that their environments control organizations. These theorists also suggested how organization management could learn to navigate the harsh seas of environmental shifts and turbulence (Hatch, 2006). Resource dependence theory argues that an interorganizational network analysis can assist an organization's decision-making management to comprehend the power-dependence relationships that subsist between its organization and other connected entities. This knowledge could empower management to identify as well as predict the sources of influence from the environment (Hatch, 2006). By using this knowledge,

management can incorporate counteracting dependence and could offset some of this environmental influence.

Each organizational action taken to reduce or manage uncertainty may amend the connectedness of the system with possible alteration of the business, technical, and management processes and work flow to other organizations. In other words, actions taken to manage interdependence in the long run may increase the interdependence among influential environmental entities, requiring further actions to manage the newly created uncertainties (Pfeffer & Salancik, 2003). Along the same lines, examining the influence of ASDM on software maintainability in this study is a step toward managing the new resultant interdependence within IT organizations.

The organizational response to environment conditions and changes in processes may reach beyond just the core organization and its structure. According to Pfeffer and Salancik (1978), there are two broadly defined contingent adaptive responses: the organization can adapt and change to fit environmental requirements or the organization can attempt to alter the environment so that it fits the organization's capabilities. Adaptations, as posited by Pfeffer and Salancik, can include initiatives that bridge the boundaries of the organization in order to manage interdependence with other organizations. Additionally, these adaptations that are highlighted by resource dependence theory are not limited to changes in the focal organization’s internal structure alone (Tsoukas & Knudsen, 2003). It thus indicates the proliferation of the changes and its implications reaching the related suborganizational processes and governing

management domains through interorganizational dependency links. In summary, resource dependency theory is aptly positioned to strengthen the business case for the research question of this study as well as guide the research design examining implications on software maintainability characteristics.

Lastly, Lehman's (1997) software evolution framework put forward several laws that provide the foundational framework to theorize the impact of Agile attributes on software maintainability and validate the research questions directly. According to Lehman's software evolution laws:

1. Systems must be continually adapted to maintain its usability (Law of continuing change).
2. System complexity increases as it evolves making change implementation harder (Law of increasing complexity).
3. System evolution processes tend to show self-regulation and evolvement as directed by feedback systems (Law of self-regulation).
4. System work rate tends to remain constant over system evolution (Law of conservation of organizational stability).
5. Average incremental growth of systems tends to remain constant or decline (Law of conservation of familiarity).
6. System functional capability must be continually increased to maintain user/business satisfaction and usability of the system over system's life cycle (Law of continuing growth).

7. System quality will decline unless it is rigorously adapted to meet changes in an operational environment both - business and technical (Law of declining quality).
8. Software system evolution processes are multilevel, multiloop, and multi agent feedback systems. (Grubb & Takang, 2003, p. 44)

Software evolution is inseparable from software maintainability, a key software quality attribute that deteriorates with changes that continue to get integrated throughout the evolution cycle (Ping, 2010). Lehman's laws outline the principles common to all—small as well as large-size proprietary or E-type software systems—and hence, assessment of how ASDM impacts software maintainability is incomplete without consideration of this foundational software evolution framework. Sindhgatta et al. (2010) recently evaluated Lehman's laws of software evolution dealing with continuous change and growth, self-regulation and conservation, increasing complexity, and declining quality within an Agile-driven project. The hypotheses of this study were thus guided primarily by the existing ASDM literature grounded in the Agile manifesto, software maintainability literature, and Lehman's laws of software evolution.

Collectively, system theory, resource dependency theory, and Lehman's software evolution framework can guide IT management, ASDM teams, ASDM practitioners, software maintenance teams, and the researcher community to elucidate the interdependencies with causal links between the ASDM as an approach and software maintainability.

Software Evolution and Maintenance Theory

M.M. Lehman has been considered the originator and key proponent of software evolution and maintenance theory that began in early 1970s. Grubb and Takang (2003) iterated the significance of Lehman's theory and posited the same as prerequisites and underlying essentials before attempting software evolution and maintenance analysis. The exploratory analysis from Lehman and Ramil (2002) also concluded that software evolution involves programming paradigms, approaches, languages, usage domain, and that evolution cannot just be restricted to the programming process artifacts such as specifications, designs, and documentation. The Agile software development model has evolved from previous software development models–waterfall and iterative development approaches in specific–and this evolution is in alignment with the feedback system hypothesis by Lehman (1996,), as well as the law of continuing change (Lehman, 1976). Furthermore, all of the software systems evolve over their life cycles, undergoing several changes including modifications to their other attributes such as software maintainability. With maintenance costs exceeding initial design and development costs (Gibson & Seen, 1989), software evolution and maintenance laws underscore the greater need of their retrospection within the context of the Agile model of software development.

Lehman's investigative work spans over 30 years within the software process domain that formulated some software evolution laws, later supported by Turski (1996) and several other practitioners. However, this same theory was criticized by Yuen (1981) and Lawrence (1982) for its observational as well as rigid nature; these laws of software

evolution and maintenance when revisited in 1997 confirmed their relevancy within the changing landscape of software development engineering. Most notably, a recent study from Sindhgatta et al. (2010) evaluated Lehman's laws of software evolution that deals with continuous change and growth, self-regulation and conservation, increasing software complexity, and declining quality within the Agile-driven developed software system.

No.	Brief Name	Law
I 1974	Continuing Change	*E*-type systems must be continually adapted else they become progressively less satisfactory.
II 1974	Increasing Complexity	As an *E*-type system evolves its complexity increases unless work is done to maintain or reduce it.
III 1974	Self Regulation	*E*-type system evolution process is self regulating with distribution of product and process measures close to normal.
IV 1980	Conservation of Organisational Stability (invariant work rate)	The average effective global activity rate in an evolving *E*-type system is invariant over product lifetime.
V 1980	Conservation of Familiarity	As an *E*-type system evolves all associated with it, developers, sales personnel, users, for example, must maintain mastery of its content and behaviour [leh80a] to achieve satisfactory evolution. Excessive growth diminishes that mastery. Hence the average incremental growth remains invariant as the system evolves.
VI 1980	Continuing Growth	The functional content of *E*-type systems must be continually increased to maintain user satisfaction over their lifetime.
VII 1996	Declining Quality	The quality of *E*-type systems will appear to be declining unless they are rigorously maintained and adapted to operational environment changes.
VIII 1996	Feedback System (first stated 1974, formalised as law 1996)	*E*-type evolution processes constitute multi-level, multi-loop, multi-agent feedback systems and must be treated as such to achieve significant improvement over any reasonable base.

Fig. 9 Laws of software evolution–The Nineties view.

Note. (IEEE) Retrieved from Lehman, M. M., Ramil, J. F., & Wernick, P. D. (1997), p. 21.

This software evolution theory closely intertwines the research questions in this study from various perspectives. For instance, the law of continuing change (Lehman, 1974) implicated software system adaptation as a dominant force that in turn strengthens continuing growth theory (Lehman, 1980) and furthermore warned about possible declination in software quality (Lehman, 1996) in the absence of adaption and maintenance. It is evident in this chain of interrelation of laws that the need of software

adaptation to the changes and changing environment originates from the quest of maintaining higher software quality, improving maintainability, and resultant customer satisfaction. When examining the influence of the Agile model on software maintainability and its sub-characteristics, software evolution laws signify the urgency to improve maintainability to avoid declining quality (Lehman, 1996). According to this declining quality hypothesis, the software quality will appear to be declining unless these systems are rigorously maintained and adapted to environmental changes in which they operate.

Additionally, to attain satisfactory software evolution, the conservation of familiarity hypothesis emphasized the development and maintenance team's mastery over the software content and behavior. This law of software evolution also states that excessive growth, however, may impact this familiarity level negatively. With domain knowledge distribution in Agile teams through collective code ownership practice, ideally this familiarity and hence the software analyzability, one of the key maintainability sub-characteristics should be unchanged. In reality, however, this quality trait may be influenced by other factors outside of ASDM itself. Additionally, this 5th law along with Lehman's 3rd, 4th, and 8th Laws of self regulation, conservation of organizational stability, and feedback systems hypothesize and focus on the stability characteristics of software as it grows. The next section addresses the software evolution and maintenance outlook pertaining to ASDM, which primarily stem from the changing needs of business domains (user).

Software Evolution and its Relevance within ASDM

The chaotic behavior of the software development process before release is attributed mainly to feedback loops, according to the law of the feedback system (Lehman, 1996). As the system stabilizes due to feedback, the ripple effect of the growth of a software system going through evolution is evident as hypothesized by Lehman (1996). Within the ASDM-driven software evolution, increasing software stability and reducing chaos towards the end of each release is suggestive of the adherence of this law given that frequent feedback is an integral part of ASDM.

Increasing software complexity is connected to poor software maintainability. Lehamn's 2nd law of evolution postulates that complexity of a software system increases as the program evolves unless steps are taken to reduce this complexity. The complexity aspect of software is significantly non-trivial within ASDM as it is addressed through several Agile practices such as refactoring and simple design principles.

For successful software evolution, Hulse et al. (1999) contended that a thorough understanding of the architect's intentions about software organization is a must. They further argued that software maintenance costs can be reduced significantly when the architecture is well defined and documented. In another review, Porter (1997) attempted to group the maintainability related factors into four distinct groups: product, people, process, and task. He further contended that software complexity and software structure, the subgroups of software maintenance category, includes the metrics and programming approach, which in turn, influence software understandability, which is often

synonymously termed as *software analyzability*. Secondly, system structure elements are the main determinants that explain change implication to software system. The software change and its ripple effect (Ravichandar et al., 2008), software architecture considerations and its implications (Jensen et al., 2008), design patterns and software language domain, fall into system structure subcategory. These studies were also touched on in the maintainability related discussion in this research study.

During every software development effort, source code is added, changed and removed as a result of the requirement integration. In turn, the software structure as well as execution behavior of the code may have changed after implementation of the code changes (Law & Rothermel, 2003; Orso, 2003). Some code changes will also have a long-term impact on the software project that is often difficult to estimate, according to Herzig (2010). He also contended that instability often refers to low quality. Within the software life cycle, software product is expected to withstand storms of changes. As a result, deterioration continues degrading software changeability further with successive changes. The law of increasing entropy or Lehman's 2nd law states that entropy of software, also reflected in its unstructuredness, increases with time unless specific efforts are undertaken to maintain or reduce it. Fowler (2001) advocated simple design and refactoring, key practices of ASDM, to address software entropy related concerns.

When analyzing software development problems in their study, Hanssen et al. (2010) argued that code duplication is an unfortunate yet common way of reacting to "cognitive overload" originating from poor analyzability of the software (p.). They

further contended that the testability and stability of software gets impacted due to size of the code and complexity. Poor test coverage, unstable and inconsistent existing tests could potentially impact even changeability, discouraging the developers to change the software code with confidence.

Code reuse approach has its own value, and it is perceived as an expedited avenue in some development scenarios. Although code reuse saves time and cost with saved coding efforts, Baldo and Thomas (1996) argued that it may redefine maintenance. Porter (1997) contended that ad-hoc changes are relatively inexpensive and are rapidly applied, but are likely to degrade the software structure; but, changes that are planned and that preserve the structure may not impact the software structure. When discussing change-prone software modules, Harrison (1996) argued that ad-hoc changes may be compounded in software modules that frequently change. Within the context of ASDM, changes are not really ad-hoc in their emergence nor fully planned in advance.

Software changes are entertained and accommodated relatively well in ASDM-driven development cycles. However, even successful development projects do not necessarily account very well for future maintainability. According to Castro and Mendes-Moreira (1996), the cost and effort spent in software maintenance are the "dues of success" (p. 5). When using program comprehension theories in their case study, Von Maurhauser and Vans (1996) suggested that maintenance task type, background or exposure of the software application, domain expertise, and language expertise impact software maintenance. Furthermore, Schneberger (1996) posited that with technology

changes, maintenance bottlenecks can also shift, which is well noted in distributed architecture. The contention is that the ASDM model, including its underlying principles, practices, tools, and technologies that support these methodical practices may influence the software maintenance and maintainability differently, as well.

System documentation has been debated within numerous software evolution studies. Tryggeseth (1996), when studying impact of documentation availability on maintenance productivity, reported that system-level knowledge has greater impact on software maintenance quality when compared to resource programming knowledge. In another controlled experiment, Visaggio (1996) corroborated that the quick fixes indeed degraded software structure and were less reliable when compared to iterative enhancements. None of these studies however examined the ASDM approach primarily because the Agile model realistically did not begin spreading its roots until early 2001.

Individual change and its implications for costs and risk comprehension are the first step towards developing an outlook towards software evolution and maintenance. Benestad et al. (2009) discussed various approaches taken in several other researchers' empirical investigations and highlighted that individual change-specific attributes can be alternatively used to assess software evolution and maintenance. They referred to these studies as *change-based studies* in their extensive literature review targeted to improve the understanding of causal factors of cost and risk through identification of change attributes. Lehman's laws, on the other hand, were formed based upon measurement of changed or impacted software components over the release cycle. The software

evolution and maintenance theory is here to stay and will continue to inform software engineering practitioners, specifically functional within ASDM-driven development initiatives. The next section of the literature review focuses on software maintainability and related themes.

Software Maintainability Themes and Implications

Software maintainability, according to ISO/IEC 9126-1(2001) is the capability of the software systems to be modified. Maintainability is also defined as "the ease with which a software system or component can be modified to correct faults, improve performance, or other attributes, or adapt to a changed environment"(IEEE, 1990, p. 127). Furthermore, according to the IEEE (1517 -1999) standard, the maintenance process is performed to modify existing software and correct a defect or deficiency or to adapt the software to comply with new or updated requirements. The purpose of the maintenance processes is to modify, migrate, or retire an existing software application. Because the efforts undertaken on software changes and fixes are a major cost driver (Hordijk & Wieringa, 2005), software maintainability is the critical quality aspect of the software life cycle. Within the ASDM life cycle, improved software maintainability has become a significant capability dimension for IT management. The stake extends beyond Agile practitioners and development project teams.

Software companies continue to select outsourcing options when addressing their software maintenance and support, but mostly view this as a competitive advantage to their core business (Ahmad, 2006). A software maintenance outsourcing discussion is,

however, mainly out of scope of this study due to the necessity of controlled examination of the research question posed in this study. Chen and Huang (2009) conducted a qualitative study wherein they examined the impact of software development problem-factors on software maintainability. They reported that software process improvement (SPI) is capable of increasing software maintainability level moderately. Their analysis also yielded ranked software development problem factors listing inadequate source code comments, obscure documentation, inadequate change related documentation, poor traceability, noncompliance and inconsistency to adhere with programming standards, continually changing requirements, frequent turnover within active project team, poor adherence to programming techniques, and inadequate considerations for software quality requirements. Although some of these factors appear to be relevant to any development model in general, this study did not include any Agile project-specific data and fell short of building any strong inferences applicable to ASDM.

The software maintenance phase constitutes several core activities that vary in terms of their long-term and short-term impact on software maintainability. Osborn (1985) defined *software maintenance* as the post deployment activities that are a must to maintain the acceptable level of operation of the software system. He further categorized the maintenance into three groups: correcting, enhancing or improving, and perfecting. Dekleva (1992), however, criticized these categories, as “too broad classification” (p. 356).

Table 4:

Types of Software Maintenance

Types of software maintenance	
Type	Description
Corrective	Repair design and programming errors
Adaptive	Modify system to environmental changes
Perfective	Evolve system to solve new problems or take advantage of new opportunities
Preventive	Safeguard system from future problems

Note. Retrieved from Hoffer et al. ,2008, p.564.

Software Maintenance Decomposition

The four types of software maintenance, according to Lientz and Swanson (1980), are corrective, adaptive, perfective, and preventive maintenance that were further normalized internationally in the ISO/IEC 14764 (2006) standard.

1. The repair of software defects is handled through corrective maintenance. A software defect may be linked to software design errors, logic errors, or programming/coding errors, according to Takang and Grubb (1996). Software design

errors often occur due to incorrect, incomplete, wrongly communicated changes. Software logic errors could result from invalid assumptions, invalid tests, inaccurate conclusions, incorrect implementation of design specifications, faulty logic flow, or incomplete test of data. Programming or coding errors are the results of incorrect implementation of logic design and incorrect usage of the source code logic. Software defects could also be caused by data processing errors and system performance errors. These software errors, commonly referred to as *residual errors* or *software bugs*, deviate the software from conforming to its originally agreed specifications or requirements.

A bug reported from end users often initiates corrective maintenance (Coenen & Bench-Capon, 1993). Examples of corrective maintenance include: correcting a failure to test for all possible input conditions or a failure to process the specific data type or specific record in a file (Martin & McClure, 1983). In the recent study, Li et al. (2010) found that software size and complexity, maintainer's experience, tool and process support, and domain knowledge are the most influential cost drivers of corrective maintenance.

2. Adaptive maintenance is composed of adapting the existing production software to changes in the environment, such as the hardware, or the operating system, or newer sub components of software. The environmental changes could also include business rules, government policies, work patterns, and software and hardware operating platforms (Takang & Grubb, 1996). An example of a security policy triggering changes in a software system one that includes the data encryption mechanism for data in transit

and storage. This change could require the organization and its software systems to make significant changes to its software systems to accommodate the encryption requirements to comply with security policies. Other examples are: an implementation of a DBMS or database management system for an existing application system, and an adjustment of two programs to make them use the same record structures (Martin & McClure, 1983). Another example is to include the new business rules in the programming logic in response to changes in the business processes to address customer feedback on the longer delivery time. According to McConell (1996), the development team can face a 25% change in software requirements during the software development projects.

3. Perfective maintenance often handles new or changed user requirements and is related to functional enhancements to the software system and changes to improve the system's performance (Van Vliet, 2000). Software, once deployed successfully, goes through several changes during the life cycle, with obvious increases in the number of requirements. This is based on the premise that as the software becomes an essential tool for the business organization, the users continue to ask for new cases or functionalities well beyond the scope for which the software was initially designed and developed (Takang & Grubb, 1996).

Examples of perfective maintenance include modifying the engineering software program to include newly added tool standards and vendor details, adding a new business performance report in the operation management dashboard system, adding error handling and capture module to make it more support-friendly.

4. Preventive maintenance tasks and activities are aimed to enhance the software maintainability. Some of these actions are adding comments in the code, improving the modularity of the system, improving the system documentation, and improving the database architecture (Van Vliet, 2000). Over the life cycle, the impact of corrective, adaptive, and perfective changes can be seen in the increased system's complexity (Takang & Grubb, 1996). With these continual changes, software complexity increases, reflecting a deteriorating structure. The preventive maintenance function is targeted to maintain or reduce this impact on software. This work is also referred as *preventive change* (Stafford, 2003).

Users of a software system could report the symptomatic issues, such as performance degradation or non-satisfactory software behavior, which result from a lack of preventive maintenance. The change usually gets initiated from within the software maintenance organization in this category, with the intent to improve the software code's understandability and reduce the maintenance work in the future (Takang & Grubb, 1996). Examples of preventive change include refactoring, restructuring of the program, code cleaning and optimization, and updating documentation with the relevant and current software specific information. Stafford (2003) suggested that, among these four types of maintenance, corrective maintenance is the only traditional maintenance type. The other maintenance types can be considered as software evolution. The term *evolution* has been used to characterize the growth dynamics of software since the early 1960s (Chapin et al., 2001), and it is widely used in the software maintenance

management and practitioner community. The *Journal of Software Maintenance* appended the term *evolution* to its journal title to echo this transition (Chapin & Cimitile, 2001).

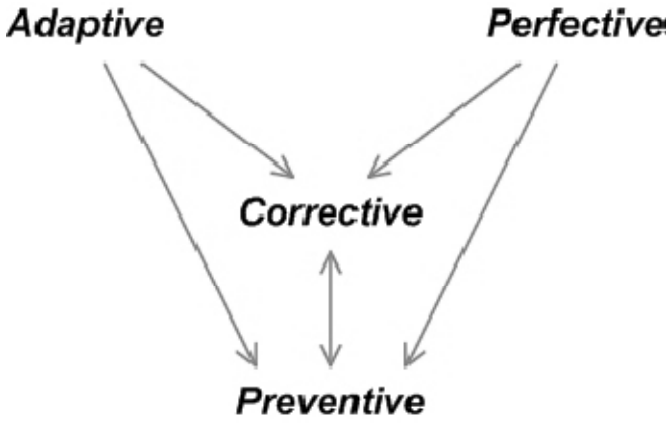


Figure 10. Illustration of relationship between types of software change (Retrieved from Kunstar and Havlice, 2008).

Software changes, as discussed earlier, were categorized into three classes by Basili and Weiss (1984), Lientx et al. (1978), and Swanson (1976) as adaptive, corrective and perfective. Within the Agile development cycle, Cao et al. (2010) argued that refactoring practice supports implementation of perfective change. New requirements planned for upcoming iterations fall into the adaptive change class, according to Highsmith and Cockburn (2001). Corrective change prioritization decides their implementation order, which is also a function of customer involvement. A change to the recently developed feature in current or previous iteration is also a corrective change within the Agile development cycle. The Agile model thus influences the ease or difficulty with which these changes may be implemented.

Software Quality: Internal and External Characteristics and Subcharacteristics

This study uses the ISO/IEC 9126 software engineering quality model to examine the implication of ASDM on software maintainability. The ISO 9126 software quality model categorizes three distinct, yet related views on software quality: (a) internal quality view, which emphasizes on the software properties that can be measured without executing it; (b) external quality view, which emphasizes the software properties that can be witnessed during software execution; and (c) in-use quality view, which emphasizes the software properties experienced by varied classes of users during the operation and maintenance life cycle stages. Heitlager et al. (2007) asserted that internal quality impacts external quality, which in turn impacts software quality in use.

Within the ISO/IEC 9126–TR (technical report), the consensual measure of metrics exists for evaluating various product quality characteristics and, in specific, TR-9126-3 includes internal metrics. For the maintainability characteristic, 16 external and 9 internal quality measures are defined within the 9126 standard.

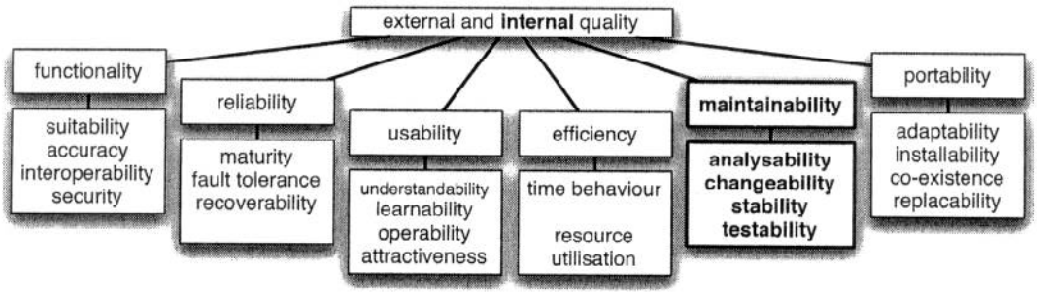


Figure 11. Illustration of software quality notion; Heitlager et al. (2007), based on ISO/IEC 9126 model of software quality.

In this study, the software maintainability and its four internal sub-characteristics were mapped to software code level measures that are discussed to further detail in chapter 3.

Software Development and Maintainability

When debating about inclusion of software maintainability into the development approach and processes, several researchers favored this reiteration even within modern software development methodologies. Singh and Goel (2007) proposed a model of preventive maintenance that outlines life cycle of maintenance request within the context–the analysis phase and design and implementation phase. Being that the software maintenance is a microcosm of the software development cycle, as argued by Singh and Goel (2007), required considerations pertaining to software maintainability are needed during the development phase.

Huang (2009), in a quantitative study, showed that problem factors in the software development phase can negatively impact software maintainability. Huang further suggested that the requirement of software maintainability should be taken into account. Furthermore, the associated software problem factors should also be properly dealt with when performing analysis, design, and implementation activities during the software development phase in order to achieve higher levels of software maintainability.

Jain et al. (2008) posited that the recent exponential development and growth of technology and increase in user demands has resulted in an increase of complexity in the systems integration and development process. Complexity in this context, is "the degree to which a system or system component has a design or implementation difficulty

associated to understandability, and verification" (IEEE, 1990, p. 46). Complexity, according to Evans and Marciniak (1987), is "the degree of complication of a system or system component determined by the number, intricacy of interfaces, the number and intricacy of conditional branches within the program, the degree of usage of nesting, and the types of data structures" (p.33). Ravichandar et al. (2008) argued that a software system's complexity increases the susceptibility to change with given dynamic landscape of business, technology, and user needs. They further examined the ripple effect of needs change and found that capabilities-based design improved the change-tolerance of the system with volatile needs.

ASDM focuses on delivering working software or versions through iteration-driven approaches. Hanssen et al. (2010) argued that this focus coupled with high velocity could negatively impact the quality of this delivered software at the end of the iteration. This could in turn demand additional efforts before the release when the software is tested fully. The software entropy, according to Hanssen et al., however, grows, from release to release.

Development Model and Software Maintainability Correlation and Research Method

The true experimental setup to assess the impact of identified ASDM on software maintainability is an expensive and unrealistic quest. This limitation forced this research design to be based on post facto or already existing data. Prolonged engagement of resources for a longer time is not practical within a business organizational context.

To demystify the ongoing software maintainability related challenges in various IT organizations that are adapting or have successfully adopted the Agile model over the last decade or so; laws of software evolution by Lehman (1996) and Belady (1976) provide much needed theoretical guidance and relevant lenses to study the Agile-driven software evolution. These laws are mostly applicable to Sindhgatta et al. (2010), who analyzed the software evolution in an Agile development case study–in a specific scrum-driven development project within the framework of Lehman's laws relevant to continuous change and growth, self-regulation and conservation, increasing complexity, and declining quality. Most evolution and maintenance laws were reported to be followed by the Agile-driven system in this unique and only study so far that attempted to validate the Lehman's Laws within Agile dynamics. Rico (2008) studied the relationship between Agile factors and website quality in a qualitative case study, and suggested that iterative development and customer feedback related to website quality. In another quantitative study, Hanssen et al. (2010) examined how the maintainability of the system could degrade with continuous change over time, showing a negative relationship between agility and software entropy. Lastly, Olague et al. (2006), in their quantitative research design, assessed software maintainability, but with intent to examine the utility of information theory-based metrics within the Agile-driven approach.

Within a systematic review of all empirical studies of Agile software development conducted by Dyba and Dingsoyr (2008) using existing data up to and including 2005, they found 36 primary research studies, with only 4 studies related to software quality.

Furthermore, none of these 4 studies were focused on software maintainability, and none were conducted within a seasoned Agile team composition setting or environment. Instead, the product quality was compared between traditional and Agile-driven approaches within experimental settings that were not fully integrated with th eAgile approach. According to Huo et al. (2004), a software quality comparison between Agile and traditional approaches such as waterfall, is not only difficult, but an unrealistic endeavor due to varying development conditions and cost. Additionally, the accuracy of the result in such experimental research studies may not reveal the causal links between development approach and software quality characteristics including maintainability.

Notwithstanding, many studies on the Agile methodologies and individual Agile practices have focused on their effectiveness within Agile-driven development projects (Cao et al., 2010; Erdogmus & Williams, 2003; Williams & Kessler, 2000) and not their impact on maintainability related quality characteristics. On the other hand, studies employing Lehman's software evolution framework, such as the study by Sindhgatta et al. (2010), have not studied maintainability in their assessment of software quality within a specified software life cycle. As a result, this study transcended the void of studies assessing the influence of development model on software maintainability.

This study addressed this paucity of research analyzing software maintainability using the tenets of Lehman's Software evolution and maintenance theory. The next chapter will present a method to do so that follows the ISO/IEC 9000/9126 (2003) software quality standard as well as the approach followed by Kanepolious et al. (2010)

to measure software analyzability, changeability, stability, and testability across multiple software releases driven by ASDM.

The literature reviewed has highlighted that the quasi-experimental as well as correlation research as the predominant research method employed to measure the direction and the extent to which ASDM practices impacts software quality attributes and software development project success. Quasi experimental research has been used because it allows for the measurement to assess the impact of the treatment on the selected dependent group both before and after a treatment of the independent variable. In this case, the impact or influence was the changes in maintainability and its sub-characteristics. Chapter 3 addresses the research method that was used to assess the impact of ASDM on software maintainability and its sub-characteristics.

Metrics and Related Research

Software metrics have been widely used in software quality assessment within software engineering domain (Boehm, Brown, & Lipow, 1976). A literature search related to object oriented metrics resulted with some key studies from Ohlsson et al. (2001), Alshayeb and Li (2003), Gyimothy et al. (2005), Basili et al. (1996), Tsantalis et al.(2005), Nakatani and Tamai (1997), Mens and Demeyer (2001), Chidamber and Kemerer (1996), and Olague et al. (2006).

Ohlsson et al., in their 2001 case study, tracked system evolution to identify decaying components to avoid software brittleness. They used defect fix reports, degree of interaction, total number of changes to the source files, unique number of files fixed in

a specific component, average number of changes in source files, and growth in source and executable files. The software, however, was developed using a non-object oriented approach. Alshayeb and Li (2003), in their work related to design efforts prediction within iterative software development approach, used weighted method count (WMC), depth of inheritance tree (DIT), lack of cohesion of methods (LCOM), number of local methods (NLM), coupling though abstract data type (CTA), and coupling through message passing (CTM). Gyimothy et al. (2005) in part attempted to revalidate Basili et al.'s (1996) findings related to the fault-proneness ability of their metrics. The Basili et al. (1996) study focused on the Chidamber and Kemerer metrics - WMC, DIT, RFC, NOC, CBO, LCOM, and LCOMN including LOC metric. When studying the predictability based on the probability of change, Tsantalis et al.(2005) used Chidamber and Kemerer's object oriented class metrics.

Nakatani and Tami (1997) also studied the evolution patterns of multiple software systems using Chidamber and Kemerer and the LOC metrics in an attempt to comprehend and explain the evolution of boundary, domain, and common class for iterative software development processes. Mens and Demeyer (2001) identified "predictive and retrospective software evolution metrics" and their benefits in software quality evaluation and assessment (p.14). McCabe's Cyclomatic Complexity (Lorenz & Kidd, 1994; McCabe & Watson, 1994) is another widely accepted complexity-related metric that is being used in this study.

In addition, Subramanyam and Krishnan (2003) also studied single, but relatively large software versions using the Chidamber and Kemerer metrics suite and the LOC metric. They contended that there is a high correlation between object-oriented metrics and their usefulness as indicators of software quality. Lastly, Olague (2007), when examining fault prediction and ability of several object-oriented metrics, reported that object-oriented metrics could be used within the Agile-driven software development approach beyond just the assessment of initial quality. They successfully corroborated their argument by using the set of object-oriented metrics within the Agile-driven software system study. Because highly iterative or ASDM-driven software increases in size and capability through every iteration and release cycle, usage of object-oriented measures for subsequent releases remains valid. This study used several existing measures tabulated in Table 5 as shown.

Table 5

Metrics Used in this Study and Some Key Studies that Utilized These Measures

Metrics	Key studies utilizing these metrics
Cyclomatic Complexity (McCabe)	Chidamber & Kemerer (1994), Kanellopoulos et al.(2010)
Coupling Between Objects (CBO)	Chidamber, Darcy, Kemerer (1998), Harrison, Counsell, Nithi (2006), Krishnapriya & Ramar (2010)
Unit Size	Kanellopoulos (2008), Heitlager (2007)
Test Coverage	Janzen, D.C. (2006), Heitlager (2007), George & Williams (2004)
ASSERT count	Janzen, D.C. (2006), Heitlager (2007)
Duplication	Kanellopoulos (2008), Heitlager (2007),Kanellopoulos et al.(2010)

Software Maintainability Decomposition

Software maintainability is defined as, the capability of the software systems to be modified (ISO/IEC 9126-1, 2003) Furthermore, these modifications span corrections, adaptations, or improvements to changes originating in the environment (outside of the software) as well as in requirements and functional specifications (Chen & Huang, 2009).

Within the context of this research study and underlying research question, software analyzability according to ISO/IEC 9126-1 (2003) is the software system's capability to allow identification for parts that should be modified. In other words, it is a software characteristic that reflects how easy it is to diagnose the system for deficiencies or to identify the parts that need to be modified (Chen & Huang, 2008; Heitlager et al., 2007).

Software changeability is the capability to enable a specified modification to software system to be implemented, according to ISO/IEC 9126-1 (2003). It is another internal software quality sub-characteristic that is indicative of how easy it is to make software adaptations (Chen & Huang, 2008; Heitlager et al., 2007) or specified modification.

According to the same IEC 9126 standard, software stability is defined as the capability of the software product to avoid unexpected impact from modifications of the software. This sub-characteristic of software maintainability informs how easy it is to keep the system in a consistent state during modification. In short, stability shows how capable software is to remain stable after being modified (ISO/IEC 9126-1, 2003).

Lastly, ISO/IEC 9126-1 (2003) defined *software testability* as, the capability of the software to enable modified software to be tested. Alternatively, this fourth independent variable of this study represent that how easy is to test the system after the implementation of modification.

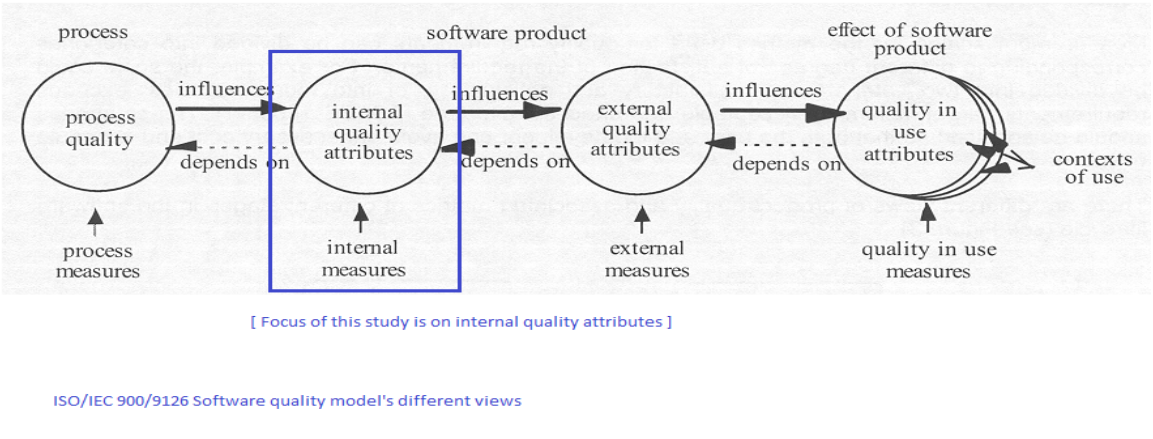


Figure 12. ISO/IEC 9000/9126 Software Quality model's different views.

The suggested metrics by ISO/IEC 900/9126 for software maintainability and its sub-characteristics are change implementation elapse time and change impact for changeability, activity recording for analyzability, and re-test efficiency for testability. However, these metrics are founded on the observation of the interaction of software and its environment that includes the software developer, maintainers, analysts, and testers, but not based on direct observation of the software product, as critiqued by Heitlager et al. (2007). They further argued that these measures are based on comparison of the software with its specifications that are vulnerable to incompleteness, invalidity, obsoleteness, and incorrectness. What follows next is the relevant, applicable, and

already used set of metrics based on the direct observation of software application as indicated in Figure 12 showing software internal quality attributes.

Table 6

Correlations Between Software Maintainability Sub-Characteristics and Software Attributes-Adopted from Hegedus et al. (2010).

	Analyzability	Changeability	Stability	Testability
Complexity	n.a	(-) Negative	(-) Negative	(-) Negative
Coupling	(-) Negative	(-) Negative	(-) Negative	(-) Negative
Size	(-) Negative	(-) Negative	(-) Negative	(-) Negative
Clones	(-) Negative	(-) Negative	n.a.	n.a.
Unit Test Effort	(+) Positive	n.a.	(+) Positive	(+) Positive

Much research related to software maintainability has occurred over last 30 years, touching its multifaceted domain. Riaz et al. (2009) conducted extensive systematic review on software maintainability prediction and metrics based on 15 key and relevant studies. Their findings showed that software size, software complexity, and software coupling collected at the source code itself were the most commonly used predictors of software maintainability. Hegedus et al. (2010) summarized the correlations between software system properties and software maintainability-related quality sub-characteristics as shown in Table 6.

This data is extracted based on several other experiments conducted by Basil et al. (1996), Yu et al. (2002), Subramanyan and Krishnan (2003), Gyimothy (2005), Olague et al. (2007), and Siket (2010) in the area of system attributes such as complexity, coupling, and size. In addition to these properties and underlying measures underscored by Riaz et

al. (2009) and Hegedus et al. (2010), this study incorporates additional measures used by Kanellopoulos et al. (2008, 2010) pertaining to software stability and testability sub-variables.

Lastly, Yu et al. (2002), Subramanyan and Krishnan (2003), Gyimothy et al. (2005), and Olague et al. (2007) also found that software complexity, coupling, and size were positively correlated to fault proneness, whereas, unit test effort was negatively correlated to it. Thus, within the context of ASDM, the independent variable of this study, Riaz et al.'s systematic review and Kanellopoulos et al.'s studies provided key foundational grid pertaining to chapter 3.

Analytical Hierarchy Process (AHP)

The Analytic Hierarchy Process (AHP) is a comprehensive framework that deals with the intuitive, the rational, and the irrational when making multi-criterion or multi-actor decisions with and without certainty for multiple alternatives that develop ratio scales (Harker & Vargas,1987). The process decomposes a problem into smaller elements and then applies pair-wise comparison judgments to develop priorities in each hierarchy.

AHP is a common decision technique in the field of decision science (Liao & Lin, 2009). Paired comparisons within AHP allow the researchers to seek answers to measurement-oriented scientific questions (Saaty, 1997). Arrington (1984) referred to AHP as a "paramorphic technique" (p. 298) that he further contended could be used to construct scales of attributes that drive preferences for certain items over others.

Kanellopolus (2008) used this process in his recent study when validating data mining model and evaluated software maintainability characteristics. Thus, within the context of evaluation of software maintainability and its sub-characteristics with a given set of measures, AHP is the right first step towards the assessment of influence of ASDM.

In this review of software evolution and maintenance, and its relationship to maintainability and its sub-characteristics, we see that the non-optimal alignment between development model and maintainability objectives is not just an IT organizational concern, but also a socially significant issue. The literature has pointed to three major domains that have facilitated software maintenance and ASDM relevant discussions: ASDM model and its key attributes, Lehman's software evolution theory, and software maintainability.

Research Methodology Preview

The core purpose of this study was to investigate the influence of the Agile software development model on resulting software maintainability characteristics as noted earlier. The nature of the research problem drove the choice or selection of specific research methods (Leedy & Ormrod, 2001; Singelton & Straits, 2005). The experimental design methodology is well suited for the studies of causality and impact assessment, including before and after assessment (Halat, 2007; Singelton & Straits, 2005; Vogt, 2007). Creswell (2003) observed that a quantitative approach integrates the examination of hypotheses and questions using measurements applied to collected data. Within true experimental design, the inferential gravity is high, therefore allowing the researcher to

control the experimental variables to greater extent. This control further bolstered the validity, both internal and external.

The primary requirements for conducting a true experiment, however, includes discrete manipulation of an independent variable, measurement of a dependent variable with single or multiple groups, and steady environmental conditions across groups under the investigation (Changeau, 2004; Creswell, 2003; Singelton & Straits, 2005). Conducting true experimental conditions may often be an unrealistic, expensive, and impractical venture specifically within software development projects within IT. Furthermore, IT researchers (Dyba & Dingsoyr, 2008) argued about the strength of internal validity of the experiments that are conducted using unqualified participant such as students, and within laboratory settings specifically within Agile and software quality-related studies. Vogt (2007) warned that when choosing specific design, it is critical to evaluate the design strengths and weakness including associated costs and benefits.

When examining existing or post facto data, researchers (Creswell, 2003;Vogt, 2007) acknowledged the lack of control limiting random assignment. However, within the field setting of mature software development projects driven by the Agile model, the benefit is the retrieval of realistic data for the analysis with high possibility of the generalization of the outcomes. A good experiment should pave the way for better understanding of potential causal and influential relationships between variables in a real-life setting. Within experimental treatment, Singleton and Straits (2005) contended that the challenge is in the ability to separate and examine independent variable treatment

apart from the effects of extraneous variables. Research design has a direct sway upon efforts to mitigate threats to external and internal validity (Trochim, 2001). When analyzing the existing or post facto data within quasi-experimental setup, the researcher needs to be selective about the groups, and this selection could benefit from the existing and previous literature.

This purposive selection of subject with well-defined criteria improves the internal validity further. Internal validity is an important factor within quasi-experimental design. Experiments with high internal validity suggest strong inferences when interpreting potential causality of relationships or influence of treatment on the dependent variables. The effects of spurious variables are isolated and controlled (Singleton & Straits, 2005) in such cases. One of the potential threats to internal validity is the researcher's bias, as cautioned by Trochim (2001). A diffusion threat, on the other hand, within the research setting utilizing past or existing data, is minimal to none.

External validity is "the degree to which the conclusions in your study would hold for other persons in other places and at other times" (Trochim, 2001, p. 42). This study will be conducted within an identified U.S.-based IT organization with stable and mature Agile model practicing teams. The external validity of this study was thus limited due to the constraints of limited available data from the software development organizations within business corporations.

This study integrated AHP within data analysis steps. This technique integrates both qualitative and quantitative dimensions of decision (Saaty, 1980) and enriches the

quality of analysis. Previous similar studies (Kanellopolious, 2008; Miranda, 2001) used this technique, which elicits subjective judgment from the domain experts using the scoring method as highlighted in Appendix B. Furthermore, as indicated earlier, this study analyzed the time series data and measured the variation or changes in software maintainability, analyzability, changeability, stability, and testability of software systems developed using the Agile model. Repeated measure design is considered statistically powerful because it controls for potential sources of variability, according to Aczel (2006). Regression analysis within subjects will be used when analyzing the data. Note that subjects in this study are correlated samples and stayed the same before and after the measurement. This study essentially involved multiple observations over time for the same software system.

Regression analysis was conducted on the collected data for maintainability characteristics to measure the changes in maintainability at every release point. Regression analysis is a hypotheses-testing technique that can be used to understand the significance and causal impact of the independent variable in the study (Larson & Farber, 2006). For this research, updated values of the dependent variables (SA, SC, SS, and ST) at every iteration were measured. The primary data analysis was presented with the results from regression analysis technique. SPSS and Minitab, the statistical software tools, were used to perform the regression analysis. Four research questions and four hypotheses statements guided the study. Hypothesis testing led to the determination of

the impact of ASDM on maintainability characteristics and also answered the research questions of this study.

Chapter Summary

This chapter highlighted literature that reiterates the concern of IT management related to higher software maintenance costs. The literature also suggested that although software maintenance continues to gain more notoriety, the Agile software development model as a modern software development methodology may be positioned to influence software maintainability and its sub-characteristics: analyzability, changeability, stability, and testability. There are ongoing debates and concerns about applicability of Agile within complex development projects and its long term implications within software life cycle, specifically in the post-deployment phase. ASDM benefits the organizations by delivering business value earlier; however, the literature discussed in this chapter clearly substantiated the need to examine its impact on software maintainability by highlighting this current gap. The lack of maintainability considerations in the development stage has been argued to be a significant detriment to the software quality, resulting into higher software maintenance cost, which further impacts the core organization's revenue. Previous research found that ASDM continues to extend its adoption with several key underlying practices influencing software delivery, complexity, development productivity, cost of development, design simplicity, as well other software quality processes.

The literature uncovered that a plausible analysis for the software maintainability particularly within the context of the Agile-driven software development life cycle can be explained by the set of Lehman's software evolution framework. With business agility becoming a norm, this researcher's contention is that ASDM may be an influential development approach to demystify the software maintainability puzzle and probably leverage its methodical practices to attain higher software maintainability. It could be possible for Agile practitioners, operation and maintenance programmers, as well as IT management that is struggling to cap the software maintenance cost, to construct more effective alignment between ASDM and maintainability objectives if the nature of influence is understood between ASDM and software maintainability related attributes.

To effectively assess how software maintainability impact ASDM, the next chapter presents a research design for determining whether and how ASDM model impacts software maintainability, as argued by Kajko-Mattson et al. (2006), and in turn influences software analyzability, software changeability, software stability, and software testability. A critical assumption of this study is the assertion by Lehamn and Belady (1976) and Lehman (1980,1996) that software evolution and maintenance laws govern the software life cycle and influence the critical attributes such as growth, complexity, entropy, and maintainability. Another assumption about ASDM is that the model adheres to the Agile manifesto (2001) and underlying Agile principles. In chapter 3, a detailed description is provided that identifies the research design, methodology, and its rationale to assess the impact of ASDM on software maintainability sub-characteristics through

source code property measurement. The chapter also presented data selection methodology, along with a detailed description for the operating measures for dependent variables and analysis process. Particular attention is given to simplified explication on how this study's research design align with current design approaches employed in similar studies with relevance to foundational concerns related to software maintainability.

Chapter 3: Methodology

In this chapter, the research method employed is explored to examine the impact of ASDM on software maintainability for the software applications that are being developed within a U.S.-based technology manufacturing organization over the last 2 years. More specifically, the chapter addresses the questions related to the extent ASDM impact software maintainability: what are the changes to software analyzability, software changeability, software stability, and software testability, and hence, overall software maintainability, if any? First, the research design is presented along with a restatement of the research questions and hypotheses. Next, the research setting and case study data are addressed as well as the instrumentation and materials used in the study. The chapter ends with a discussion of how the data was collected and analyzed.

Research Design and Approach

The approach employed in this study is post facto quasi-experimental quantitative design, which involved developing a hypothesis that points to the impact of the independent variable on dependent variables (Creswell, 2009). Quantitative research was appropriate for this study because it has been used on numerous occasions to assess the impact of the treatment on the selected group both before and after a treatment. It also enabled me to statistically interpret results, measure the direction of influence and changes in dependent variables, in this case, software maintainability and related subcharacteristics: analyzability, changeability, stability, and testability. Some of the similar studies that have employed a quantitative research approach include Rico (2008),

Nair et al. (2010), and Kanellopoulos et al. (2010). This design was applied on existing data collected from a U.S. based technology manufacturing company's software development and maintenance organization.

This time series quasi-experimental study allowed me to explain the impact of the Agile model on software maintainability characteristics—analyzability, changeability, stability, and testability—measured while controlling for software development team members' participation in an Agile-driven project, software project management leadership stability, and technology platform. The meta-analysis study conducted by Dyba and Dingsoyr (2008) underscored the need of quantitative research applied to real-life Agile software development cases. Their review however, concluded with very few specific studies (Ilieva et al., 2004; Layman et al., 2004; Wellington et al., 2003) that compared the software quality attributes, software defects, and external and internal quality measures between Agile-driven software systems and non-Agile software.

Furthermore, several researchers have used quantitative models employing both cross-sectional and time-series designs to understand and evaluate the suitability and validity of maintainability related measures within a software system developed using ASDM. These researchers include Olague et al. (2006), Abrahamsson and Koskela (2004), and Souleles (1999). As noted previously, this study involved an examination of the implications of ASDM on maintainability characteristic of software: analyzability, changeability, stability, and testability using the software's source code measures as they reflect internal design quality. Earlier researchers (Kafura & Reedy, 1987; Lewis &

Henry, 1989) found a link between the use of software quality metrics and efforts of software maintenance and that software quality metrics were found to be valuable indicators in the quantitative assessment of software maintainability. Recently, Kozlov et al. (2008), in correlation analysis study, reported that knowledge about the relationships between internal software quality-related attributes and software maintainability can be used as a foundation to address and improve software maintainability during earlier stages of the software development process.

As indicated in the literature review summary, this study would fill in the gap where the impact of ASDM on software analyzability, changeability, stability, and testability–four key maintainability characteristics representing quality indicators of the software–have not been examined with empirical data and with applicable internal quality measures. These internal quality characteristics in turn have direct influence on the software maintenance efforts, associated cost, and overall IT effectiveness. To do so, the following key research question was addressed: To what extent, if any, does ASDM impact the software analyzability, changeability, stability, and testability characteristics? The software analyzability, changeability, stability, and testability of the software are particularly important to software maintainability aspect within ASDM-driven projects in which the entire lifecycle of the development project can be considered maintenance (Beck, 2002).

This study utilized the following key research elements based on this discussion:

1. Concept: Measuring the impact of ASDM operationalized using TDD, REFR, and CI on software maintainability.
2. Dependent variables: Software analyzability, software changeability, software stability, and software testability. Additionally, software maintainability was measured as sum of weighted average of these subcharacteristics that constitute it.
3. Independent variables: ASDM is characterized by three key variables: TDD, measured in percentage of test code created in each iteration; Refactoring, measured in percentage of classes with Cyclomatic complexity higher than 21; and CI or continuous integration is a count of a successful build during each development iteration. Each iteration is typically a week-long development cycle in the XP programming approach.
4. Control variables: Agile software development team size, experience of the Agile programmers within the Agile development team, and software iterations developed using at least 10 or more Agile practices as identified in Table A1 in appendix A, serves as control variables. Being that this is a post facto analysis based on existing data, these confounding variables also served as software system selection criteria as tabulated in Table 7. It was ensured that only software system data that has same development team size, experience, and adherence to 10 or more agile practices throughout the development project was included in the data analysis.

5. Unit of analysis: This study analyzed selected software source code attributes such as complexity, coupling, unit testing efforts, and duplication of the software system developed using object oriented programming language: C#. The source code attributes are also tabulated in Table 13.

Justification for Research Design

The primary purpose of research conducted by Olague et al. (2006) was to examine software stability subcharacteristics of software maintainability, high level characteristics in software developed using a highly iterative or agile process, but with a core focus on the utility of information theory-based metrics to measure stability. Besides the software stability measurement, this study additionally measured three code quality indicators: software analyzability, changeability, and testability using the ISO/IEC 9126 standard and the similar methodology used by Kanellopoulos (2010). Quantitative research design is appropriate because the source data of this study will be the source code properties for the maintainability assessment of the software systems designed using the Agile model within U.S.-based organization. The internal data about software systems from single technology manufacturing organization is a valid data source because it offered the same group of software systems that were touched consistently by the same ASDM developers or resources, tools, and seasoned business processes followed in Agile projects. Furthermore, it allows for the stream of measurement of changes in software analyzability (SA), changeability (SC), stability (SS), and testability (ST), the four main software maintainability related

subcharacteristics that are dependent variables in this study. ASDM was also measured in terms of its key characteristics as TDD, Refactoring, and CI.

As this available data is post facto in nature, the ability to test the impact of ASDM on incremental changes in software SA, SC, SS, and ST was an involved analysis. This difficulty arose because incremental change analysis over series of time interval would require multiple maintainability measurement snapshots to determine whether ASDM variables (TDD, Refactoring, and CI) have impacted software SA, SC, SS, and ST across the Agile evolution. A longitudinal design, on the other hand, may suffer from participant attrition (Ahern & Brocque, 2005; Creswell, 2002), therefore obscuring the true intended ASDM impact. This study analyzed Agile iterations during the specific period of time chartered by a well-governed project.

Despite the shortcomings of a quasi-experimental, time series design that is characterized with time series measurement after every weekly iteration, this method provided the most appropriate design approach to measure the impact of ASDM-related variables on changes in software SA, SC, SS, and ST. These changes were measured by several sets of known metrics and by conducting regression analysis against all the values collected at every iteration of ASDM driven software. To reiterate, Agile development projects integrate software maintenance into the subsequent iterations (Beck, 2000), and all the iterations after the initial release essentially fall into maintenance stage of the development cycle (Stafford, 2003).

Finally, this quasi-experimental with case study data design is warranted because it is the most commonly used when investigating the impact of an analysis-related question with the quantitative nature of data being collected. As revealed in the literature review, many prior studies that have employed quantitative design approach being the actual data is ordinal measured value of source code properties for each maintainability subcharacteristic. This study, on the contrary, used actual discrete data about the source code as a building block of the measurement. In addition, a unique feature of the data related to the source code attributes in this study is that it allowed to integrate the software application expert's additional subjectivity using AHP in a quantitative manner.

Case Data and Selection Procedures

A software system was selected for this study with purposive selection approach from the specific IT organization's software applications drawn from the 2009 to 2011 source code repository. The rationale for the selection of the data from the same organization was that these data sets aligned well with the research goal in terms of a stable resource team that worked on these Agile projects, and well controlled ASDM practice adherence, and availability of source code for analysis. This source code data set for selected software application derived from internal business needs of this organization, administered by the internal IT organization's software development and operation group.

The information gathered from this purposive selection of data sets is intended to be representative of the other organizations that adopted ASDM and matured over the last

5 years in similar settings in various other IT organizations. To ensure this, the study collected data based on specific software development groups within an IT organization that adopted ASDM around 2005, trained internally its software development team on Agile methodologies, as well as continually maintained the Agile-consistent business processes to manage the development projects during these times. The source code repository was accessed for the information of the selected software application developed from 2009 to 2011. Although most other studies have focused on the analysis of single-version assessment or the release of software application at the end of the project, the research question in this study directed the focus on empirical impact assessment of ASDM-driven software application, which is characterized with multiple versions created through Agile development approach.

This software system was selected for its availability of data from number of iterative revisions, with consistent adherence to ASDM models, both from technical (XP), and project management (Scrum) perspective. System A was implemented with Microsoft .NET technology, programmed in C# (C Sharp) to be able to qualify as valid case data. System A-related version changes after every iteration through all underlying iteration cycles were analyzed. The development of System A also met all of the 12 Agile principles augmented by Agile Alliance supplemented by matured and experienced Agile development team. Their alignment with ASDM and relative data was tabulated below showing number of Agile developers, their experience with Agile and overall s/w development, and Agile practices followed during the development.

Table 7

Software System Selection Criteria

Key Criteria or Control Variables	Software system specific qualifying data
Development Paradigm	Object Oriented
Total Developers	>2
ASDM Experience in Years	>5
ASDM practices followed	10-12

The selected system continuously delivered revisions with changing requirements generated from well connected internal customers, released and deployed using automated tools, unit testing, TDD, refactoring, regression testing, pair programming, technologically well connected teams within the United States. Physically, the co-location of developers, customers, and other resources within project were not followed for practical reasons for this project, but the IT technologies such as live-meeting, teleconferencing, and desktop sharing compensated for its need.

Table 8

Software System: Sample Data Collection Sheet Example

Software System - A Data Set						
Version	1.1	1.2	1.3	1.4	1.5	...1.60
Cyclomatic Complexity						
Coupling between objects						
Code Duplication/Cloning						
Unit Size						
Assert count/Class						
Unit Testing						

Case Data Set Details

Software system related enhancements should also mainly be triggered by user requirements, software defects, and new defects resulting from code integration through all the iteration. Unit testing, user acceptance testing, and regression testing with heavy reliance on continuous user feedback are performed on System A versions prior to production release. The change log for each version should be available along with post release reported incidents tracked and guided by ITIL operation framework; however, this data will not be included in the analysis. Note that the research question in this study is guided by ISO/IEC 9126 model's *internal quality view* and source code assessment is

at the core of the research method to attain the objective of this study. In addition to actual meta-data collection from the source code repository, the ASDM expert's subjective judgment was elicited using AHP process. Appendices A, B, and C simplify this data collection process.

On the independent variable or ASDM side, three key variables were operationalized as discussed below. TDD has been widely examined within the Agile community. Within industrial settings, researchers found improved test coverage (Bhat & Nagappan, 2006), reduced fault rates (Damn et al., 2005), and faster defect fixing (Lui & Chan, 2004). There is no study currently found that relates or attempts to examine its impact on software maintainability characteristics.

Secondly, the effect of refactoring on code quality itself has widely been studied (Khomh, Penta, & Gueheneuc, 2009; Shrivastava & Shrivastava, 2008); however, there exists a single study conducted by Wendorff (2001) that assessed the impact of refactoring on software maintainability. With this given void in research, this study included TDD as one of the key independent variables that signify the ASDM model. The complex classes are typically worked on during refactoring efforts developers take in all iteration, so the direct change in the complexity of the complex classes are evident in each iteration. Percentage of classes with cyclomatic complexity higher than 20 were measured for complex classes to measure refactoring efforts. The log entries made by the developers within the source code repository were not used as a measure because it may

be the case that developers actually did the refactoring but failed to add an entry in the log.

Lastly, CI is also one of the selected independent variables in this study. CI strengthens the confidence of Agile teams in their code during the development phase on daily basis as they see their code succeeding the integration testing. As the development process progresses, the Agile team builds and integrate its code using automated tools and assesses the initial quality multiple times throughout the iteration cycles. Compilation and unit testing is part of this essential technique that can be operationalized by capturing the successful CI instances throughout the iteration. This study used simple measure of count of number of successful builds or integration per iteration cycle that was mined through the log history of the CI tool.

Confidentiality

The data analyzed in this study essentially is meta-data or set of internal software quality metrics yielding the numerical numbers related to dependent variable: maintainability and its four subcharacteristics also measured separately of the actual data, that is source code of the software system developed using ASDM. The confidentiality of this source code data of software system relied on the source code control related procedures and security controls that are in place and managed by the organization's IT department with clear accountable resources during the collection and analysis process.

Permission to use the identified IT organization and ASDM experts was given by the selected organizational management. All data was presented in a way that prevents

determination of origin. This included all documented evidence and AHP scoring responses. An AHP scoring process introduction was presented to an ASDM expert describing the purpose and assurance of confidentiality of their scoring of the system. The participant was also given an opportunity to decline the scoring. The subject IT organization was given an opportunity to review the final dissertation resulting from this study before it is made public to prevent loss of intellectual property (IP) or disclosure of proprietary information. Due to IP considerations, only summarized and sampled data is published in this dissertation document. Researcher may be contacted for complete set of data, calculations, and complete statistical analysis results.

In addition to relying on the existing data confidentiality, the data collection process was first presented to the Walden Internal Review Board (IRB) for approval. The full IRB approval was received with approval number 10-24-11-0113109. A literature support was included in the study to provide the IRB with a rationale for use of the software system data from technology manufacturing organization.

Instrumentation and Materials

For instrumentation, this study used an ISO/IEC 9126 software quality model to explicate software maintainability and its four sub-characteristics to evaluate the impact to software maintainability based on source code quality and static behavior. The study further used source code attributes and integrated AHP within evaluation process to enrich the data analysis quality. AHP uses the software expert knowledge by allowing weights assignment to software maintainability sub-characteristics, source code

attributes, and measures. The ISO/IEC 9126 model has been widely used by researchers, including Heitlager (2007), Chen and Huang (2009), and Kanellopoulos et al. (2010) to operationalize software maintainability and its sub-characteristics. Kanellopoulos et al. (2010) further leveraged ISO/IEC 9126 standard, enhancing it pragmatically with AHP, a decision-making technique that distills complex multi-criteria decisions to one to one comparisons (Saaty,1980). This study partly integrated Kanellopoulos et al.'s approach to assess impact on software maintainability and its sub-characteristics: analyzability, changeability, stability, and testability, and therefore examine the relationship between the independent or treatment variable (ASDM) and the dependent variable (software maintainability) as discussed in chapter 2.

Other variant models, such as the Evaluation Method for Internal Software Quality or EMISQ by Plosh et al. (2007), Fuzzy AHP (FAHP) by Liang and Lien (2007), and the decision making model using AHP to enable quantification of quality attributes for varied architecture pattern used by Svanhberg and Wohlin (2005) were reviewed in the literature related to methodology. Further, the IEEE 1219 standard model by Broy et al. (2007), maintainability index (MI) formula based on hierarchical structure by Oman et al. (1994), clustering with K-Means and Neural Gas algorithms by Zhong et al. (2004), and K-Attractors clustering algorithm with ISO/IEC 9126 model by Kanellopoulos et al. (2008) have been utilized by other researchers.

Jung, Kim, and Chung (2004) argued that the structure of ISO/IEC 9126 software standard is a multidimensional concept, when they surveyed the user's satisfaction related

to software quality using ISO/IEC sub characteristics. Their single case study revealed validity of the structure with some ambiguities for few sub-characteristics such as replacability, installability, testability, and coexistence. Al-Kilidar, Cox, and Kitchenham (2005) evaluated the quality of outputs of the design process using ISO/IEC 9126 and reported that this standard does not offer any procedural guidelines for aggregating the metrics for overall quality evaluation. This void was well recognized in this study, and hence, it integrated the widely used and applicable measures used by the researchers highlighted in the literature review.

The ISO/IEC 9126 model is being used widely; however, it does lack on providing practical guidance to apply the right measures in the realistic evaluation of internal quality of software. Lee and Lee (2005) evaluated integration of AHP in their case in the light of this given limitation of ISO/IEC 9126 in actual methodical measurement of identified software quality characteristics. The pair-wise comparison does increases significantly, and hence, the AHP method's up-scalability is limited with more number of alternatives or system properties, as argued by Karlsson and Wohlin (1998). They also further contended that the amount of redundancy in the pair-wise comparisons builds this method's insensitivity to errors in the expert's judgment. The resultant weights are relative and essentially ration scale facilitating elicitation of weights. Correia, Kanellopous, and Visser (2009) studied the mapping between software properties and quality characteristics using AHP towards the refinement of mapping using relative weights derived using AHP. This study integrated the same approach as a

part of the data analysis method and uses the internal software experts including developers, Agile practitioners, and XP developers.

Lastly, the latest systematic review on software maintainability metrics conducted by Riaz et al. (2009) reported that the most frequently used predictors were based on size, complexity, and coupling. Furthermore, these predictors were used at the source code level, and this finding is well congruent with internal quality metrics view of ISO/IEC 9126 model of software quality. This software quality model observed that identified internal attributes of the software are a pre-requisite for achieving the desired external behavior. Furthermore external behavior of the software system is a pre-requisite for achieving quality in use.

This study utilized the internal quality notion of ISO/IEC 9126 model. The research question in this study also directed the integration of the analysis of source code properties using the same predictors as well as additional ones that includes duplication, unit size, and unit testing used in the latest study conducted by Kanellopoulos (2010). Additionally, Riaz et al.'s (2009) systematic review results revealed that the most commonly used maintainability measure used ordinal scale that was based on expert judgment. Analytic hierarchy process in this study further captured the ASDM and maintainability expert's subjective judgment to alloy with quantitative measurements of software maintainability that includes software analyzability, changeability, stability, and testability.

Thus, the ISO/IEC 9126 quality model integrated with AHP similar to the integration in the study conducted by Kanellopoulos et al. (2010) is appropriate for this study because its internal quality notion clearly aligns with intent of assessing the impact on software maintainability characteristics. Additionally, the ISO/IEC 9126 standard is structured hierarchically outlining quality characteristics and sub-characteristics providing focused frame of reference on each main branch of quality characteristics, such as maintainability with ease in mapping source code measures to each sub-characteristics. This quality model uses the standard terminologies being an international standard, and hence, is a choice of researcher community as well as associated practitioners within IT organizations. Other similar hierarchical quality models such as Boehm's (1978) model focused on effort on software maintenance cost effectiveness whereas McCall's (1977) model focused on the precise measurement of the high-level quality characteristics and attempts to define how easily, reliably, and efficiently can one use software as is. McCall's hierarchical model attempted to group the quality perspectives in set of quality factors, criteria, and measures or metrics. The ISO/IEC 9126 model offers a hierarchical view of software characteristics without clear guidance on how to measure the enlisted sub-characteristics, but offers three distinct quality notions described in Figure 11. Additionally, the magnitude and direction of change from independent variables can be easily identifiable in this simplistic model.

In summary, the ISO/IEC 9126 software quality model built the clarity about dependent variables and AHP model enables the researcher to enrich the quality of the

analysis allowing quantification of the subjectivity assigned by the Agile experts in practical settings of this quasi experimental study.

Data Collection and Analysis

Upon approval from Walden University's IRB, data was collected from the IT department of a U.S.-based semiconductor manufacturing organization by downloading the source code of software system – A, directly from the source code repository using Tortoise SVN tool on a secured workstation managed within the intranet of the IT organization. The selected code metrics data was retrieved using Source Monitor and Understand tools. The data was then organized and manipulated for analysis and interpretation using AHP model of weight elicitation, statistical analysis tool–MiniTab version 15, SPSS version 19, and Excel 2007 programs. A data mapping containing a list of variables, source code attributes, and code metrics was created using the Excel 2007 program. This data-mapping table served as the basis for the case data set for software system under study. The maintainability related sub-characteristics were regressed for ASDM or X variables based on the collected data to answer the research hypotheses of this study.

Analytical Model of Theoretical Framework

The ISO/IEC 9126 quality model integrated with AHP (Kanellopoulos et al., 2010) within the framework of Lehman & Belady's (1976) software maintenance and evolution theory as summarized and studied (Sindhgatta, 2010) in Chapter 2 built the foundation for this study.

Ping (2010) posited that the software evolution is inseparable from software maintainability attribute. The ASDM-driven software life cycle complies with seven Lehman and Belady's laws of software evolution, as reported by Sindhgatta et al. (2010) from their software project case based on the Scrum method. Few earlier studies (GodFrey & Tu, 2000; Xie et al., 2009), however, were based on open source software system and related data. Furthermore, although Xie et al.'s study was partly inconclusive on four (of eight laws), they found that Continuing Change Hypothesis (CCH), Increasing Complexity Hypothesis (ICH), Self Regulation Hypothesis (SRH), and Continuing Growth Hypothesis (CGH) were valid for open source software evolution. Capiluppi et al. (2007) examined ICH's applicability within the XP-driven software evolution. As reported in the literature review, there exist scant studies related to Agile-driven software evolution, with an exception of recent work from Sindhgatta et al. (2010), which was limited to an empirical examination of Lehman's law within ASDM. This study extended further and employed analytical model shown in Figure 12 based on Lehman and Belady's software evolution laws within ASDM context impacting software maintainability characteristics and its four core constituents: SA, SC, SS, and ST.

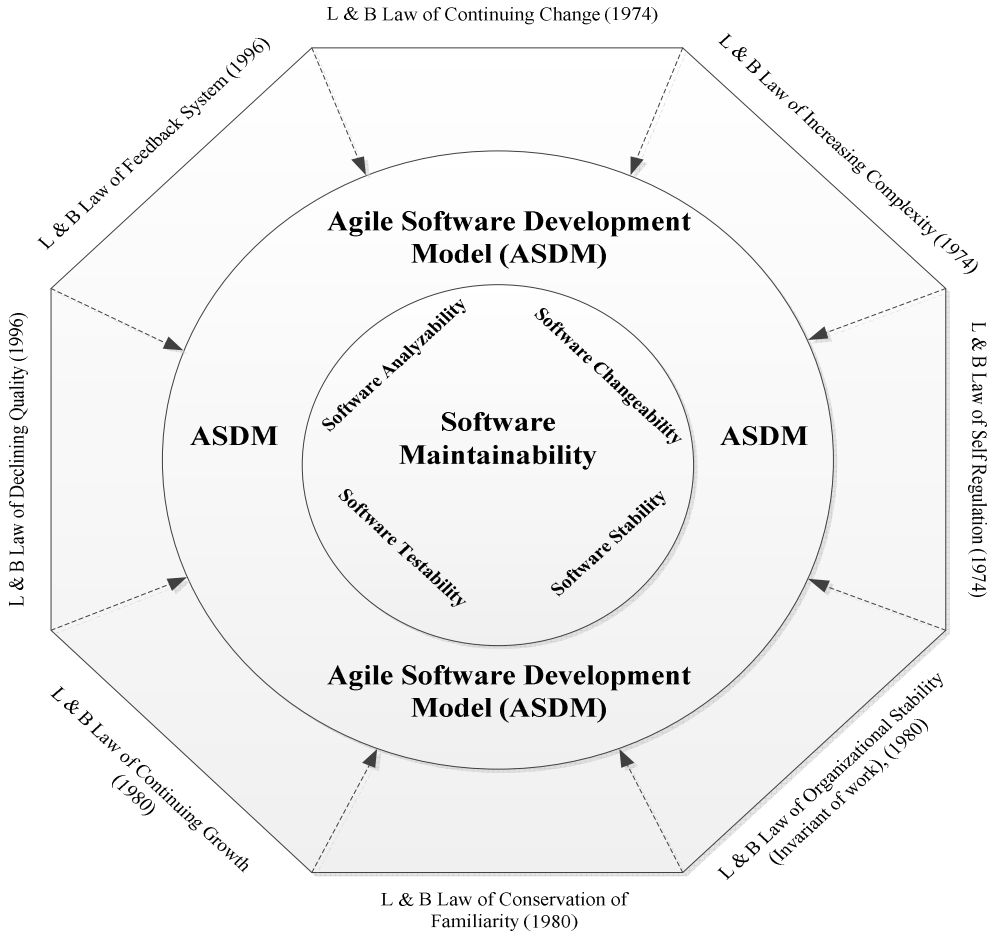
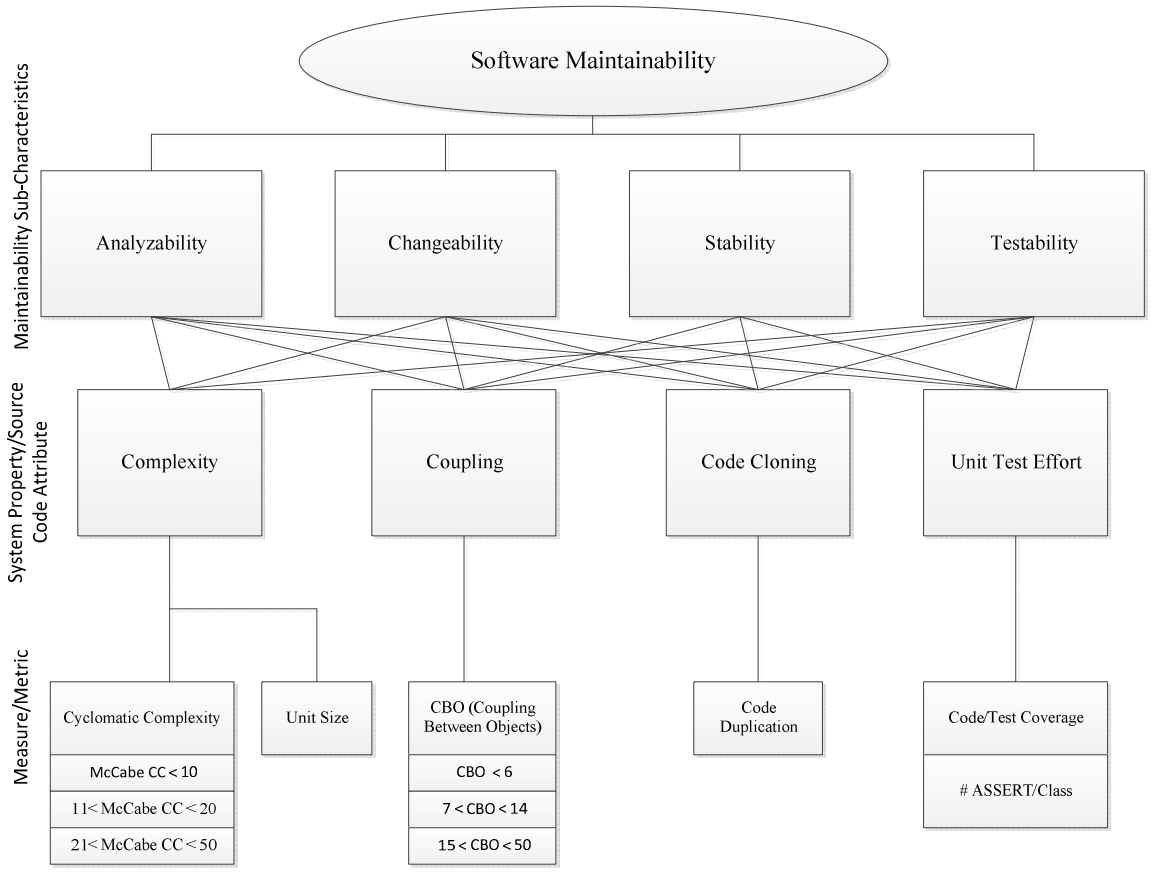


Figure 13. Theoretical model for this study

This study's objective was to investigate the software maintainability during the ASDM-driven software development project at every iteration and production release cycle, which is also grounded in the supposition of Lehman's Laws of software evolution theory as depicted in above figure of the theoretical model. Furthermore, based on the ISO/IEC 9126 software quality standard, the maintainability sub-characteristics were

decomposed further as shown in Figure 14 with indentified source code attributes and measures based on the literature review discussed earlier.



Software Maintainability Hierarchy from ISO/IEC -9126 Software Quality Standard integrated with source code metrics used in this study

Figure 14. Software maintainability model with source code metrics used in this study

The selected set of metrics was used in various previous studies and many researchers for object oriented development approach as well as within Agile-driven software development projects. The link and map diagram (Figure 14) shown guided this section of the analysis.

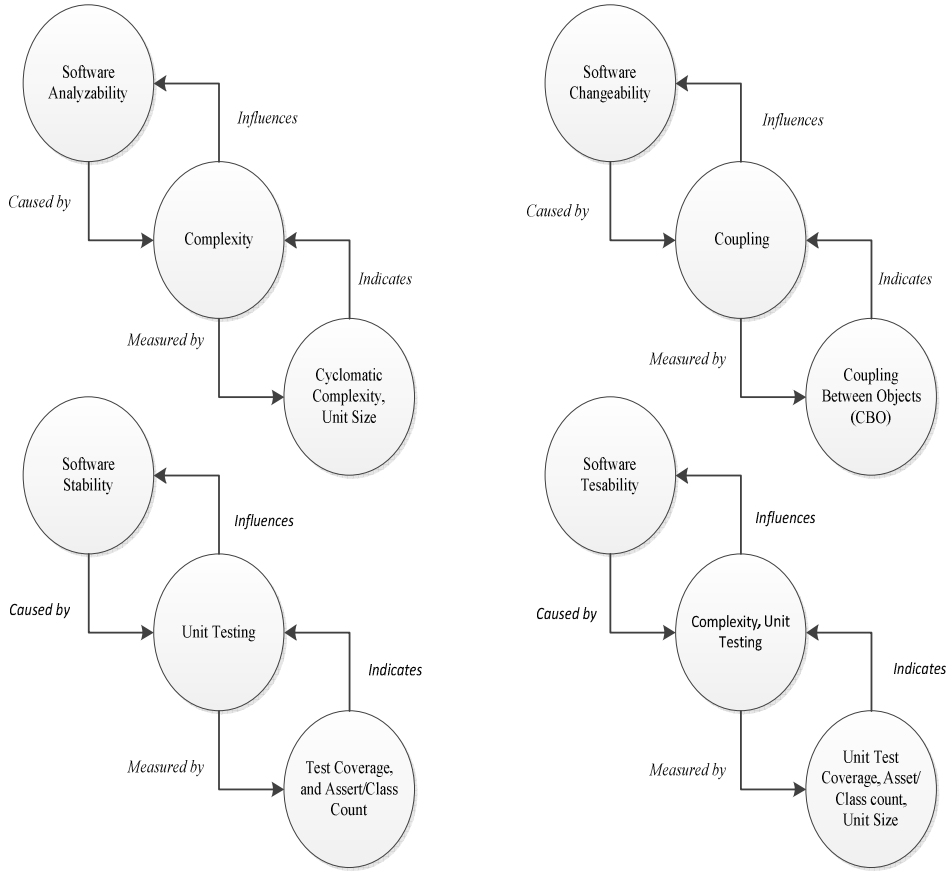


Figure 15. Link and map diagram explicating software quality attributes, code properties, and source code measure.

This study employed the metrics for selected source code properties based on the literature review discussed earlier with their purpose and operation definition as tabulated in Table 9. The next section outlines the data analysis steps.

Table 9

Software Properties or Source Code Attribute with Applicable Measures and Definition

Attribute	Metric	What it measures
Unit Size	Lines of code per unit	Unit Size (of a method)
Complexity	Cyclomatic Complexity (CC)	Level of complexity of the design & coding structure
Coupling	Coupling Between Objects (CBO)	Count of the number of classes to which a class is coupled.
Duplication	Code Duplication or Cloning (D)	Degree of source code duplication
Unit Testing	Test Coverage and Count of 'Asserts' per class file	Small test program to test the code and depth of the testing

Attribute	Operationalization
Unit Size	Number of lines of code/statements in a method
Complexity	CC (McCabe Metric)= number of linearly independent paths through a source code (method)
Coupling	Count of the number of classes to which a class is coupled (CBO)
Duplication	The % of all code that occurs more than once in equal code blocks of at least 6 lines
Unit Testing	Percent of Test/code coverage and count of Assert within class files measured at the end of every iteration

Data Analysis Steps

Enlisting software attributes for dependent variables or software maintainability sub-characteristics: This step begins with clearly listing the applicable measures for each dependent variable as shown in Table 10. Note that these measures are retrieved based on the literature review conducted in chapter 2. The specific studies that used these measures are Kanellopolious (2008), Kanellopolious (2010), Heitlager (2007), and a systematic review conducted by Riaz et al. (2009).

Table 10

Maintainability - Dependent Variables and Their Measures

Software Analyzability	Software Changeability	Software Stability	Software Testability
Complexity measured in McCabe CC	Complexity measured in McCabe CC	Complexity measured in McCabe CC	Complexity measured in McCabe CC
Coupling measured in CBO	Coupling measured in CBO	Coupling measured in CBO	Coupling measured in CBO

Table 10 *(Continued)*.

Software Analyzability	Software Changeability	Software Stability	Software Testability
Duplication measured in cloning percentage	Duplication measured in cloning percentage	Duplication measured in cloning percentage	Duplication measured in cloning percentage
Unit Test Efforts measured in test coverage percentage and Assert counts per Class	Unit Test Efforts measured in test coverage percentage and Assert counts per Class	Unit Test Efforts measured in test coverage percentage and Assert counts per Class	Unit Test Efforts measured in test coverage percentage and Assert counts per Class

The discrete values measured using metrics shown in Figure 14, are multiplied by the weights derived using AHP process, for each software attribute to compute value for each software attribute as outlined in equations A and B.

Table 11

ASDM: Independent Variables and Their Measures

Variable >	Test-Driven Development	Refactoring	Continuous Integration (CI)
Operational measure	Percentage of Test Classes (Test Class Count/Total Classes)	Cyclomatic Complexity (1 /Count of Classes with Cyclomatic Complexity > 21)	Total Count of successful builds
Scale/Data Type	Ratio/Discrete	Ratio/Discrete	Ratio/Discrete

Note that there is more than one measure for some variables such as complexity and unit test efforts, inherited from the specific literature discussed earlier related to these source code attributes.

As shown in Figure 13, the data collection focused on maintainability aspect of a software system quality as defined by ISO/IEC 9126 model with sub-characteristics defined as:

Analyzability: how easy or difficult is it to understand where in the system, specific changes need to be made?

Changeability – how easy or difficult is to make actual change?

Stability – how easy or difficult is to maintain the software in stable and consistent state, after making a change?

Testability – how easy or difficult is to determine whether change made has been implemented correctly?

Software maintainability compliance sub-characteristics listed under software maintainability hierarchy of ISO 9126 is not included in this study since the compliance measurement is not applicable within the scope of this research question. This study is rather focused on the assessment of the impact on maintainability itself and not on the compliance aspect of maintainability.

AHP, one of the many multi-criteria decision processes, allowed weight elicitation from the ASDM experts in selected IT organization of identified technology manufacturing corporation, but those that did not work on the software system chosen under this study. This is to prevent any bias towards scoring the system attributes. The outline of this process is explained below with details in addition to Appendix B.

Analytic Hierarchy Process (AHP) for Weights Assignment

The AHP was employed for the weights assignment in this study. The process often begins with distillation of complex decisions into a series of one to one comparisons. Originally proposed by Saaty (1980), this technique integrates both

qualitative and quantitative dimensions of decisions. Saaty characterized AHP as a method of prioritization. Kanellopolious (2008) asserted that AHP systematically compares a list of objectives and within systems engineering discipline; it serves to evaluate alternative engineering design concepts through comparison as a powerful tool. The best alternatives are chosen with supportive rationale at the end. AHP has previously been used in Kanellopolious's (2008) scholarly study that was related to a data mining technique in the software maintenance domain. Alternatively, Liang and Lien (2007), when selecting optimal ERP software integrating the ISO 9126 software quality standard, also used Fuzzy AHP, another variant of AHP. Miranda (2001) compared ad hoc estimation with paired comparison approach and found higher-level accuracy and precision with paired-comparison method. As standardized in the ISO 9126 software quality model, the software maintainability sub-characteristics–analyzability, changeability, stability, and testability–are related to software maintainability stem or the objective itself. The AHP process is a natural fit for assessing the research question of this study for which qualitative and quantitative aspects of decisions needed to be analyzed.

When comparing the alternatives with set objectives, AHP generates a pair wise comparison matrix. The relative importance of objective O(i) as compared with objective O(j), is expressed in a number, and it is placed in the i th row and j th column. These numbers or values are picked from 1-5 scale using below norms with:

a(i j) is assigned value 1, if the two objectives are equal in their importance;

a (i,j) is assigned value 2, if O(i) is weakly more important than O(j);

a(i,j) is assigned value 3, if O(i) is strongly more important than O(j);

a(i,j) is assigned value 4, if O(i) is very strongly more important than O(j); and

a(i,j) is assigned value 5, if O(i) is absolutely more important than O(j).

Once this value assignment step is complete, this comparison matrix was normalized and eigenvalues were calculated. When evaluating the alternatives for the objectives under study, these eigenvalues represent coefficients or weights.

In this study, AHP was applied only on the 2nd level of the hierarchy shown in Equation B. At the first level, the source code properties (Complexity, Coupling, Duplication, and Unit Testing efforts) were evaluated using the selected metrics. AHP was applied to the 2nd level in the hierarchy where software maintainability sub-characteristics viz., analyzability, changeability, stability, and testability, were assessed using source code properties and their weights. Lastly, software maintainability was evaluated from all four listed sub-characteristics, but with simple addition of weighted average of all corresponding Y variables or sub-characteristics: changeability, analyzability, stability, and testability. AHP was not be applied at the 1st level because maintainability is simply a function of all sub-characteristics and the sum of their weighted average was used to yield final Y value.

In sum, the following equations were used to calculate the values for each sub-characteristic shown in the maintainability model using ISO/IEC-9126 quality characteristics hierarchy.

$V(SCi) = V(D1)*W(D1i) + V(D2)*W(D2i) +...+ V(Dn)*w(Dni)$ **Equation A**

And

$V(Di) = V(M1) + V(M2) +...+ V(Mn)$ …… … **Equation B**

V(Di) = Value of Source Code Property Di

W(Dji) = Weight of Source Code Property Dji for Sub -Characteristic i

V(Mi) = Value of Metric Mi derived for the source code using software tools

(D represents Source code property, and M represents Metrics)

In the subsequent section, a process detail of deriving the number of ASDM experts is discussed. To elicit the weights for source code attribute such as coupling, complexity, duplication, and unit test efforts, an expert Agile developer or developers following ASDM within selected technology manufacturing organization, were asked to provide their experience-based judgment. An ASDM expert was selected purposively from the same technology manufacturing organization from where the software system related source code properties were retrieved. An overview of ISO/IEC 9126 maintainability characteristics and their definitions were provided to the selected experts to ensure their familiarity with it. Next, the actual data collection pertaining to weight elicitation relied on the input collected from them. The ASDM experts essentially filled in their comparative score using scale of 1-5 to evaluate the importance of the source code attributes for each sub-characteristic. The scored data is shown in Appendix B.

Miranda (2001) suggested that pair-wise comparison requires the selection of qualified experts and a tool for automating the underlying calculations. He also further

cautioned that the number of experts used to evaluate the entities (n) should not exceed entities the number generated by (n) / 3. With a higher number of experts, the opportunity to perform multiple comparisons diminishes. This study incorporated Miranda's (2001) recommendation of allocating comparisons to experts in which every other comparison will be assigned to a different expert in the sequential order as needed. Table 12 shows the simple derivation of the number of ASDM experts needed of weight elicitation. Again, in this study the AHP was applied on 2nd level only. The Excel-based AHP template tool was used to automate the calculations based on the weights assignment by the ASDM experts as documented in Appendix B.

Table 12

Experts' Weight Elicitation in AHP on System Property or Attribute Level

Entities	Number of entities *(n)*	Number of experts *(n /3)*
Complexity Coupling Duplication Unit Test Efforts	*n = 4*	4 / 3 = Rounded to 1

Once the weight assignment was completed for the system code property (D) for each sub-characteristics (SC) as outlined above; software maintainability, analyzability, stability, and testability were calculated based on the actual values collected in template of Table 13. Lastly, to validate the research hypotheses, regression analysis was

conducted on the series of derived values at each subsequent release, for all four sub-characteristics or dependent variables, and the resultant maintainability that is simply a sum of weighted average of all four sub-characteristics or Y1, Y2, Y3, and Y4 variables.

Repeated measure design is considered statistically powerful because it controls for potential sources of variability (Aczel, 2006). There exists a small chance of residual error or experimental error beyond the researcher's control. Regression analysis within subjects design involves repeated measures on the same subjects with multiple observations overtime. Regression analysis was conducted using the MiniTab tool for (ASDM) variables–X1 (TDD), X2 (Refactoring), and X3 (CI) and four maintainability related variables. One-way repeated measure ANOVA technique and paired-samples T-test may be an alternate approach; however, it does not establish concrete causal link details between independent and dependent variables. Because there are repeated measurements taken at every software release during an ASDM development project, and because the intent of the study is to assess the impact of ASDM model characterized by key variables listed earlier, this study used multiple regression analysis run against the maintainability sub-characteristics or dependent variables. It also assessed the impact on final weighted maintainability, which is a function of all Y variables or sub-characteristics. This regression results confirmed with individual regressions conducted for Y1 through Y4 variables.

Data Collection and Analytical Method Validation

Validation of the data collection and analytical method are important steps because discrepancies in interpretation may occur, resulting in type I or type II errors. However, in this study, the data set, that is, the actual source code was generated as a result of software systems being developed using ASDM based on selected technology manufacturing organization's real life business needs. Additionally, the data set corresponding to selected software system for this study was based on the key selection norms such as development paradigm, developers' experience with Agile, and ASDM practices followed during the projects. This data was collected from the software system details and history managed using internal software tool within the technology manufacturing organization managed by the project management team.

The data analysis method employed the ISO/IEC 9126 quality standard to define the variables and AHP process for weight elicitation. The AHP process is being used in the software engineering discipline as well as in the IT and software industries, as cited in the literature review highlighting relevant researchers' empirical work. From a conceptual and analytical framework, an analytical model is crucial to measure the software maintainability during the ASDM or independent variables in action. In other words, it is critical to measure the software maintainability computed along with software analyzability, software changeability, software stability, and software testability during the actual software development work beginning with iteration-one, for existing software systems that evolve through ASDM. The maintainability was then subsequently

measured through all release cycles until the final release. However, there are certain steps that were conducted to measure the impact of ASDM on software maintainability accurately and to ensure that the results were reliable and valid.

First, the data was filtered from only the software system that was developed and maintained by trained and experienced Agile developers and Agile-driven project management practitioners with development approach adhering to key ASDM characteristics as outlined in the Agile manifesto. These software systems are developed using XP and or Scrum methodologies, that is, software systems developed using the non-ASDM approach and unsuccessful development projects will be excluded. These filtered software systems with their evolutionary source code-related data sets were then standardized using the SPSS tool because the unit of measurement for independent variables was not the same. The standardized Z score was created for all independent variables, as shown in appendix I.

Further, software systems that were maintained or updated based on infrastructural compliance requirements and platform architectural requirements, and essentially were not touched by programmers–including small size emergency fixes with the duration–less than two iterations were eliminated from the final analytical data. This filtering logic to examine the research question of assessing the impact on software maintainability related characteristics within controlled and well-seasoned organizations adapting ASDM was also driven by the stronger need for accurate assessment.

This study did not include any open source developed system data, experimentally developed software system in unreal life settings, or non-proprietary software system related data due to inherent limitations of such a data set being prone to the impact of other uncontrolled variables, subsequently leading to inaccurate findings. More importantly, the study included single software system that operate in or address a problem or activity of real world or E-type software systems (Lehman & Belady, 1976) to comply with the core analytical model of this study. As noted, Sindhgatta et al. (2010) reported that most of Lehman's laws hold true for the ASDM-driven software development project in their case study of the E-system.

Second, the source code was the only entity that was analyzed. No post deployment related defects data were part of the scope of this study. This focus is mainly driven by the scope of the research question as well as the fact that it would remove any invalid results related to software maintainability triggered and managed outside of ASDM-driven projects. Some of the these trigger points include code changes incurred from other technical initiatives not qualified for the ASDM project, emergency or critical bug fixes handled outside of ASDM teams, resulting configuration changes, database migration efforts, security control integration changes outside of ASDM project work involving either software code changes or no changes. The software system was also filtered for a minimum of 2 years and maximum of 5 years of life cycle age, meaning that no software system older than 5 years and shorter than 2 years of age was part of this study. Agile is just a decade-old movement, therefore it is a practical and wise approach

to examine the software systems that have been developed in relatively stable and mature Agile ecosystems. Additionally, this study did not aim for the measurement of whether the software life cycle age or period of its usage itself influences the software maintainability and its sub-characteristics.

Lastly, the analysis involved software system that had released at least two production versions during the development project. The selected software application was used in the real life or production setting for a minimum of 2 years and was already being developed and or maintained using ASDM. Non-Agile-driven developed or maintained software systems were excluded from this study. This allowed for measurement of whether the ASDM-driven approach has an impact on software maintainability changes or not.

The filtered data thus allowed for a clean data set that was analyzed, comprehending the change or variations in analyzability, changeability, stability, and testability. This analysis helped to answer the question associated with software maintainability and ASDM within the framework of Lehman and Belady's software evolution and maintenance hypotheses, testing the impact of ASDM on software maintainability, which Sindhgatta et al. (2010), did not analyze in their study. Additionally, this analysis also assisted to measure changes in software maintainability within the ASDM-driven software system life cycle using ISO/IEC 9126 software quality model coupled with the AHP decision model that Kanellopoulos et al. (2010) used in their maintainability related study. This Kanellopoulos et al.'s study, however, was not

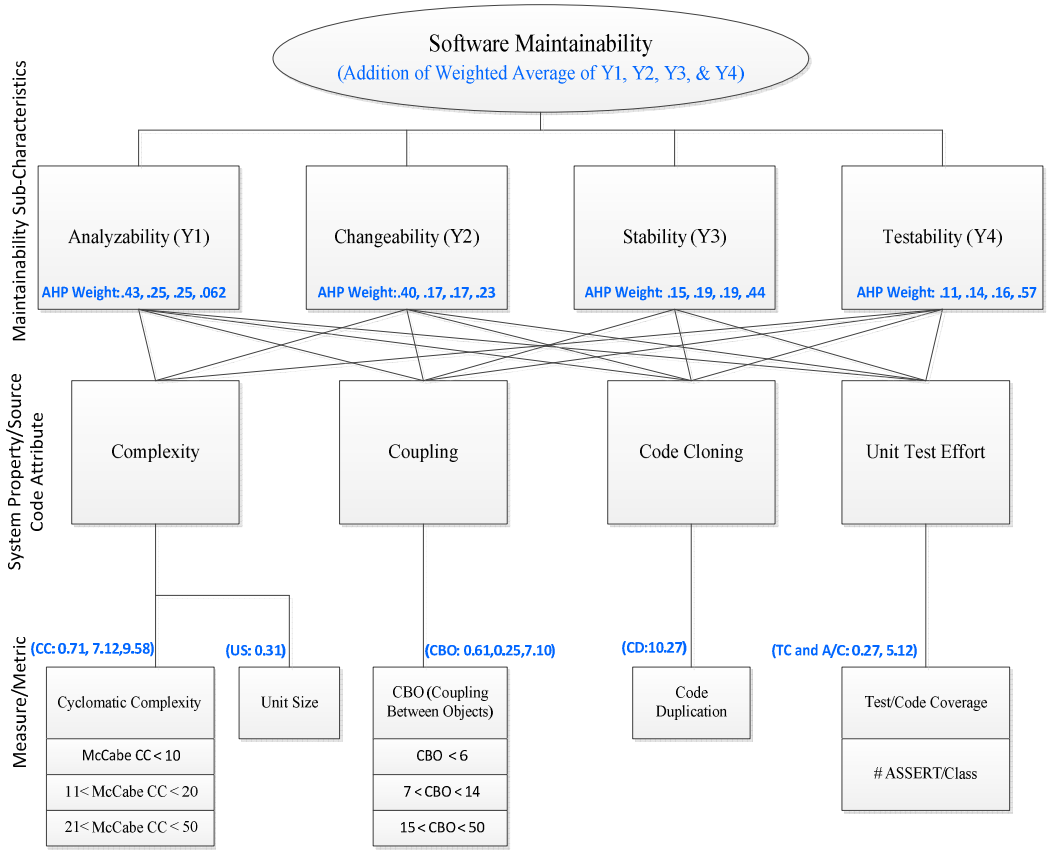
targeted to assess *how* ASDM as a software development approach may impact the software maintainability, including its four sub-characteristics. The analysis in this study precisely addressed this open gap within the Agile driven software development and software maintainability literature guided by Lehman's software evolution theory.

The use of the multiple regression analysis is a prevalent method when the researcher wants to test the impact of a treatment or independent variable through observation in changes in the slope of the regression equation. Within the context of the given research question, in this study:

1. The post facto source codes stored in each revision were analyzed throughout the development cycle by measuring the values for selected metrics and finally values for software maintainability and its sub-characteristics were computed.
2. The source code was analyzed based on the tools listed in Table C6 and also using the source code repository log. ASDM-specific variables were also measured by counting the total class files, test class files for TDD, counting percentage of classes with CC higher than 21 using the Understand tool for the REFR variable, and by counting the number of successful builds for CI variable using the CCNET tool.
3. The selected source code properties were collected after every iteration completion at the end of week within ASDM development project, including all releases, for assessing the impact of ASDM on software maintainability and sub-characteristics. Finally, the values for all the pertaining software attributes shown

in Figure 16 were calculated using the shown measures before running the multiple regression analysis.

The data filtering norms controlled for the non-related data set as well as for confounding variables such as software age, developer's technical ability to develop software within ASDM framework, all of which are argued to impact resulting software maintainability that is a function of analyzability, changeability, stability, and testability, thus ensuring the validity and reliability of the results.



Example of Weights values and actual source code attributes for one specific iteration. These values are then used in equations A and B (page 129/130) to compute the index value for each maintainability sub-characteristics. Final Maintainability will not use AHP weights but simply weighted average of Y1,Y2, Y3, and Y4.

Figure 16. Example of actual source code values and weights for single iteration

Figure 16 shows the example of the values that leads to the actual computation of the index values for dependent variables based on the actual source code metrics and the weights derived using AHP. For example, for software iteration 5, the specific software revision was downloaded from software source repository first. Next, the values of each software attribute were derived using the software tools listed in Table C6 with actual values shown in this figure as an example. Also, the weight for each source code attribute for each subcharacteristics was derived using AHP one time. Final values were calculated for all Y1-Y4 variables using Equations A and B. The maintainability or Y value is the sum of weighted average of these four Y variables. Appendix I shows these calculated values for all Y variables. The next section addresses the reliability and validity aspect of the measure to support the experimental design of this study.

Operationalization and Computation of Y and X Variables

As defined earlier, ISO 9126 provides operational definitions for maintainability and its sub-characteristics based on software's internal quality notion. Object-oriented code-specific metrics that are widely used by the researchers in software engineering domain are selected as M1 through M10. The actual values for these measures were collected using the various software tools listed in the Table C6. Each SA, SC, SS, and ST construct was quantified simply using the collected raw data from the source code during at the end of iteration, and corresponding Weights yielded from AHP, using Equations C-F.

Analyzability = $W_{AHP\text{-}Complexity\ For\ Analyzability}$ X (M1+M2+M3 +M4) + $W_{AHP\text{-}Coupling\ For\ Analyzability}$ X (M5+M6+M7) + $W_{AHP\text{-}Duplication\ For\ Analyzability}$ X (M8) + $W_{AHP\text{-}UnitTestEfforts\ For\ Analyzability}$ X (M9+M10) …… **Equation C**

Changeability = $W_{AHP\text{-}Complexity\ For\ Changeability}$ X (M1+M2+M3 +M4) + $W_{AHP\text{-}Coupling\ For\ Changeability}$ X (M5+M6+M7) + $W_{AHP\text{-}Duplication\ For\ Changeability}$ X (M8) + $W_{AHP\text{-}UnitTestEfforts\ For\ Changeability}$ X (M9+M10)…… **Equation D**

Stability = $W_{AHP\text{-}Complexity\ For\ Stability}$ X (M1+M2+M3 +M4) + $W_{AHP\text{-}Coupling\ For\ Stability}$ X (M5+M6+M7) + $W_{AHP\text{-}Duplication\ For\ Stability}$ X (1/M8) + $W_{AHP\text{-}UnitTestEfforts\ For\ Stability}$ X (M9+M10) ……..**Equation E**

Testability = $W_{AHP\text{-}Complexity\ For\ Testability}$ X (M1+M2+M3 +M4) + $W_{AHP\text{-}Coupling\ For\ Testability}$ X (M5+M6+M7) + $W_{AHP\text{-}Duplication\ For\ Testability}$ X (M8) + $W_{AHP\text{-}UnitTestEfforts\ For\ Testability}$ X (M9+M10)……. **Equation F**

The Weights (W_{AHP}) used in the equations were derived using the AHP process explained earlier, using the AHP scoring protocol, and the normalization of eigenvalues was derived using the table shown in Appendix C.

Table 13

Software Attributes, Their Measures, and the Actual Data Template Sheet

Measure	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10

Iteration	(0 < McCabe CC <10)	1 /(11 < McCabe CC < 20)	1 /(21 < McCabe CC < 50)	1/ Unit Size	0 < CBO < 6	7 < CB0 < 14	1/ (15 < CBO < 50)	Assert/Class	Test Coverage	1/Cloning
I1										
I61										

The values of M1 through M10 were retrieved using the tools for all iterations as shown in appendix G. As an example of the final software analyzability for iteration-1, the calculation is shown below. All 61 iteration specific values along with sum of weighted maintainability essentially were tabulated at the end, for final regression analysis against all X variables. Appendix H shows the computed values for all Y variables.

SA= 0.43 (.71+7.12+9.58+.31) + 0.25 (.61+.25+7.10) + 0.25 (10.26) + 0.06 (5.12 +.27)

SC = 0.40 (.71+7.12+9.58+.31) + 0.17 (.61+.25+7.10) + 0.17 (10.26) + 0.23 (5.12 +.27)

SS = 0.15 (.71+7.12+9.58+.31) + 0.19 (.61+.25+7.10) + 0.19 (10.26) + 0.44 (5.12 +.27)

ST = 0.11 (.71+7.12+9.58+.31) + 0.14 (.61+.25+7.10) + 0.16 (10.26) + 0.57 (5.12 +.27)

Similarly, for each iteration, values for corresponding X variables were also calculated in a separate table, as shown below.

Table 14

ASDM or X Variables and Their Specific Measure with the Actual Data Collection Template Sheet as an Example for 10 iterations

	X1	X2	X3
Revision /Iteration	Test Driven Development/TDD (Test Class Count/Total Class)	Refactoring (REFR) (1 / Count of Classes with Cyclomatic Complexity >21)	Continuous Integration (CI) (Count of Successful builds)
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			

Lastly, multiple regression analysis was conducted over final values of all Ys separately as well as for the sum of weighted Y on each X variable separately, yielding 5 regression models and statistics.

Reliability and Validity

Reliability and Validity of the Measurement

Reliability and validity within a quantitative study is important to ensure that the set of measurements are consistent and valid. These two quality aspects of measurement are often analogously referred to as *precision* and *accuracy*. In this study, inter-rater reliability, and test-retest reliability tests were conducted during the measurement of X and Y variables. Through inter-rater reliability, the same software tools and metrics were used by a different person other than the researcher of this study to measure the coupling, complexity, duplication, and unit size of the same software system including, measurement of TDD, REFR, and CI variables. Additionally, the researcher then measured and analyzed the collected metrics for all the operationalized variables (TDD, REFR, CI, and SA, SC, SS, ST, and SM) at two different times to support test-retest reliability. Note that the post-facto source code was downloaded for analysis of static quality attributes that were least likely to change over time, and this test-retest step showed no change over time. Lastly, all the source code revisions were ordered randomly, and the source code attributes were measured again through multiple-form reliability test. No change in the measured values was seen at the end of this test as well.

Reliability of the measure thus ensured through these three reliability tests viz. inter-rated reliability and test-retest reliability, and multiple-form reliability. Next, reliability is necessary, but not a sufficient condition for validity and hence additional approaches were taken to address validity as below.

Validity, according to Kaplan & Saccuzzon (2001), can been described as the agreement between a measure and the quality it is intended to measure. It also refers to the accuracy of the measurement. The software tools listed in Table C6 were used to automate the collection of key metrics data pertaining to each independent and dependent variable. To address threats to validity of the measurement, the source code classes and methods were selected from every iteration for their validation against the actual status of source code classes and methods within every revision or iteration. In other words, the researcher validated the actual type and structure of the class against the measured value of their Cyclomatic complexity, coupling, unit size, and assert counts, for their accuracy. Note that software tools listed in table C6 were used to conduct the measurement of source code attributes that were correlated with actual set of measurement calculated using the underlying construct of the actual measure. For instance, Cyclomatic complexity is the number of linearly independent path in the program or source code, in this study for a class or method. The researcher measured the actual path manually for selected sampled classes/methods from all iterations and compared it with the actual measured paths by the tool used in this study. The manual measurement yielded the same values as that of the automated measurement.

Content validity of the measure was addressed through measurement of complete revision of source code during that iteration. In other words, all the software classes were evaluated when measuring system attributes. Content under-representation was thus 0 with 100% source code examination with no exclusions.

Additionally, a test has content validity when the test items are selected to comply or adhere to specified criteria built through an extensive examination of the subject domain (Anastasia & Urbina, 1997). This study utilized well-proven measures and metrics that are being used by the researchers in the field of software engineering. The selected set of measures and the researchers are tabulated in Table 5.

External Validity

I took proactive steps to ensure that the selected instrument or metrics measured all of the dependent variables due to its relationship with similar studies conducted earlier to uphold construct validity. In order to protect content validity, filtering based on existing research literature was applied while collecting the data from selected software systems developed using ASDM, in order for this study to address the research questions. The literature review is also being used to validate all the employed measures of software maintainability and sub-characteristics in this study. Although the nature of research question demanded the data set selection from stable an IT organization where ASDM has been used extensively over several years, the external validity (Leedy & Ormrod, 2005) of this study is limited since it studied the software system within single IT organization.

Internal Validity

Internal validity is applicable to this quantitative study because its implied intent is to establish a causal relationship between independent variables (ASDM model's key characteristics: TDD, Refactoring, and CI) and software maintainability, analyzability, changeability, stability, and testability. According to Trochim (2007), internal validity assists to comprehend whether the observed changes in the dependent variable can be attributed to a program, treatment, intervention, or exposure. For this study, the goal is to determine whether and to what extent ASDM impacts software maintainability that is essentially a function of SA, SC, SS, and ST. Analysis for homoscedasticity was conducted to ensure the variance of errors is the same across all the levels of independent ASDM constituting: TDD, Refactoring, and CI. Slight heteroscedasticity may not have a significant effect, but higher heteroscedasticity in the residual variance can distort the findings, paralyze the analysis, and lead to type I error (Berry & Feldman, 1985; Tabachnick & Fidell, 2001). Additionally, Variation Inflation Factors (VIF) were also measured and recorded to check for the presence of multi-colinearity among independent variables. Attention to the strength of the regression model was also given when conducting regression analysis.

Chapter Summary

The literature review revealed that software maintainability is indeed a key quality attribute and hence a valid concern for IT management within the software life cycle specifically driven by Agile as a development approach. The literature also

suggested that analyzability, changeability, stability, and testability of software further aggregates and directly impact software maintainability, and thus its significance must be closely integrated at the development phase. The increase in maintenance cost and efforts has been burdening IT organizations as well as core business organizations. Further, the literature also revealed that the poor software maintainability is a concern for IT management when adapting and leveraging ASDM within dynamic business landscape. A concern was that the ASDM and its impact to software maintainability, a key software product quality characteristic, have received little research attention.

This chapter discussed the research method for analyzing the software maintainability as an impact of ASDM within the software evolution cycle. To remedy the shortcoming of past studies that relate the ASDM to software maintainability, this research employed a method leveraging the data analysis approach used by Kanellopoulos et al. (2010). The variables analyzed included ASDM-specific independent variables and software maintainability, software analyzability, software changeability, software stability, and software testability, with iterative changes analyzed in all these listed independent and dependent variables.

In summation, this chapter presented the research design, selected data set, research analytical method, and reasoning for choosing a quantitative design and quasi experimental method employing actual case study specific data sets. This section of the study also related the research questions and hypothesis to the Lehman and Belady's (1996) software evolution and maintenance laws. Study participants were IT

professionals working for the semiconductor organization who worked from multiple work locations within the United States. Purposive selection of data was used for this study, as it is driven by the specific research question that demands selection of software system that is designed and evolved using ASDM approach. The existing primary data was used in this research that was a post facto in nature. The data analysis was conducted using AHP process, SPSS, and MiniTab software tool. Regression analysis using MiniTab was performed to analyze the extent of impact of ASDM specific key variables on SA, SC, SS, ST, and SM. The focus was also on assessing the impact on resultant weighted maintainability through final regression analysis, which essentially was in compliant with each independent regression models run with Y1, Y2, Y3, and Y4. In chapter 4, results of the data analysis are presented and synthesized to answer the hypotheses and research questions. Accordingly, the stated hypotheses were accepted or rejected. The results analyzed in Chapter 4 provide data-driven insight into how ASDM as a development approach may shape and impact software maintainability. From the results obtained in chapter 4 based on the model developed in this chapter, ASDM practitioners, Agile PMO, IT application management, and software maintenance organizations can better comprehend, manage Agile-driven software life cycle, and provide pragmatic intervention to maintainability aspect of the software system.

Chapter 4: Data Analysis and Results

This quantitative study was conducted to determine whether and how the Agile software development approach in an IT organization impacts the software maintainability characterisitics. The Agile software development model can be statistically attributed to the changes in software maintainability in terms of software analyzability, changeability, stability, and testability of the software. The analysis consists of a multiple regression analysis, which incorporates the following predictive or independent variables: the TDD or X1, REFR or X2, and CI or X3. The dependent variables were obtained from the ISO/IEC 9126 quality standard for software maintainability. SA or Y1, SC or Y2, SS or Y3, and ST or Y4 were calculated utilizing equations C, D, E, and F from Chapter 3. As a part of these calculations, source code properties such as complexity and coupling were analyzed and measured using selected attributes at every revision marked by the Agile iteration. The results presented in this chapter answered the following five research questions:

RQ1. How does the Agile software development model (ASDM), which is characterized by Test driven development (TDD), Refactoring (REFR), and Continuous Integration (CI), impact software analyzability (SA)?

RQ2. How does the Agile software development model (ASDM), which is characterized by Test driven development(TDD), Refactoring (REFR), and Continuous Integration (CI), impact software changeability (SC)?

RQ3. How does the Agile software development model (ASDM), which is characterized by Test driven development (TDD), Refactoring (REFR), and Continuous Integration (CI), impact software stability (SS)?

RQ4. How does the Agile software development model (ASDM), which is characterized by Test driven development (TDD), Refactoring (REFR), and Continuous Integration (CI), impact software testabiliy (ST)?

Additionally, the following research question is also answered through an examination of the impact on the resultant maintainability characteristic that is a function of SA, SC, SS, and ST.

RQ5. How does the Agile software development model (ASDM), which is characterized by Test driven development (TDD), Refactoring (REFR), and Continuous Integration (CI), impact resultant weighted software maintainability (SM)?

The results also tested the following 15 null and alternative hypotheses. To answer RQ1, below three pairs of hypotheses were tested.

$H_0$1: TDD has no influence on the Software Analyzability.

H_A1: TDD has a positive influence on the Software Analyzability.

$H_0$2: Refactoring has no influence on the Software Analyzability.

H_A2: Refactoring has a positive influence on the Software Analyzability.

$H_0$3: CI has no influence on the Software Analyzability.

H_A3: CI has a positive influence on the Software Analyzability.

To answer RQ2, the next three pairs of hypotheses were tested.

$H_0$4: TDD has no influence on the Software Changeability.

H_A4: TDD has a positive influence on the Software Changeability.

$H_0$5: Refactoring has no influence on the Software Changeability.

H_A5: Refactoring has a positive influence on the Software Changeability.

$H_0$6: CI has no influence on the Software Changeability.

H_A6: CI has a positive influence on the Software Changeability.

To answer RQ3, three pairs of hypotheses were tested.

$H_0$7: TDD has no influence on the Software Stability.

H_A7: TDD has a positive influence on the Software Stability.

$H_0$8: Refactoring has no influence on the Software Stability.

H_A8: Refactoring has a positive influence on the Software Stability.

$H_0$9: CI has no influence on the Software Stability.

H_A9: CI has a positive influence on the Software Stability.

To answer RQ4, three pairs of hypotheses were tested.

$H_0$10: TDD has no influence on the Software Testability.

H_A10: TDD has a positive influence on the Software Testability.

$H_0$11: Refactoring has no influence on the Software Testability.

H_A11: Refactoring has a positive influence on the Software Testability.

$H_0$12: CI has no influence on the Software Testability.

H_A12: CI has a positive influence on the Software Testability.

Lastly, to answer RQ5, three pairs of hypotheses were proposed and tested for the resultant Maintainability that was derived from sum of weighted SA, SC, SS, and ST.

$H_0$13: TDD has no influence on the resultant Software Maintainability.

H_A13: TDD has a positive influence on the resultant Software Maintainability.

$H_0$14: Refactoring has no influence on the resultant Software Maintainability.

H_A14: Refactoring has a positive influence on the resultant Software Maintainability.

$H_0$15: CI has no influence on the on the resultant Software Maintainability.

$H_0$15: CI has no influence on the on the resultant Software Maintainability.

Data Collection and Analysis

Prior to beginning the study, approval was obtained from Walden University's IRB for the submitted dissertation proposal. The data used in this study were made available from an IT department in a U.S.-based semiconductor technology manufacturing organization. The data contained source code properties retrieved from each software revision marked by weekly iteration cycle, located within the software code repository. The selected software system followed the XP approach, an Agile software development model. A total of 61 software revisions or versions were collected for data analysis beginning from August 2010 through June 2011. Using this post facto data from the source code repository from the IT organization was deemed appropriate because this data provides a comprehensive measurement points during the Agile model driven evolution of software life cycle , as noted by Sindhgatta et al. (2010). The

variables pertinent to the evaluation of the impact of the Agile model on software maintainability characteristics were retrieved from existing and current Agile software development and maintainability literature . Appendix D presents the material used from this source code data as well as the variables collected.

Incomplete or Missing Data

The source code properties or attributes-related data were collected using various software tools tabulated in Table C6. The subjective expert judgment scoring related data was collected from the ASDM expert from the IT department of this semiconductor technology manufacturing organization. Data for iteration in the 1st week of January 2011 that make up the variables in this study were not recorded due to vacation observed by the development team. Additionally, the data for test coverage metrics were not present for the initial 10 iterations, and those values were derived using the EM technique. The final data set was significant enough to conduct statistical analysis over 61 software revisions or iteration points for the software system that was developed using Agile approach.

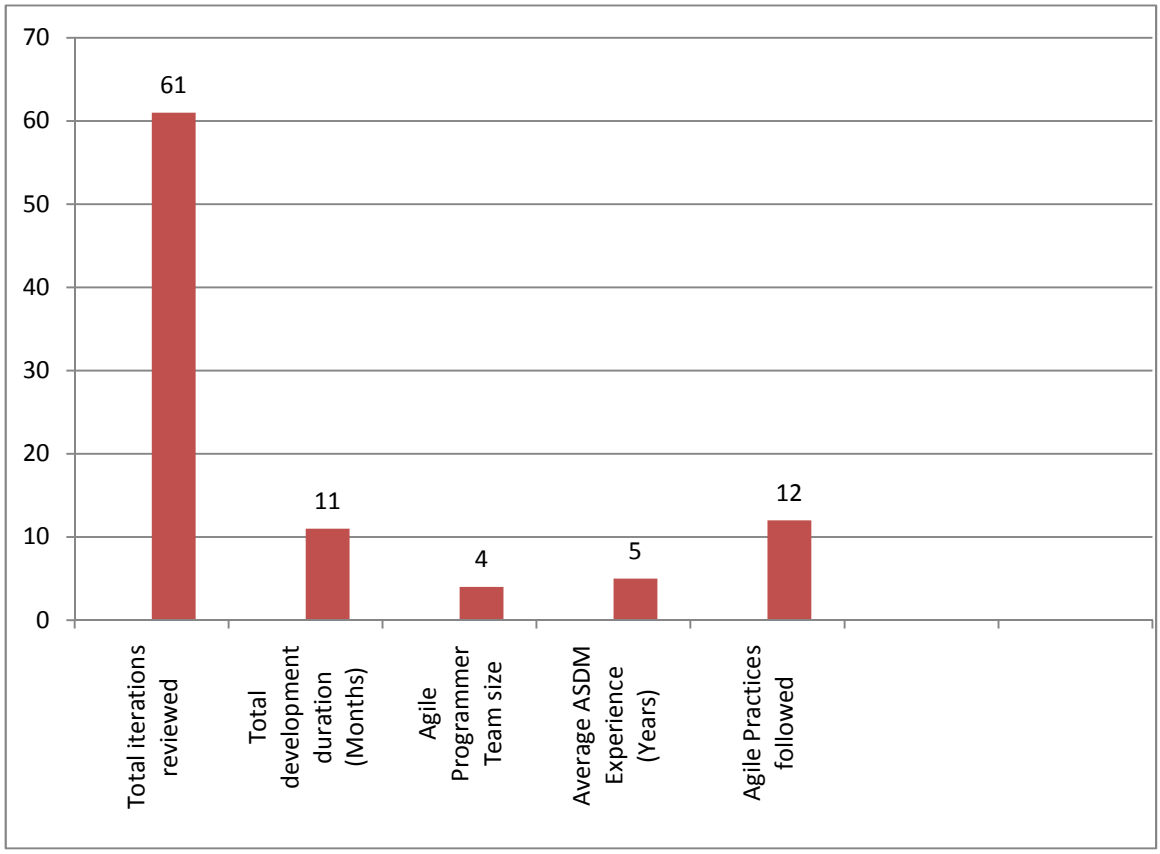


Figure 17. A chart illustrating the analyzed software system specific statistics.

Software System Data Selection

From three unique software systems developed during 2010-2011, one software system was deemed eligible for analysis based on the criteria highlighted in Table 7. The purposive selection was necessary to select the right software system and ensure that it did not contain missing values as well as any missing software iteration and that it would accurately allow the measurement to assess the impact of ASDM on software maintainability characteristics. The following steps were also taken to improve the data analysis quality of the available data set of software system.

1. Testing coverage of specific data was not available for 10 iterations because the CCNET software tool was not integrated for these iterations to collect test coverage-specific data. Hence, the expectation maximization (EM) was computed from all the available data set.
2. Cyclomatic complexity-specific data was collected for every iteration and was grouped into three main categories based on the risk level. This grouping allowed for granular analysis of the software classes influenced due to ASDM. The three categories that were employed were $0 < CC < 10$, $11 < CC < 20$, and $21 < CC < 50$.
3. Coupling between object (CBO) metric data was also collected for all 61 iterations and it was grouped into three main categories based on the risk levels. This categorization allowed the analysis of the impact of ASDM variables to the granular level and the identity of the percentage of classes influenced in each category. The three categories employed for CBO metrics were $0 < CBO < 6$, $7 < CBO < 14$, and $15 < CBO < 50$.

Data Standardization

Data was first transferred into data analysis tool: SPSS 19.0. Prior to conducting any analyses, data was manipulated in several ways. First, for all three predictors (TDD, REFR, CI), standardized values (*z* scores) were created so that all independent variables were on the same scale. Next, expectation maximization (EM) was used to estimate the values of missing data for test coverage (subvariable). EM creates a correlation matrix

by assuming the distribution for the partially missing data and basing inferences about missing values on the likelihood under that normal distribution. In the first step, the procedure determines the expected values of the missing data using the observed values and the parameter estimates. The second step is the maximum likelihood estimation wherein the missing data is filled in and convergence is achieved (Tabachnick & Fidell, 2006).

Descriptive Statistics

Descriptive statistics were conducted on the independent (prior to standardizing and after) and dependent variables, as well as the numbers that were used to calculate the independent and dependent variables. Prior to standardizing the independent variables, the TDD measurements ranged from 0.17 to 0.35 with a mean of 0.26 (*SD* = 0.05). The REFR measurements ranged from 6.78 to 7.94 with a mean of 7.21 (*SD* = 0.30). The CI measurements ranged from 1.00 to 74.00 with a mean of 33.07 (*SD* = 19.57). After standardizing the independent variables, the mean will always be set at 0.00 and the standard deviation will be 1.00. TDD ranged from -1.86 to 1.65, REFR ranged from -1.44 to 2.44, and CI ranged from -1.64 to 2.09.

SA ranged from 12.50 to 14.56 with a mean of 13.71 (*SD* = 0.39). SC ranged from 11.43 to 13.52 with a mean of 12.85 (*SD*= 0.48). SS ranged from 8.49 to 10.17 with a mean of 9.41 (*SD* =0.60). SM ranged from 472.99 to 563.28 with a mean of 537.86 (*SD* = 20.93). Means and standard deviations for the independent and dependent variables are presented in Table 15.

Table 15

Means and Standard Deviations for TDD, REFR, CI, SA, SC, SS, ST, and SM

Variable	*M*	*SD*
TDD	*0.26*	*0.05*
REFR	*7.21*	*0.30*
CI	*33.07*	*19.57*
SA	*13.71*	*0.39*
SC	*12.85*	*0.48*
SS	*9.99*	*0.50*
ST	*9.41*	*0.60*
SM	*537.86*	*20.93*

Descriptive statistics were also conducted on the source code attributes that were used to calculate the dependent variables. The low risk category for the Cyclomatic complexity, 0 < McCabe CC <10 ranged from 0.71 to 0.75 with a mean of 0.74 (*SD* = 0.01). The medium risk category for the Cyclomatic complexity, 1 /11 < McCabe CC < 20 (I/CC) ranged from 6.96 to 9.26 with a mean of 8.08 (*SD* = 0.67). The highest risk category for the Cyclomatic complexity, 1 /21 < McCabe CC < 50 (I (CC) ranged from 9.51 to 11.74 with a mean of 10.26 (*SD* = 0.48). Similarly, the low risk category for coupling between objects, 0 < CBO < 6 ranged from 0.60 to 0.65 with a mean of 0.62

(*SD* = 0.01). The medium risk category for coupling between objects, 7 < CB0 < 14 ranged from 0.24 to 0.29 with a mean of 0.27 (*SD* = 0.01). The highest risk category for the coupling between object, 1/ 15 < CBO < 50 ranged from 7.10 to 10.66 with a mean of 9.23 (*SD* = 0.73). The count of "Assert statements" per class or "Assert/Class" ranged from 5.12 to 8.71 with a mean of 7.31 (*SD* = 1.04). Test Coverage ranged from 25.52 to 32.01 with a mean of 29.83 (*SD* = 1.07). The "1/ Unit Size" ranged from 0.31 to 0.36 with a mean of 0.34 (*SD* = 0.01). The "1/Cloning" ranged from 6.99 to 11.63 with a mean of 9.52 (*SD* = 1.41). The actual cloning percentage ranged from 0.09 to 0.14 with a mean of 0.10 (*SD* = 0.02). The normality of the regressors and the dependent variable were adequate for performing the analysis.

The trend charts for all the source code properties were created over 61 software revisions or iterations. Software complexity showed positive improvement for all of the risk categories. Total percent of software classes with CC (0 < CC < 10) or low risk category increased from 71 to 75%. Total percent of software classes with CC (11 < CC < 20) or medium risk category decreased from 14 to 11%; finally, the total percentage of software classes with CC (21 < CC < 50) or higher risk category decreased from 10 to 9%. This trend showed that the Agile development approach effectively yielded less complex software classes at the end of the development cycle, inferring improved software analyzability and changeability. Secondly, CBO also showed positive improvement for all the three risk categories. Total percent of software classes with CBO (0 < CBO < 6) or low risk category increased from 60 to 64%. Total percent of software

classes with CBO (7 < CBO < 14) or medium risk category increased slightly from 25 to 26%; and finally, the total percentage of software classes with CBO (15 < CC <50) or higher risk category decreased from 14 to 9%. This trend showed that the Agile development approach effectively yielded more loosely coupled software classes at the end of the development cycle, inferring improved software changeability and stability. Code duplication trend showed that duplication increased from 9 to 14% inferring negative influence on software analyzability and changeability. The overall increase in code duplication was, however, compensated by reduction in complexity and coupling of the software code.

Lastly, the number of asserts indicating the unit test efforts trend showed positive improvement with an increase from 5.12 to 8.63. The trend for test code coverage over 61 iterations showed moderate increase from 27 to 29%. The unit size or the size of the class showed positive trend with unit size reduction from 3.24 to 2.74 statements per class inferring the improvement in the software analyzability, changeability, and testability. The large size of classes and methods continue to pose a higher risk as it directly influences the understanding by the software developers and maintainers (Lorenz & Kidd, 1994). The smaller the size of the class, the more it becomes easy to understand, changes, and test for the development and maintenance resource.

Multiple Regression Data Analysis

This research study used a multiple regression approach on the ASDM related variables and variations observed in software analyzability, changeability, stability, and

testability to derive the best-fit model for analyzing the data set as per the methodology presented in Chapter 3. The results are explicated in this section of the chapter.

The computation of actual values for software analyzability, changeability, stability, and testability was accomplished using equations C, D, E, and F for all 61 software development iterations and corresponding source code revisions. Essentially, all 61 revisions were analyzed using the tools listed in Table C6 to yield the values for all the listed software measures. The ASDM-specific variables were then regressed over software analyzability, changeability, stability, testability, and resultant weighted maintainability. The regression analysis results indicated that overall the ASDM is statistically significant predictor of software maintainability and its four sub-characteristics. The results in Table 27 also show that model predicted 82.8% of the variation in software maintainability at a 95% confidence level.

Preliminary Analysis

In preliminary analysis, the assumptions of linearity, normality, homoscedasticity, and absence of multicollinearity were assessed before conducting the regression analysis. Linearity was assessed by examining scatter plots; the assumption was verified. To assess normality, skew and kurtosis values were assessed. As tabulated in Table 16, skew was found to be less than the absolute value of 2 and kurtosis less than the absolute value of 7, meeting the assumption of normality (Kline, 2005). The residual plots are shown for all Y variables in Figures 18 through 22.

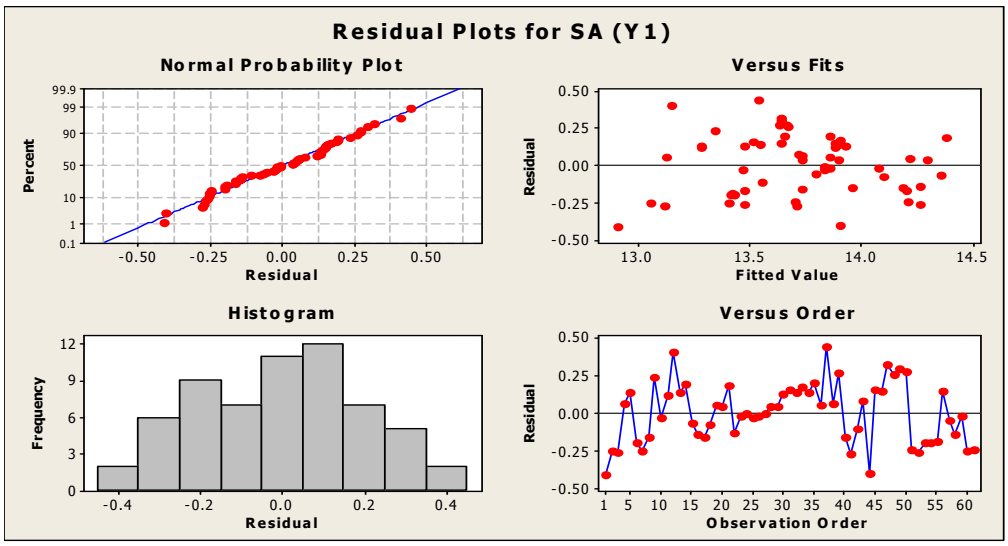


Figure 18. Residual plot for software analyzability (SA).

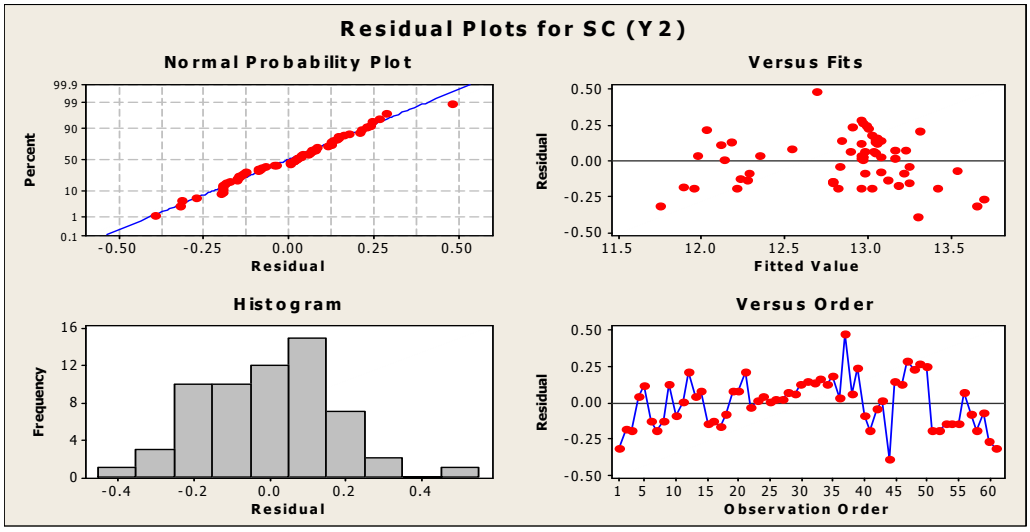


Figure 19. Residual plot for software changeability (SC).

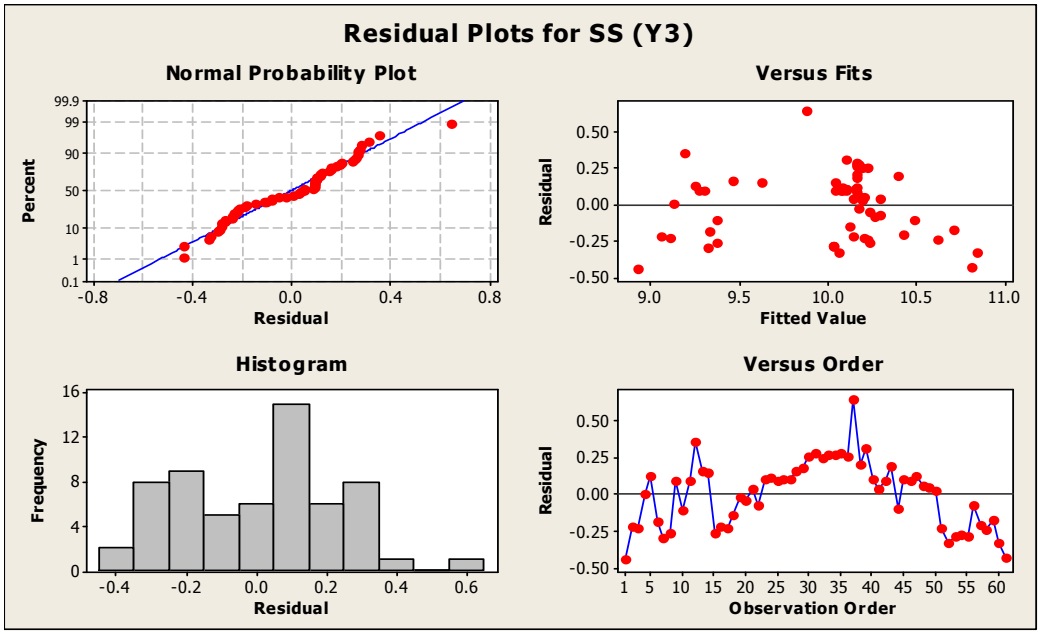


Figure 20. Residual plot for software stability (SS).

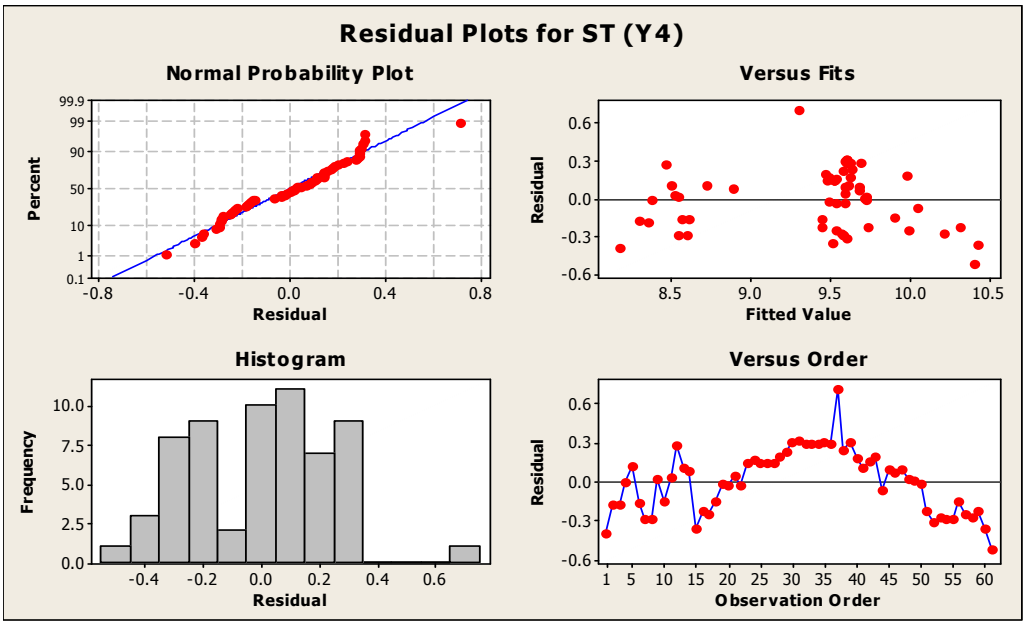


Figure 21. Residual plot for software testability (ST).

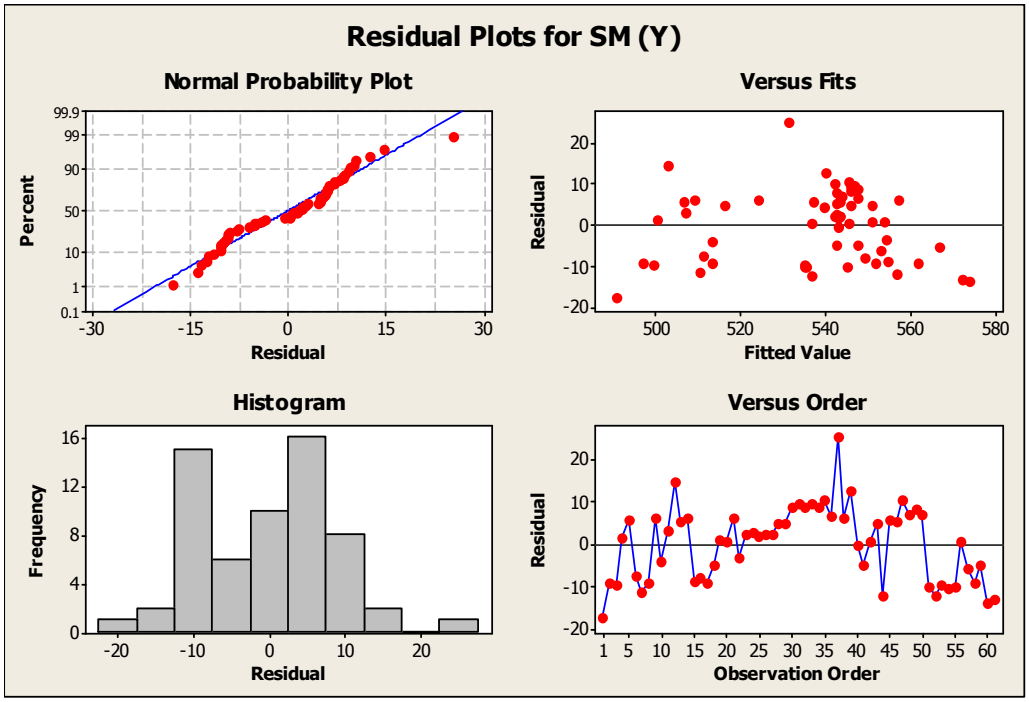


Figure 22. Residual plot for software maintainability (SM).

Homoscedasticity was thus assessed using values for error residuals and viewing the scatter plots of the error residuals. The scatter plot appeared to be rectangular, and thus, the assumption was met. The assumption of absence of multicollinearity was also assessed by examining the Variance Inflation Factors (VIF).

Additionally, a correlation matrix was created to assess multicollinearity and to be certain an $r < .80$ was observed among the independent variables.

Table 16

Skew and Kurtosis for TDD, REFR, CI, SA, SC, SS, ST, and SM

Variable	*Skew*	*Kurtosis*
TDD	*-0.40*	*-0.81*
REFR	*0.79*	*0.12*
CI	*0.17*	*-1.21*
SA	*-0.77*	*0.77*
SC	*-1.15*	*0.58*
SS	*-1.21*	*0.64*
ST	*-1.08*	*0.06*
SM	*-1.24*	*0.80*

Results of the correlation matrix are presented in Table 17. All the assumptions related to homoscedasticity, linearity, and normality was validated before conducting the regression analysis using MiniTab tool. VIF values are shown in all the regression models below. These are also tabulated in Appendix K showing almost no multicollinearity with all the values close to 1 (1.074, 1.138, 1.205 respectively for TDD, REFR, and CI variables).

Table 17

Pearson Product Moment Correlations among TDD, REFR, and CI

Variable	*TDD*	*REFR*
REFR	*-.12*	
CI	*.26**	*-.35***

Note. *$p < .05$, **$p < .01$.

Research Question One

RQ1: How does the Agile software development model (ASDM), which is characterized by Test driven development (TDD), Refactoring (REFR), and Continuous Integration (CI), impact software analyzability (SA)?

To assess the first research question, one multiple regression analysis was conducted. The result of the multiple regression was significant, F (3, 57) = 54.50, $p <$ .001, suggesting that the model predicted (R^2) 74.1% of the variance in SA. The coefficient of determination, or R^2 value, for this regression model measures the proportion of the variation in the dependent variable, SA, which is explained by the combination of the independent variables (TDD, REFR, and CI) in this regression model. Further analysis revealed that TDD successfully predicted SA, $B = 0.19$, $p < .001$, suggesting that for every point/percent increase in TDD, SA increased by 0.19 points/percent. Also, REFR or refactoring variable successfully predicted SA, $B = 0.30$,

$p < .001$, suggesting that for every point/percent increase in REFR, SA increased by 0.30 points/percent. Lastly, CI is not shown to be a significant predictor looking at the p value of .561 ($p > .001$). The regression equation is SA = 13.71 + 0.19 * TDD + .30 * REFR. The null hypothesis is rejected; the model is significant as a whole, and both TDD and REFR offered a unique contribution to predicting SA. Results of the multiple regression are presented in Table 18.

Table 18

Multiple Regression for TDD, REFR, and CI Predicting SA

Model	B	SE	β	*t*	*p*
TDD	0.19	0.03	.49	7.01	.001
REFR	0.30	0.03	.78	10.78	.001
CI	0.02	0.03	.04	0.59	.561

Note. F (3, 57) = 54.50, $p < .001$, $R^2 = 0.741$.

Table 19

Analysis of Variance for software analyzability (SA)

Analysis of Variance					
Source	DF	Sum of squares	Mean square	F value	Pr > F
Model	3	6.8262	2.2754	54.50	<.0001
Error	57	2.3799	0.0418		
Corrected total	60	9.2061			
		R-square	74.1%		
		Adj. R-sq	72.8%		

Predictor	Coef	SE Coef	T	P	VIF
Constant	13.7151	0.0262	524.23	0.000	
TDD (X1)	0.19079	0.02731	6.99	0.000	1.074
REFR (X2)	0.30394	0.02815	10.80	0.000	1.138
CI (X3)	0.01700	0.02895	0.59	0.559	1.205

Research Question 2

RQ2: How does the Agile software development model (ASDM), which is characterized by Test driven development (TDD), Refactoring (REFR), and Continuous Integration (CI), impact software changeability (SC)?

To assess research question 2, one multiple regression analysis was conducted. The result of the multiple regression was significant, F (3, 57) = 126.70, $p < .001$, suggesting that the model predicted (R^2) 87.0% of the variance in SC. Further analysis revealed that TDD successfully predicted SC, $B = 0.40$, $p < .001$, suggesting that for every point increase in TDD, SC increased by 0.40 points. Also, REFR successfully predicted SC, $B = 0.25$, $p < .001$, suggesting that for every point increase in REFR, SC increased by 0.25 points. CI is not shown to be a significant predictor. The regression equation is SC = 12.85+ .40 * TDD + .25 * REFR. The null hypothesis is rejected; the model was significant as a whole, and both TDD and REFR offered a unique contribution to predicting SC. Results of the multiple regression are presented in Table 20.

Table 20

Multiple Regression for TDD, REFR, and CI Predicting SC

Model	B	SE	β	*t*	*p*
TDD	0.40	0.02	.83	16.85	.001
REFR	0.25	0.02	.52	10.29	.001
CI	0.01	0.03	.02	0.40	.693

Note. F (3, 57) = 126.70, $p < .001$, $R^2 = 0.870$.

Table 21

Analysis of Variance for software changeability (SC)

Analysis of Variance					
Source	DF	Sum of squares	Mean square	F value	Pr > F
Model	3	11.9247	3.9749	126.70	<.0001
Error	57	1.7882	0.0314		
Corrected total	60	13.7129			
		R-square	87.0%		
		Adj. R-sq	86.3%		

Predictor	Coef	SE Coef	T	P	VIF
Constant	12.8533	0.0227	556.77	0.000	
TDD (X1)	0.39765	0.02367	16.80	0.000	1.074
REFR (X2)	0.25085	0.02440	10.28	0.000	1.138
CI (X3)	0.01044	0.02510	0.42	0.679	1.205

Research Question 3

RQ3: How does the Agile software development model (ASDM), which is characterized by Test driven development (TDD), Refactoring (REFR), and Continuous Integration (CI), impact software stability (SS)?

To assess research question 3, one multiple regression analysis was conducted. The result of the multiple regression was significant, F (3, 57) = 73.90, $p < .001$, suggesting that the model predicted (R^2) 79.6% of the variance in SS. Further analysis revealed that TDD successfully predicted SS, $B = 0.42$, $p < .001$, suggesting that for every point increase in TDD, SS increased by 0.42 points. Also, REFR successfully predicted SS, $B = 0.19$, $p < .001$, suggesting that for every point increase in REFR, SS increased by 0.19 points. CI is not shown to be a significant predictor. The regression equation is SS = 9.99 + .42 * TDD + .19 * REFR. The null hypothesis is rejected; the model was significant as a whole, and both TDD and REFR offered a unique contribution to predicting SS. Results of the multiple regression are presented in Table 22.

Table 22

Multiple Regression for TDD, REFR, and CI Predicting SS

Model	B	SE	β	*t*	*p*
TDD	0.42	0.03	.85	13.72	.001
REFR	0.19	0.03	.38	5.95	.001
CI	0.01	0.03	.01	0.21	.838

Note. F (3, 57) = 73.90, $p < .001$, $R^2 = 0.796$.

Table 23

Analysis of Variance for software stability (SS)

Analysis of Variance					
Source	DF	Sum of squares	Mean square	F value	Pr > F
Model	3	11.7998	3.9333	73.90	<.0001
Error	57	3.0338	0.0532		
Corrected total	60	14.8336			
		R-square	79.6%		
		Adj. R-sq	78.5%		

Predictor	Coef	SE Coef	T	P	VIF
Constant	9.99151	0.02954	338.25	0.000	
TDD (X1)	0.42227	0.03083	13.70	0.000	1.074
REFR (X2)	0.18906	0.03178	5.95	0.000	1.138
CI (X3)	0.00724	0.03269	0.22	0.826	1.205

Research Question 4

RQ4: How does the Agile software development model (ASDM), which is characterized by Test driven development (TDD), Refactoring (REFR), and Continuous Integration (CI), impact software testabiliy (ST)?

To assess research question 4, one multiple regression analysis was conducted. The result of the multiple regression was significant, F (3, 57) = 98.43, $p < .001$, suggesting that the model predicted (R^2) 83.9% of the variance in ST. Further analysis revealed that TDD successfully predicted ST, $B = 0.54$, $p < .001$, suggesting that for every point increase in TDD, ST increased by 0.54 points. Also, REFR successfully predicted ST, $B = 0.16$, $p < .001$, suggesting that for every point increase in REFR, ST increased by 0.16 points. From the p value for CI variable, CI is not shown to be a significant predictor. The regression equation is ST = 9.41 + .54 * TDD + .16 * REFR. The null hypothesis is rejected; the model was significant as a whole, and both TDD and REFR offered a unique contribution to predicting ST. Results of the multiple regression are presented in Table 24 along with ANOVA results tabulated in Table 25 below.

Table 24

Multiple Regression for TDD, REFR, and CI Predicting ST

Model	B	SE	β	*t*	*P*
TDD	0.54	0.03	.91	16.47	.001
REFR	0.16	0.03	.26	4.63	.001
CI	0.00	0.04	.00	-0.04	.970

Note. F (3, 57) = 98.43, $p < .001$, $R^2 = 0.839$.

Table 25

Analysis of Variance for software testability (ST)

Analysis of Variance					
Source	DF	Sum of squares	Mean square	F value	Pr > F
Model	3	17.8760	5.9587	98.43	<.0001
Error	57	3.4505	0.0605		
Corrected total	60	21.3265			
		R-square	83.8%		
		Adj. R-sq	83.0%		

Predictor	Coef	SE Coef	T	P	VIF
Constant	9.41236	0.03150	298.79	0.000	
TDD (X1)	0.54048	0.03288	16.44	0.000	1.074
REFR (X2)	0.15790	0.03390	4.66	0.000	1.138
CI (X3)	-0.00009	0.03486	-0.00	0.998	1.205

Research Question 5

RQ5: How does the Agile software development model (ASDM), which is characterized by Test driven development (TDD), Refactoring (REFR), and Continuous Integration (CI), impact resultant software maintainability (SM)?

To assess research question 5, one multiple regression analysis was conducted. The result of the multiple regression was significant, F (3, 57) = 91.50, $p < .001$, suggesting that the model predicted (R^2) 82.8% of the variance in SM. Further analysis revealed that TDD successfully predicted SM, $B = 17.05$, $p < .001$, suggesting that for every point increase in TDD, SM increased by 17.05 points. Also, REFR successfully predicted SM, $B = 10.68$ $p < .001$, suggesting that for every point increase in REFR, SM increased by 10.68 points. CI is not shown to be a significant predictor.

Table 26

Multiple Regression for TDD, REFR, and CI Predicting SM

Model	B	SE	β	*t*	*p*
TDD	17.05	1.19	.81	14.30	.001
REFR	10.68	1.23	.51	8.70	.001
CI	0.28	1.26	.01	0.22	.825

Note. F (3, 57) = 91.46, $p < .001$, $R^2 = 0.828$.

The regression equation is SM = 537.86 + 17.05 * TDD + 10.68 * REFR. The null hypothesis is rejected; the model was significant as a whole, and both TDD and REFR offered a unique contribution to predicting SA. Results of the multiple regression are presented in Table 26 above.

The strength of the software maintainability is further reinforced by the generated F value of 91.50 for F (3, 57), which indicates that the model is statistically proven to be

linear. The F table put forward by Aczel and Sounderpandian (2006) showed that the critical value for $f_{(3,57)}$ *df* at a 95%-confidence level is 2.57. The *f*-value obtained also indicates that in combination, the coefficients of the regressors are greater than zero (0), and thus there is a linear relationship between software analyzability, changeability, stability, and testability, and the independent variables of ASDM. Further, statistical reinforcement is addressed by the *p* value, which is less than 5%, also indicating the statistical significance of the model in aggregate.

Table 27

Analysis of Variance for software maintainability (SM)

Analysis of Variance					
Source	DF	Sum of squares	Mean square	F value	Pr > F
Model	3	21771.8	7257.3	91.50	<.0001
Error	57	4520.7	79.3		
Corrected total	60	26292.4			
		R-square	82.8%		
		Adj. R-sq	81.9%		

Predictor	Coef	SE Coef	T	P	VIF
Constant	537.865	1.14	471.71	0.000	
TDD (X1)	17.024	1.19	14.3	0.000	1.074
REFR (X2)	10.679	1.227	8.7	0.000	1.138
CI (X3)	0.29	1.262	0.23	0.819	1.205

The *p* value for CI is .819 inferring that the CI variable is insignificant predictor of software maintainability. Based on the above regression results, all the coefficients are statistically different from 0, with the exception of continuous integration (CI) variable with value of .26 that is close to 0. More important, and related to testing the hypotheses (*H*13, *H*14, *H*15), is the fact that the variation in software maintainability (SM) is statistically explained by TDD and REFR. All other independent variables except CI, the TDD and REFR are statistically significant at the 95% confidence level measured by the *p* values. The *p* values are measured for *df* = 57 against an alpha (α) of 5%. This statistical significance is highlighted in Table 27 with ANOVA results. The low variance inflation factor (*VIF*) of 1 (1.074, 1.138, 1.205 respectively for TDD, REFR, and CI) as seen in all five regression models, and collinearity diagnostics illustrate that no multicollinearity exists between the independent variables.

Lastly, the ANOVA was conducted for the adjusted model without CI variable in the regression. The TDD and REFR are significant at the 95% confidence interval by the *p* values. The *p* values are measured for *df* = 58 against an alpha (α) of 5%. This statistical significance is highlighted in Table 28 with ANOVA results. The low variance inflation factor (*VIF*) of 1 (1.014) for TDD and REFR as seen in this adjusted regression model, and collinearity diagnostics illustrate that no multicollinearity exists between the independent variables. The regression equation for this adjusted model is SM (Y) = 538 + 17.1 TDD (X1) + 10.6 REFR (X2).

Table 28

Adjusted model without CI - Analysis of Variance for software maintainability (SM)

Analysis of Variance					
Source	DF	Sum of squares	Mean square	F value	Pr > F
Model	2	21768	10884	139.51	<.0001
Error	58	4525	78		
		R-square	82.8%		
		Adj. R-sq	82.2%		

Predictor	Coef	SE Coef	T	P	VIF
Constant	537.865	1.131	475.61	0.000	
TDD (X1)	17.089	1.147	14.90	0.000	1.014
REFR (X2)	10.585	1.148	9.22	0.000	1.014

The results are also tabulated in appendix L. The result shows that this adjusted model is the best model that explains the variation in software maintainability with R^2 of 82.8%.

Hypotheses Testing

Using the results shown in ANOVA tables, the following hypotheses were tested for statistical significance based on the *t* and *p* values of the independent coefficients, as shown in Chapter 1 of this study. The criteria for the hypothesis tests were based on *df* = 57 at 95%-confidence level. In addition, the direction of the relationship between

software analyzability, changeability, stability, testability, and maintainability and the independent variables were also tested.

Hypothesis 1

$H_{1o}: \beta_1 = 0$: TDD has no influence on the Software Analyzability.

: TDD has a positive influence on the Software Analyzability.

Based on the results obtained, the alternative hypothesis is accepted because the coefficient for TDD has a *p* value of 0.001 and a *t* value of 7.01, which is above the *t* critical value of 1.67. Since I tested this analysis at a 95%- confidence level, the *p* value was .001, below 5%. The results led to the conclusion that and there is a relationship between TDD and SA. At the same time, the null hypothesis $\beta_1 = 0$ was rejected. Further, the results indicate that change in TDD is positively related to SA so that when TDD increases by a percentage point (1%), SA increases by approximately 0.19%.

Hypothesis 2

$H_{2o}: \beta_2 = 0$: Refactoring has no influence on the Software Analyzability.

: Refactoring has a positive influence on the Software Analyzability.

In this test, the alternative hypothesis was also accepted based on a *p* value of 0.001 at a 95% confidence level. The *p* value was .001, below 5%, while the *t* value (10.78) was above the *t* critical of 1.67. At the same time the null hypothesis $\beta_2 = 0$ was rejected. The test of the second hypotheses illustrates that a change in REFR

is positively related to software analyzability so that when refactoring efforts increases by a percentage point (1%), software analyzability level increases by approximately 0.30%.

Hypothesis 3

H_{30}: $\beta_3 = 0$: CI has no influence on the Software Analyzability.

: CI has a positive influence on the Software Analyzability.

For these hypotheses tests, the null hypothesis was not rejected because the regression analysis generated a *t* value of 0.59. Also, the *p* value of .561 was also higher than the 5% allowed for this type of test. At the same time, the null hypothesis was not rejected $(\beta_3 = 0)$ because the variable could be accepted when tested at a lower confidence level. The direction of the sign in the test of hypotheses 3 was positive, meaning that when a CI increases, SA increases. However, due to the statistical insignificance of CI, there is no meaningful elasticity explanation.

Hypothesis 4

H_{4o} : $\beta_4 = 0$: TDD has no influence on the Software Changeability.

: TDD has a positive influence on the Software Changeability.

For this test, the alternative hypothesis was accepted because the coefficient for TDD on the software changeability had a *p* value of approximately 0.001 and a *t* value at $t_{(57)}$ of 16.80. As this hypothesis was tested at a 95%-confidence level, the *p* value was .001, below 5%, while the *t* value of 16.80 was beyond the *t* critical of 1.67. The null hypothesis $\beta_4 = 0$ was rejected. The test of hypotheses 4 illustrate that change in TDD is positively related to the SA. Therefore when the test driven

development efforts during the Agile development increases by 1%, software changeability increased by approximately 0.40%. In summary, the capability of the software to allow the identified change to be implemented was influenced positively by TDD practice.

Hypothesis 5

$H_{50}: \beta_5 = 0$: Refactoring has no influence on the Software Changeability.

: Refactoring has a positive influence on the Software Changeability.

The null hypothesis was rejected based on the results. From the final hypotheses tests, the alternative hypothesis was also accepted. The coefficient for refactoring had a *p* value of <0.001 and *t* value of 10.29 at $t_{(57)}$ measured at a 95%-confidence level supporting this acceptance. The test of elasticity for the coefficient of refactoring has a positive relationship with software changeability. That is, when TDD increases by 1%, SC or software changeability increases by approximately 0.25%.

Hypothesis 6

$H_{60}: \beta_6 = 0$: CI has no influence on the Software Changeability.

: CI has a positive influence on the Software Changeability.

For these hypotheses tests, the null hypothesis was not rejected because the regression analysis generated a *t* value of 0.40 that is lower than the critical value of 1.67. Also, the *p* value of .693 was also higher than the 5% allowed for this type of test. The alternative hypothesis that there exists a relationship between continuous integration and software analyzability could not be accepted. At the same time, the null

hypothesis was not rejected $(\beta_6 = 0)$ because it may be accepted when tested at a lower confidence level. The direction of the sign in the test of hypotheses 6 was positive, meaning that when a CI increases software changeability increase. However, due to the statistical insignificance of CI, there is no meaningful elasticity explanation.

Hypothesis 7

$H_{7o}: \beta_7 = 0$: TDD has no influence on the Software Stability (SS).

: TDD has a positive influence on the Software Stability (SS).

For this test, the alternative hypothesis was accepted since the coefficient for TDD on the software stability had a *p* value of approximately 0.001 and a *t* value at $t_{(57)}$ of 13.71. As this hypothesis was tested at a 95%-confidence level, the *p* value was .001, below 5%, while the *t* value of 13.70 was beyond the *t* critical of 1.67. The null hypothesis $\beta_7 = 0$ was thus rejected. The test of hypotheses 7 illustrate that change in TDD is positively related to the SS. Therefore when the test driven development efforts during the Agile development increases by 1%, software stability increase by approximately 0.42%. In summary, software capability to avoid unexpected effects from modifications was influenced positively by TDD practice within the Agile-driven development project.

Hypothesis 8

$H_{8o}: \beta_8 = 0$: Refactoring has no influence on the Software Stability.

: Refactoring has a positive influence on the Software Stability.

From the hypotheses tests, the alternative hypothesis was accepted. The coefficient for refactoring had a *p* value of <0.001 and *t* value at $t_{(57)}$ of 5.29 measured at a 95%-confidence level. The *t* value for REFR variable was above the *t* critical value. The test of elasticity for the coefficient of refactoring has a positive relationship with software stability. That is, when REFR increases by 1%, SS increases by approximately 0.19%.

Hypothesis 9

H_{90}: $\beta_9 = 0$: CI has no influence on the Software Stability.

: CI has a positive influence on the Software Stability.

For these hypotheses tests, the null hypothesis was not rejected because the regression analysis generated a *t* value of 0.22 that is lower than critical value of 1.67. Also, the *p* value of .826 was also higher than the 5% allowed for this type of test. At the same time, the null hypothesis was not rejected $(\beta_9 = 0)$ because the variable could be accepted when tested at a lower confidence level. The direction of the sign in the test of hypotheses 9 was positive, meaning that when a CI increases software stability increases. However, due to the statistical insignificance of CI, there is no meaningful elasticity explanation.

Hypothesis 10

H_{10_0}: $\beta_{10} = 0$: TDD has no influence on the Software Testability.

: TDD has a positive influence on the Software Testability.

From the final hypotheses tests, the alternative hypothesis was also accepted. The coefficient for TDD had a *p* value less than 0.001 and *t* value at $t_{(57)}$ of 16.44 measured at a 95%-confidence level. The *t* value was above the *t* critical of 1.67. The test of elasticity for the coefficient of TDD has a positive relationship with software testability. That is, when TDD increases by 1%, ST increases by approximately 0.54%. In summary, software capability to enable modified software to be validated was influenced positively by TDD practice within the Agile-driven development project.

Hypothesis 11

$H_{11o}: \beta_{11} = 0$: Refactoring has no influence on the Software Testability.

: Refactoring has a positive influence on the Software Testability.

From the hypotheses tests, the alternative hypothesis was accepted. The coefficient for refactoring had a *p* value of <0.001 and *t* value of 4.66 at $t_{(57)}$ measured at a 95%-confidence level. The *t* value was above the *t* critical of 1.67. The test of elasticity for the coefficient of refactoring has a positive relationship with software testability. That is, when REFR increases by 1%, SC increases by approximately 0.16%. Software capability to enable modified software to be validated or tested was influenced positively by refactoring practice within the Agile-driven development project.

Hypothesis 12

$H_{120}: \beta_{12} = 0$: CI has no influence on the Software Testability.

: CI has a positive influence on the Software Testability.

For these hypotheses tests, the null hypothesis was not rejected because the regression analysis generated a *t* value of -0.004 less than critical value of 1.67. Also, the *p* value of .998 was also higher than the 5% allowed for this type of test. The alternative hypothesis that there exists a relationship between continuous integration and software testability was rejected. At the same time, the null hypothesis was not rejected $(\beta_9 = 0)$ because the variable could be accepted when tested at a lower confidence level. The direction of the sign in the test of hypotheses 12 was negative, meaning that when CI increases software testability decreases. However, due to the statistical insignificance of CI, there is no meaningful elasticity explanation.

Hypothesis 13

H_{13_o}: $\beta_{13} = 0$: TDD has no influence on the resultant weighted Software Maintainability.

: TDD has a positive influence on the resultant weighted Software Maintainability.

From the final hypotheses tests, the alternative hypothesis was also accepted. The coefficient for TDD had a *p* value of <0.001 and *t* value of 14.30 at $t_{(57)}$ measured at a 95%-confidence level. The *t* value was above the *t* critical value of 1.67. The test of elasticity for the coefficient of TDD has a positive relationship with software maintainability. That is, when TDD increases by 1%, ST or software maintainability increases by approximately 17.02%. In summary, software capability of a software system or component that can be understood, modified to correct faults, improve

performance or other attributes, or adapt to a changed environment was influenced positively by TDD practice within the Agile-driven development project. Note that higher the maintainability of the software, the lower is the maintenance efforts and cost incurred by IT management to maintain it.

Hypothesis 14

$H_{14o}: \beta_{14} = 0$: Refactoring has no influence on the Software Maintainability.

: Refactoring has a positive influence on the Software Maintainability.

From the hypotheses tests, the alternative hypothesis was accepted. The coefficient for refactoring had a *p* value of <0.001 and *t* value at $t_{(57)}$ of 8.70 measured at a 95%-confidence level. The *t* value was above the *t* critical of 1.67. The test of elasticity for the coefficient of refactoring has a positive relationship with software maintainability. That is, when REFR increases by 1%, SC or software maintainability increases by approximately 10.68%.

Hypothesis 15

$H_{15O}: \beta_{15} = 0$: CI has no influence on the Software Maintainability.

: CI has a positive influence on the Software Maintainability.

For these hypotheses tests, the null hypothesis was not rejected because the regression analysis generated a *t* value of 0.23 that is lower than critical value of 1.67. Also, the *p* value of .819 was also higher than the 5% allowed for this type of test. The alternative hypothesis that there exists a relationship between continuous

integration and software maintainability was therefore rejected. At the same time, the null hypothesis was not rejected $(\beta_{15} = 0)$ because the hypotheses may be accepted when tested at a lower confidence level. The direction of the sign in the test of hypotheses 15 was positive, meaning that when CI increases, software maintainability increases. However, due to the statistical insignificance of CI, there is no meaningful elasticity explanation. Lastly, employing the results from ANOVA table and the interpretation of coefficient results, the analysis yielded the following regression equation:

SM (Y) = 537.86 + 17.00 * TDD + 10.70 * REFR + .29 * CI

The CI is not a statistically viable predictor in this model.

Summary of Findings

The research results indicate that TDD and REFR variables that characterized ASDM, with the exception of the CI variable, are significant predictors of software analyzability, changeability, stability, testability, and resultant maintainability for the software developed using Agile development approach. The model explains 82.8% of the variability in software maintainability from the independent variables regressed. Further, the CI variable was not a statistically significant predictor of software maintainability in the generated regression model.

In conclusion, TDD and REFR practices do play a significant role in the way software maintainability and constituting characteristics gets inbuilt during the Agile software development process. Note that software maintainability is the internal quality

attribute of the software that has potential compounding effect on external software quality characteristics. Shaping the maintainability of the software product during the Agile development iteration process itself may now be possible with given findings of this study. The results of the best regression model indicated a TDD elasticity of 17.05 and REFR elasticity of 10.68, meaning that a 1% change in TDD within the Agile development approach will result in an approximate 17.05% change in software maintainability and that a 1% change in REFR efforts within the Agile development approach will result in an approximate 10.68% change in software maintainability.

Following the presentation of the results in this chapter, and the statistical significance of the results, summary of the study is provided in Chapter 5. This summary discusses the relations of the tested hypotheses and results of the study with its implication to IT management, Agile project management, and application maintenance management. It also recommends further study avenues.

Chapter 5: Summary, Conclusions, and Recommendations

Overview of the Study

This quantitative study was conducted in an attempt to understand and determine the impact of ASDM on software maintainability, which constitutes software analyzability, changeability, stability, and testability. Incorporating the software evolution framework in this study was an extension of prior studies that leveraged external quality attributes of software. This study, on the contrary, leveraged internal software quality notion and maintainability specific sub-characteristics to understand whether the Agile software development model impacts the critical software quality attribute: software maintainability. Integrating software maintainability objectives within the software development approach allows the IT management to attain healthy software life cycle as lower software maintainability often plagues the organizational profitability and productivity. Agile software development model thus holds the potential to address this ongoing challenge of higher maintenance cost for IT organizations.

Higher maintainability for software is a desired quality attribute. However, there was minimum empirical evidence on the extent of influence of ASDM on software maintainability and its subcharacteristics, and this gap led to this study. The reason this study was pursued was that growing software maintenance costs continue to paralyze the efficacy of IT as well as core business organizations. Furthermore, Agile and non-Agile practitioners within IT management lacked understanding about the impact of Agile approach on maintainability hindering their abilities to leverage Agile to actually

influence the software maintainability and underlying attributes namely, software analyzability, changeability, stability, and testability. Prior to the emergence of Agile as a software development model, many researchers (Banker, 1993; Bennett & Rajlich, 2000; Takang & Grubb, 1996) had raised concerns related to software maintenance cost that continued to increase because of poor maintainability inherent within the software. With Agile model's proliferation into software engineering and the practitioner's world of development, businesses continue to leverage this approach for multifold reasons. The underlying Agile practices such as test-driven development, refactoring, pair programming and iterative and continuous delivery of working software, offer numerous benefits known to the core business organizations. The impact of these practices on the internal quality attributes such as maintainability, however, was largely unknown.

The study was needed because there has been little research conducted on software maintainability within the context of ASDM-driven software life cycle. Partly this is due to a paucity of real life-Agile software data. This little research has been relatively recent (Sindhagatta, 2010) and it has limited focus primarily on the validation of Lehman's laws of software evolution. Another study, conducted by Bhadauria (2009) within an academic setting, focused on understanding the impact of test driven development on software quality, learning, and task satisfaction. Bhat and Nagappan (2006) on the other hand, examined the impact of TDD in industry setting on software quality in terms of reduced defects and improved test coverage. Additionally, research to understand software maintainability from the non-software development life cycle

(SDLC) perspective has been conducted by Moser et al. (2007) and Chen and Huang (2009). These researchers used postdevelopment metrics and employed external quality notion to understand software maintainability in relation to software development by estimating the effect of development efforts. Coram and Bohner (2005) studied the impact of Agile development model on software project management itself, which potentially may improve the success of the software development project. Software maintainability was not part of their impact assessment.

The effect of refactoring on software quality was also studied by several researchers (Alshayeb, 2009; Du Bois, Demeyer, Verelst, Mens, & Temmerman, 2006) showing its positive influence on software quality characteristics including maintainability, reusability, and testability. Geppert, Mockus, and Rossler (2005) also studied the effect of refactoring on changeability and reported that the changeability efforts and defect rate were reduced. This study built upon such prior Agile factor-specific research by incorporating the theoretical frameworks of Lehman's software evolution and maintenance laws. The aim was to understand whether Agile software development approach characterized by its key attributes such as TDD, refactoring, and CI influenced software maintainability characteristics to enhance internal quality of software. This approach is a departure from previous Agile and maintainability literature that has primarily focused on the relationship between Agile factors on software defects in the post deployment phase, impact assessment within non-Agile software development projects, and even open source software development environment.

Research Conclusions

To achieve the goals of this study, the following research questions were statistically examined and answered. What is the statistically significant impact of Agile software development approach or model and software maintainability characteristics?

1. How does the Agile software development model (ASDM), which is characterized by Test driven development (TDD), Refactoring (REFR), and Continuous Integration (CI), impact software analyzability (SA)?
2. How does the Agile software development model (ASDM), which is characterized by Test driven development (TDD), Refactoring (REFR), and Continuous Integration (CI), impact software changeability (SM)?
3. How does the Agile software development model (ASDM), which is characterized by Test driven development (TDD), Refactoring (REFR), and Continuous Integration (CI), impact software stability (SS)?
4. How does the Agile software development model (ASDM), which is characterized by Test driven development (TDD), Refactoring (REFR), and Continuous Integration (CI), impact software testabiliy (ST)?

Lastly, the following research question is also answered through the examination of impact on resultant maintainability characteristic that is a function of SA, SC, SS, and ST.

5. How does the Agile software development model (ASDM), which is characterized by Test driven development (TDD), Refactoring (REFR), and

Continuous Integration (CI), impact resultant weighted software maintainability (SM)?

The study provided a quantitative analysis attempting to measure the impact of the Agile development approach on the software analyzability, changeability, stability, and testability to achieve a higher software maintainability. Based on the results in this study, it is clear that the Agile development model is significantly related to software maintainability changes and hence it bears the potential to influence this key software quality attribute within the development phase itself. When the software revisions were analyzed for the changes in the source code properties such as complexity, coupling, duplication, unit size, and unit test efforts; the results has shown that software analyzability, changeability, stability, testability, and hence software maintainability were influenced positively by test driven development and refactoring techniques. The continuous integration variable was found to be insignificant predictor of software maintainability and all its subcharacteristics.

The results in this study do confirm the findings by Janzen and Saiedian (2005) and Bhat and Nagappan (2006), who examined TDD variable in specific. Alshayeb (2009) and Du Bois et al. (2006) found that refactoring improved the maintainability including changeability within Agile development project. There was no study conducted previously to assess the influence of continuous integration practice on software maintainability. This practice, however, is primarily followed to improve the software integrity and assist the Agile developers to test the build on continuous basis

before rolling out to the production system. The study findings suggested that this practice does not influence the maintainability or any of the underlying characteristics. The higher the test driven development, and refactoring efforts undertaken to reduce the complexity for high-risk category of the software classes, the more likely the software would increase its maintainability. These results are also in line with Sindhgatta et al. (2010), who posited that iterative software development nature in Agile-driven development projects allows the development team to influence the complexity of the software and thus improve the maintainability. However, this study contrasts to the concerns from the panel of Majko-Mattson et al. (2006), who asserted that Agile development approach could potentially jeopardize the software quality. The results conform to underlying objectives of Agile approach suggested by Cockburn (2002), that Agile-driven development sought to improve the software quality besides attaining higher business value for the IT organizations through early delivery of the software. Finally, the results also align with the findings of Giblin, Brennan, and Exton (2010) that Agile methods guide the developers to produce the software code with better maintainability characteristics.

Answering the Research Questions

The analysis and results presented in Chapter 4 provided sufficient statistical evidence to answer the research questions as follows:

Question 1

How does the Agile software development model (ASDM), which is characterized by Test driven development (TDD), Refactoring (REFR), and Continuous Integration (CI), impact software analyzability (SA)?

The regression model (Table 18) illustrated a strong statistical relationship between TDD and REFR, and SA. The results showed that the alternative hypothesis should be accepted because of the statistical significance of SA in relation to TDD and REFR. This means that a positive increase in TDD and REFR translates into higher software analyzability. However, the CI did not influence SA. This Agile practice is mainly used to allow the developers to improve the code integration during the earlier stage of the development. The SA is the capability of the software system to allow the modifications, and it signifies the ease with which the analysis can be done before making the software modifications. TDD enforces the programmers to write and test the unit tests as the actual code gets developed. This practice makes the software code manageable and easier to analyze before applying the identified software modifications. Similarly, the REFR efforts reduce the overall code complexity, further improving software analyzability. It is noteworthy that refactoring explained .30% of the changes compared to TDD, which explained .19% of the changes in analyzability, which is in harmony with the theoretical premise of refactoring objectives within the Agile development approach. CI practice has no statistically significant relationship with SA.

This research question tests the assertion of increasing software complexity when software evolves unless efforts are undertaken to reduce its complexity. Agile factors such as REFR allow the developers to reduce the code complexity without affecting its functionality, and in turn improve the analyzability throughout Agile iterations. In congruence with this assertion, the results support the theory that the software system could become less complex when development teams work on it as it evolves. The results illustrated that SA is positively related to TDD and REFR, whereas CI attribute of the ASDM does not influence analyzability.

Question 2

How does the Agile software development model (ASDM),which is characterized by Test driven development (TDD), Refactoring (REFR), and Continuous Integration (CI), impact software changeability (SC)?

The regression model (Table 20) illustrated a strong statistical relationship between TDD and REFR and software changeability. The results showed that the alternative hypothesis should be accepted because of the statistical significance of SC in relation to TDD and REFR. This means that a positive increase in TDD and REFR translates into higher SC. However, the CI did not influence SC. This Agile practice is mainly used to allow the developers to improve the code integration during the earlier stage of the development. The SC is the capability of the software system to implement the modifications, and it signifies the ease with which the change implementation can be done. TDD enforces the programmers to write and test the unit tests as the actual code

gets developed. This practice further improves the software ability to change when implementing the identified software changes. Similarly, the REFR efforts reduce the code complexity, in turn improving SC. It is noteworthy that TDD explained .40% changes compared to REFR, which explained .25% change in changeability, which is in alignment with the objectives of TDD practice within the Agile development approach. TDD improves the capability of the software system to implement the change through writing the test in conjunction with code development efforts. CI practice has no statistically significant relationship with SC.

This research question tests the assertion of increasing software complexity when software evolves unless efforts are undertaken to reduce its complexity. Agile factors such as REFR allow the developers to reduce the code complexity without affecting its functionality, and in turn improve the analyzability throughout Agile iterations. In congruence with this assertion, the results support the theory that the software system could become less complex when development teams work on it as it evolved. The results illustrate SC is positively related to TDD and REFR, whereas the CI attribute of ASDM does not influence changeability.

To conclude answering Question 2, ASDM that is characterized by TDD and REFR is positively related with SC with an exception of CI. This means that the more TDD and REFR practices are adhered within Agile development approach, the higher the propensity to improve the SC.

Question 3

How does the Agile software development model (ASDM), which is characterized by Test driven development (TDD), Refactoring (REFR), and Continuous Integration (CI), impact software stability (SS)?

The regression model does highlight a statistically significant relationship between Agile development approach and software stability. This model was regressed for the entire data set of 61 software iterations.

The regression model shown in Table 22 illustrated a strong statistical relationship between TDD and REFR and SS. The results showed that the alternative hypothesis should be accepted because of the statistical significance of SS in relation to TDD and REFR. This means that a positive increase in TDD and REFR translates into higher SS. However, the CI did not influence SS as indicated earlier. This Agile practice is mainly used to allow the developers to improve the code integration during the earlier stage of the development. The SS is the capability of the software system to avoid unexpected effects from modifications of the software. It also denotes the ease with which the software can be maintained in stable and consistent stage after the change implementation. TDD practice enforces the programmers to write and test the unit tests as the actual code is developed. This practice further improves the software ability to stabilize after implementing the identified software changes through test-ahead of coding protocol. Similarly, the REFR efforts reduce the code complexity, in turn improving SS. Less complex classes are easy to change, and hence inform better stability after the

change is implemented. It is noteworthy that TDD explained .42% of the changes compared to REFR, which explained .19% change in stability, which is in alignment with the objectives of TDD practice within the Agile development approach. TDD improves the capability of the software system to stabilize the state of the software after the change implementation because the accomplished change get tested well through writing the test in conjunction with code development efforts. Any failed test would indicate possible issues early and would stop programmers from implementing the change itself. Lastly, CI practice has no statistically significant relationship with SC. However, this Agile practice is actively adhered during the build testing efforts that occur prior to deployment to nondevelopment environment. The insignificance of CI within this regression model does not alter or undermine its relevance in actual Agile development projects.

To conclude answering Question 3, ASDM that is characterized by TDD and REFR is positively related with SS with an exception of the CI variable. This means that the more TDD and REFR practices are adhered within Agile development approach, the higher is the capability of the software to stabilize and maintain the consistent state after the change implementation. This capability or SS is influenced by ASDM positively.

Question 4

How does the Agile software development model (ASDM), which is characterized by Test driven development (TDD), Refactoring (REFR), and Continuous Integration (CI), impact software testabiliy (ST)?

When the ST was regressed for the ASDM variables, the regression model (Table 24) showed a strong statistical relationship between a TDD as well as REFR variables and ST. The TDD practice has a positive relationship with testability or the capability of the software to validate the implemented change. This means that when developers write a test class to test the actual code written in given software class, it becomes easy for the developers to validate the change that eventually are implemented. The TDD influenced .54% of variation in testability, whereas REFR accounted for .16% of variation. More efforts spent on refactoring related to reducing the complexity of the classes that are higher than 20 McCabe CC, also influenced the testability positively. The overall strength of this regression model was strong, with 83% indicating higher coefficient of determination.

Question 5

How does the Agile software development model (ASDM), which is characterized by Test driven development (TDD), Refactoring (REFR), and Continuous Integration (CI), impact resultant software maintainability (SM)?

Lastly, resultant SM was computed by adding weighted dependent variables (analyzability, changeability, stability, and testability) and its final value was regressed for TDD, REFR, and CI over 61 Agile iterations or revisions. The regression model (Table 27) indicated that SM was influenced positively by all three variables. The CI variable, however, was statistically insignificant in explaining the variation on SM due to higher *p* value of .819. The TDD explained 17% of variation, whereas the REFR

explained 11% of variation in SM with 82.8% as coefficient of determination or strong strength of regression model. An ASDM that is characterized by TDD and REFR does impact the resultant SM positively. The CI variable was found to be statistically insignificant predictor of SM indicating that observed variation in maintainability cannot be attributed to variation in CI.

Implications

The summarized results have shown that ASDM is statistically significant predictor of SM. True to form of a quantitative study, the U.S.-based IT organization's data was used to evaluate how ASDM influences software complexity, coupling, duplication, and testing efforts, which in turn influence maintainability characteristics, employing a multiple regression technique. The results obtained from the multiple regression models were mixed. For the selected software system that was developed using the Agile approach, TDD and REFR were statistically significant contributors or predictor variables; however, CI was found to be an insignificant variable. The finding are in congruence with the arguments by Beck (2003), Highsmith (2004), and Fowler (1999) that TDD and REFR practices allow the developers to write less complex, loosely coupled code that is testable and stable. The Agile model also promotes a simple design technique that is also evident in the smaller method creation during the development. Based on the findings from this study, there are various implications to IT management, Agile driven software development project managers, Agile practitioners, and software support and maintenance organizations.

Implications to IT and Business Management

The first implication from this study is the assessment of the impact of Agile approach and its specific three variables on key software quality attribute: SM. Without this much-needed assessment, IT management may not confidently advocate Agile to the business organizations they serve. Additionally, without knowing how the Agile approach can control and influence key maintainability attributes during the software development process, IT management may fail to fully leverage the Agile model to actually control and reduce the software maintenance efforts within their relevant business domains. As articulated in the Need for the Study in Chapter 1, the recent growth in Agile adoption has been purely fueled by its highlighted benefits to the organizations in swift change management and adoption with changing business processes. IT and business organizations, however, may incur systemic maintenance costs due to inadequate knowledge and poor understanding about Agile's implications on maintainability-related attributes. This study provides a quantitative and lightweight assessment model that can be incorporated into Agile-driven projects to improve the alignment between Agile practices and maintainability objectives. This is major implication to IT management as this study indicated that ASDM is a statistically significant predictor of software maintainability during the Agile-driven iterations of development.

The results in Chapter 4 also revealed that TDD and REFR practices are good predictors of SM and its all subcharacteristics: analyzability, changeability, stability, and testability. Knowing how TDD and REFR impacts various software internal quality attribute levels can help IT management, including Agile project management, improve the quality profile of application management portfolios by identifying software applications that could potentially be improved over time for their maintainability.

Understanding these specific software applications could also potentially result in quality-loss mitigating actions such as focused Agile development projects geared towards the improvement of business critical application plagued with poor maintainability. These could serve as potential remedial actions against high maintenance efforts buildup and economic strife due to high maintenance cost. In summary, the ability to better identify software maintainability characteristics that are highly sensitive to Agile factors could potentially result in mitigation of the software quality related risks.

Implications to Agile Project Management

The second implication of this study pertains to the Agile software development project management. The Agile PM leadership and project sponsor organizations will better understand how Agile factors influence the internal software quality characteristic during the development iteration cycles. In 2009, the Standish group reported that software development projects often struggled on achieving the required software quality, impairing the overall efficacy of IT organizations that often need to maintain the software

for many years. When attempting to deliver the development project on time and within budget, PM leadership may not always succeed to produce the acceptable quality of the software. More specifically, within Agile development projects, Chow and Cao (2008) identified three key success factors, one of which is Agile software engineering practices besides delivery strategy and team capability. This study strengthens the need of integrating maintainability goals within Agile development and reiterate the significance of key Agile practices within the context of maintainability. Software functionalities continue to grow rapidly within today's business landscape, further demanding higher reliability and availability of the software. An Agile development model can play a crucial role in ensuring that organizations achieve these higher quality norms with well-informed Agile project management leadership.

Understanding the impact of Agile factors could result in incorporating realistic software maintainability goals into software development projects to explicitly target long-term software quality influencing metrics and their proactive measurement. Project management leadership may actively work on the quantifiable maintainability assessment throughout the Agile development iterations in favor of the customers as well as IT organization itself.

Implications to Software Maintenance Management

The third implication of this study is the affirmative effect on software maintenance management including IT organizations that are responsible to sustain the post-development software life cycle. Software engineering practitioners and researchers

have largely argued that lack of maintainability considerations within software design (Kyte, 2011; Pigoski, 1997) often increases software maintenance cost. Agile proponents attested to its expeditious system delivery (Paulk, 2001), but some researchers argued that this process-light approach could potentially postpone the system problem by breaching good software development principles (Khan, 2003). The findings in this study revealed the positive correlation between Agile software development approach and software maintainability, strengthening the case for Agile-driven development approach to proactively reduce the futuristic maintenance cost. In specific, this study has shown that TDD and REFR practices contribute to improved maintainability of the software. This inference further suggests the possibilities of bridging the well-known gaps between development and maintenance organizations. Lastly, precious IT resources may be reallocated to software development efforts as well as higher business value yielding initiatives rather than maintenance tasks due to improved software maintainability.

Limitations of the Study

When conducting this study, some limitations were encountered, particularly in regard to the availability of viable software systems that were developed using Agile approach. This study was conducted on a single software system developed using the Agile model within a single IT organization. Secondly, as detailed in the data manipulation section of Chapter 4, the software system data was missing partial values for one of the metrics: test coverage, related to unit test efforts. Missing data was replaced with the data values computed using expectation maximization (EM) technique.

The final data analyzed was statistically significant, at 61 software revisions. Perhaps having a larger sample would produce different results. The study analyzed the software classes and not the database related attributes. Lastly, the data came from 2010-2011 periods for a specific project. The results obtained in this quasi-experimental quantitative study may likely be different from other IT organizations with varied levels of Agile competency and maturity.

Significance of the Study and Implications for Social Change

On a larger scale, this study is significant because it has the potential to aid IT application management to better understand quality traits of their application portfolios by closely analyzing the impact of Agile development model on software maintainability characteristics. As part of the new business process integration and new software development initiatives; IT software program management and business managers routinely examine application portfolios. These reassessments attempt to scientifically quantify the potential impact of unhealthy software applications on the IT organizational overhead and the organization's returns. To support the claims of risk mitigation, this study used empirical evidence (presented in Chapter 4) to demonstrate that the Agile development model is a significant predictor of software maintainability. The study has also demonstrated how ASDM influences various subsets of the software maintainability. Such knowledge has the potential to help IT management and Agile teams to better develop their software while maximizing end-to-end business value.

The results detailed in Chapter 4 may provide Agile advocacy groups with a scientifically backed analysis for software projects to influence software maintainability characteristics, specifically for complex projects. The ability to tune Agile factors within the Agile projects has the potential of substantially lowering software maintenance efforts, which could eventually mitigate the transfer of systemic risk into the software life cycle and lead to a stable and reliable software applications supporting the business economy.

Finally, based on the existence of a statistical relationship between the Agile development approach and software maintainability within IT organization in the United States, I assert that integrating software maintainability-related objectives within Agile development projects will further bolster Agile adoption as well as improve the business value of IT organization. Agile development teams should sharpen the focus on the underlying key Agile practices that influences the internal quality characteristics of the software. Additionally, with proactive measurement of software maintainability throughout the development iterations, Agile project management is now positioned to steer the development team well ahead of costly maintenance efforts. Lastly, IT operation and application maintenance teams can embrace Agile more confidently within their organizations to mitigate the software quality concerns within maintainability realms. The application of these findings will potentially lead to healthy software application life cycles, resulting into more reliable software product and efficient business organizations.

Recommendations for Future Study

Several recommendations for future research emerge from this study. First, the data from the Agile-driven software project repository may further be merged with postdevelopment maintenance efforts, maintenance incident, and productivity-gain related data. This is to determine whether ASDM can in turn improve the ability of IT management to enhance the application life cycle health, improve business value through higher maintainability, and efficient resource allocation within software development organizations. The process of continuous proactive assessment could be revolutionary as it has the potential to identify specific Agile factor and tune it precisely within active iteration or cycle for better maintainability goals. Agile teams could potentially optimize Agile-specific practices in order to improve software maintainability.

The TDD explained 17% of the variation in software maintainability within the sampled Agile project. REFR explained 10.68% of the variation, whereas CI was not found to be a significant variable and did not explain statistically significant variation in any of the maintainability sub-characteristics. Future studies could incorporate the effect of pair programming practice into the model to determine whether it is a significant driver of software maintainability and any of its subcharacteristics. Additional studies may examine the indirect influence of CI practice or attribute on software maintainability using other acceptable measure than the count of successful builds as a measure of CI.

Using varied data from several project teams rather than single software project-specific data could also provide more insight into whether maintainability has been

changing among software applications, whose development project teams and leadership vary. One other avenue of future research is to conduct studies in other IT organizations within the United States and outside of the United States. In summary, this study paved the way for additional research that may incorporate the tenets of software maintainability to be conducted in additional agile business domains, additional software development projects, and maintenance organizational setting integrating ASDM. This study also provided a foundation for analyses of other project success factors within Agile driven development; factors such as customer satisfaction based on the software maintenance quality, reduction in the software maintenance cost, lower software maintainability, and software maintenance team's performance.

Study Conclusion

This study provides statistically significant findings that TDD and REFR practices within the Agile model of software development influence software maintainability and its sub-characteristics positively. TDD specifically influenced the software testability the most compared to other attributes of maintainability. REFR on the other hand influenced analyzability the most compared to all other attributes of maintainability. These conclusions are well in alignment with core Agile practice intentions. For instance, TDD is targeted to improve the ease of the software to be tested after the modification is done. REFR is followed to simplify the design and reduce the code complexity that in turn yield the improvement in the east with which software can be analyzed before the change is made. Furthermore, it also raises some interesting questions. For instance, why does

TDD within Agile development improve the analyzability or the ease with which programmers can understand the specific part or module that needs to be changed? The answer is difficult to determine using the data used in this study. However, it may be inferred that the TDD approach allows the development team to focus on testing efforts through unit test creation and its successful execution, that also in turn allow them to improve their ability to identify the specific section or sections of the program. This ease in the identification of specific module or component of the program improves the software analyzability. Additionally, the REFR efforts may even be resulting from the failed testing that needed some additional REFR efforts to reduce the complexity. The reduced complexity then further improves the software analyzability as well. CI practice is critical at the code creation & integration stage to avoid possible rework. It also builds the developer's confidence incrementally. With given data, the findings didn't establish its direct influence on maintainability. The results of this study strengthens the confidence of IT management in Agile practices further as they continue to address the challenges of increasing software maintenance efforts and cost. The biggest social impact of this study is its ability to provide IT management with an additional assertion that could be used to improve the alignment between software maintainability objectives and Agile practices. This improved alignment could further solidify business value of Agile approach through qualitative, productive, and cost-effective IT services. Potential reduction in software maintenance efforts and cost is a big win for organizations, resulting from a higher quality of software that is developed using the Agile approach.

References

Aczel, A. D. (2006). *Complete business statistics* (6th ed.). New York, NY: McGraw-Hill.

Agile Alliance. (2005). *Manifesto for Agile software development.* Retrieved from http://www.agilealliance.org

Ahern, K. & Brocque, R. (2005). Methodological issues in the effects of attrition. *Sage Journal of Field Methods*, *17*(1), 53-69. doi: 10.1177/1525822X04271006

Ahmed, R. N. (2006). Software maintenance outsourcing: Issues and strategies. *Computers & Electrical Engineering*, *32*(6), 449-453. Retrieved from http://dx.doi.org/10.1016/j.compeleceng.2006.01.023

Alkhatib, G. (1992). The maintenance problem of application software. An empirical analysis. *Journal of software maintenance–Research and Practice*, *4(2)*, 83-104. doi: 10.1002/smr.4360040203

Alshayeb, M., & Li, W. An empirical validation of object-oriented metrics in two different iterative software processes. *IEEE Transactions on Software Engineering*, *29*(11), 1043-1049. doi: 10.1109/TSE.2003.1245305

Apfelbaum, L., & Doyle, J. (1997). *Model based testing. Proceedings of 10th International Software Quality Week Conference.* San Francisco, CA: Software research institute.

Arnold, R. S. & Parker, D. A. (1982). The dimensions of healthy maintenance. *Proceedings of the 6th International Software Engineering Conference*,10-27. IEEE Computer Society Press. Los Alamitos, CA.

Arthur, L. J. (1988). *Software evolution: The software maintenance challenge*. New York, NY: John Wiley and Sons.

Augustine, S. (2005). *Managing Agile projects* (1st ed.). Lebanon, IN: Prentice Hall Publication.

Bagheri, E., & Gasevic, E. (2011). Assessing the maintainability of software product line feature models using structural metrics. *Software Quality Journal*, 19(30), 579-612 doi: 10.1007/s11219-0109127-2

Balci, O. (2003). Verification, validation and certification of modeling and simulation applications. *Proceedings of the 2003 Simulation Conference, 1*, 150-158. doi: 10.1109/WSC.2003.1261418

Banker, R. D., Datar, S. M., & Kemerer, C. F. (1991). A model to evaluate variables impacting the productivity of software maintenance projects. *Management Science, 37*(1), l-18. doi: 10.1287/mnsc.37.1.1

Banker, R. D., Datar, S. M., Kemerer, C. F., & Zweig, D. (1993). Software complexity & software maintenance costs. *Communications of the ACM, 36*(11), 81-94. doi: 10.1145/163359.163375

Banker, R. D., Davis, G. B., & Slaughter, S. A. (1998). Software development practices, software complexity, and software maintenance performance: A field study. *Management Science, 44*(4), 433-450. doi: 10.1287/mnsc.44.4.433

Basili, V. R., & Perricone, B. (1984). Software errors and complexity: An empirical investigation. *Communications of the ACM, 27*(l), 42-52. doi: 10.1145/69605.2085

Beck, K. (1999a). Embracing change with extreme programming. *IEEE Computer Journal, 32*(10), 70–77. doi: 10.1109/2.796139

Beck, K. (2000). *Extreme programming explained: Embrace change*. San Francisco, CA: Addison-Wesley.

Beck, K., & Andres, C. (2005). *Extreme programming explained.* (2nd ed.). Boston, MA: Addison Wesley.

Berry, W. D., & Feldman, S. (1985). Multiple regression in practice. *Sage University Paper Series on Quantitative Applications in the Social Sciences, 7*(50), 16-20. Newbury Park, CA: Sage

Beck, K. (1999). *Extreme programming explained. Embrace change.* Reading, MA: Addison-Wesley.

Bendifallah, S., & Scacchi, W. (1987). Understanding software maintenance work. *IEEE Transactions on Software Engineering, 13*(3), 311-323. doi: 10.1109/TSE.1987.233162

Bennet, K. H. (1990). An introduction to software maintenance. *Information and Software Technology*, *12*(4), 257-264. Retrieved from sciencedirect - Elsevier

Bendifallah, S., & Scacchi, W. (1987). Understanding software maintenance work. *IEEE Transactions on software engineering, 13*(3), 311-323. doi:10.1109/TSE.1987.233162

Bellini, E., Canfora, G., Garcia, F., Piattini, M., &Visaggio1, C.A. (2005). Pair designing as practice for enforcing and diffusing designs knowledge. *Journal of Software Maintenance and Evolution: Research and Practice, 17*, 401-423. doi: 10.1002/smr.322.

Bhatt, P., & Shroff, G.(2004). Dynamics of software maintenance. *ACM SigSoft Software Engineering Notes, 29*(5), 1-5. doi: 10.1145/1022494.1022513

Bhatt, P., Shroff, G., Anantaram, C., & Misra, A. K. (2006). An influence model for factors in outsourced software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice, 18*,385-423. doi: 10.1002/smr.v18:6

Boehm, B., & Turner, R. (2004). *Balancing agility and discipline: A guide for the perplexed*. Boston, MA: Addison-Wesley.

Boehm, B. W., Brown, J. R., Kaspar, H., Lipow, M., McLeod, G., & Merritt, M. (1978). *Characteristics of software quality*. North Holland: Elsevier Science.

Boehm, B. W., Brown, J. R., & Lipow, M. (1976). Quantitative evaluation of software quality. *In the Proceedings of the 2nd International Conference on Software Engineering*, 592-605. Retrieved from IEEE Computer Society Press. Los Alamitos, CA.

Brooks, F. (1995). *The mythical man-month*. Reading, MA: Addison-Wesley.

Briand, L., Bunse, L., Daly, J., &Differding, C. (1997). An experimental comparison of the maintainability of object-oriented and structured design documents. *Empirical Software Engineering*. Retrieved from at: http://citeseer.nj.nec.com/12374.html.

Broy, M., Deissenboeck, F., & Pizka, M. (2006). Demystifying maintainability. *In the Proceedings of the International workshop on Software quality*, *ACM*, *21-26*. doi: 10.1145/1137702.1137708

Burns, N., & Grove, Su.K. (1993). *The practice of nursing research: Conduct, critique & utilization* (2nd ed.). Oxford: W.B. Saunders.

Cao, L., Balasubramaniam, R., Tarek Abdel-Hamid (2010). Modeling dynamics in agile software development. *ACM Transaction Management Information system,* 1(1):5 doi.10.1145/1877725.1877730

Chapin, N. (1989). An entropy metric for software maintainability. *Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences*, 522-523. doi: 10.1109/HICSS.1989.48047

Charette, R.(2004). The decision is in: Agile versus heavy methodologies. *Agile Development and Project Management*. *Cutter Consortium, 2*(19). Retrieved from www.cutter.com/freestuff/epmu0119.html.

Changeau, D. (2004). Citizenship and constructing sense in voting: An experimental approach. *Conference Papers–American Sociological Association, 1*(1), 1-16.

Chidamber, S. & C. Kemerer (publication date). A metrics suite for object-oriented design. *IEEE Transactions on Software Engineering, 20*(6), 476-493.

Chiang, I. R., & Mookerjee, V. S. (2004). Improving software team productivity. *Communications of the ACM, 47*(5), 89-93. doi.:10.1145/986213.986217

Chen, J., & Huang, S. (2009). An empirical analysis of the impact of software development problem factors on software maintainability. *The Journal of Systems and Software*, *82*, 981-992. doi: 10.1016/j.jss.2008.12.036

Chidamber, S., Darcy, D. P., & Kemerer, C.F. (1998). Managerial use of metrics for object-oriented software: An exploratory analysis. *IEEE Transactions on Software Engineering, 24*(8), 629-639. doi: 10.1109/32.707698

Chidamber, S. R., & Kemerer, C.F. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering, 20*(6), 467-493. doi: 10.1109/32.295895

Chow, T., & Cao, D.(2008). A survey of critical success factors in agile software projects. *Journal of Systems and Software, 81*(6), 961-971. doi: 10.1016/j.jss.2007.08.020

Cockburn, A.(2006). *Agile software development*, *the cooperative game*. London: Pearson Education.

Cockburn, A. (2002). *Agile software development*. Boston, MA: Addison Wesley.

Cockburn, A. & Highsmith, J. (2001). Agile software development, the people factor. *IEEE Journals, 34*(11), 131-133. doi: 10.1109/2.963450

Conboy, K.,& Fitzerald, B. (2010). Method and developer characteristics for effective agile method tailoring: A study of XP expert opinion, *ACM Transactions on software engineering and methodology*, 20(1),1-29. doi: 10.1145/1767751.1767753

Correia, J. P., Kanellopoulos, Y., &Visser, J. (2009). A survey-based study of the mapping of system properties to ISO/IEC 9126 maintainability characteristics. *IEEE ICSM*. doi: 10.1109/ICSM.2009.5306346

Coram, M., & Bohner, S. (2005). The impact of Agile methods on software project management. *Engineering of Computer Based Systems*, 363-370. doi: 10.1109/ECBS.2005.68

Creswell, J. (2003). *Research design. Qualitative, quantitative, and mixed methods approaches* (2nd ed.). Thousand Oaks, CA: Sage publications.

Creswell, J. (2007). *Qualitative inquiry and research design: Choosing among five approaches*. Thousand Oaks, CA: Sage Publications, Inc.

Cugola, G., & Ghezzi, C. (1998). Software processes: A retrospective and a path to the future. *Software Process Improvement and Practice, 4*, 101-123. doi: doi: 10.1002/(SICI)1099-1670(199809)4:3<101::AID-SPIP103>3.0.CO;2-K

Das, S., Lutters, W., & Seaman, C. (2007). Understanding documentation value in software maintenance. *ACM*. doi: 1-59593-635-6/07/0003

Dethomas, A. (1987). Technology requirements of integrated, critical digital flight systems. In: *AIAA Guidance, Navigation and Control Conference, Monterey, CA, Technical Papers,* (Vol. 2), 1579-1583. American Institute of Aeronautics and Astronautics, New York.

Dekleva, S. (1992a). Delphi study of software maintenance problems. In: *Proceedings of the 1992 Conference on Software Maintenance. IEEE Computer Society*, 10-17. doi: 10.1109/ICSM.1992.242564

Dekleva, S. M. (1992b). The influence of the information systems development approach on maintenance. *MIS Quarterly*, *16*(3), 355-372. doi: 10.2307/249533

Evans, M. W. & Marciniak, J. (1987). *Software quality assurance and management*. New York: Wiley.

Fenton, N. E., & Ohlsoon, N. (2000). Quantitative analysis of faults and failures in a complex software system. *IEEE Transactions on Software Engineering, 26*(8), 797-814. doi: 10.1109/32.879815

Fowler, M. (2001). Reducing coupling. *IEEE Software, 18*(4), 102-104. doi: 10.1109/MS.2001.936226

Fowler, M., Beck, K., Brant, J., Opdyke, W., & Roberts, D. (1999). *Refactoring: Improving the design of existing code*. New Jersey: Addison-Wesley Professional.

Glass, R. L., & Noiseux, R. A. (1981). *Software maintenance guidebook*. Englewood Cliffs, NJ: Prentice-Hall.George, B., & Williams, L. (2004). A structured experiment of test-driven development. *Information and Software Technology, 46*, 337-342. doi: http://dx.doi.org/10.1016/j.infsof.2003.09.011

Gilb, T. (1988). *Principles of software engineering management*. London: Addision Wesley.

Giblin, M., Brennan, P., & Exton, C. (2010). Agile processes in software engineering and extreme programming. *Lecture Notes in Business Information Processing, 48*(1), 58-72. doi: 10.1007/978-3-642-13054-0_5

Gibson, V. and J. Senn. (1989). System Structure and Software Maintenance Performance. *Communications of the ACM,* 32(3), 347-358. doi: 10.1145/62065.62073

Gremiilion, L. L. (1984). Determinants of program repair maintenance requirements. *Management of Computing–Communication of ACM, 27*(8), 826-832. doi:10.1145/358198.358228

Grubb, P., & Takang, A. A. (2003). *Software maintenance: Concepts and practice* (2nd ed). NJ: World scientific publication.

Gyimothy, T., Ferenc, R. & Siket, I. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering, 31*(10), 897-910.

Hayes, J. H., Dekhtyar, A., Sundaram, S. K., Holbrook, E. A., Vadlamudi, S., & April, A. (2007). Requirements tracing on target (RETRO). Improving software maintenance through traceability recovery. *Innovations in Systems and Software Engineering*, *3*(3), 193–202. doi: 10.1007/s11334-007-0024-1

Hayes, J. F., Mohamed, N., & Gao, T. H. (2003). Observe-mine-adopt (MOA): An agile way to enhance software maintainability. *Journal of Software Maintenance and Evolution Research and Practice, 15*, 297-323. doi:10.1002/smr.287

Helms, G. L. & Weiss, I. R. (1985). Application software maintenance: Can it be controlled? *ACM Journal*, 16(2), 16-18.doi: 10.1145/1040688.1040691

Heitlager, I., Kuipers, T., &Visser, J.(2007). A practical model for measuring maintainability. *Sixth International Conference on the Quality of Information and Communications Technology*. doi: 10.1109/QUATIC.2007.8

Highsmith, J. (2002). *Agile software development e*cosystems. Boston, MA: Addison-Wesley Professional.

Highsmith, J. (2009). *Agile project management: Creating innovative products*. (2nd ed.). Boston: Addison-Wesley.

Holcombe, M., 2008. *Running an Agile software development project*. Hoboken, NJ: John Wiley & Sons, Inc.

Hordijk, W., & Wieringa, R. (2005). Surveying the factors that influence maintainability. *ACM Journal*, 385-388. doi: 1595930140/05/0009

Hoffer, J. A., George, J. F., & Valacich, J. S. (2008). *Modern systems analysis and design.* (5th ed.). Upper Saddle River, NJ: Prentice Hall.

Howcroft, D., & Trauth, E. (2008). *Handbook of critical information systems research-theory and application*. Cheltenham: Edward Elgar Publishing

Huffman, J. E., & Burgess C. (1988). Partially automated in-line documentation (PAID): Design and implementation of a software maintenance tool. *Proceedings Conference on Software Maintenance.* Los Alamitos, CA: IEEE Computer Society Press, 60-65. doi: 10.1109/ICSM.1988.10140

Huffman, H. J., & Offutt, A. J. (2000). *Product and process: Key areas worthy of software maintainability empirical study*. Sixth IEEE Workshop on Empirical Studies of Software Maintenance. Retrieved from http://members.aol.com/geshome/wess2000/janeHwess.pdf

Hulse, C., Edgeron, S.,Ubnoske, M.,& Vazquez, L.(1999). Reducing maintenance costs through the application of modern software architecture principles. *Proceedings of the annual ACM SIGAda International conference on Ada*,19(3),101-110,doi: 10.1145/319295.319311.

Huo, M., Verner, J., Zhu, L., & Babar, M. A. (2004). Software quality and Agile methods. *Proceedings of the annual International conference on Computer software and applications*. IEEE,1, 520-525. doi: 10.1109/CMPSAC.2004.1342889

IEEE. (2000). *Authoritative dictionary of IEEE standards terms*. New York: Author. IEEE Press.

IEEE. (1990). *IEEE standard computer dictionary*. A compilation of IEEE standard computer glossaries. doi. 10.1109/IEEESTD.1991.106963

ISO/IEC 9126-4 (2004). *ISO/IEC 9126-4 Software Engineering – Product Quality International Standard Quality in Use Metrics*. Geneva, Switzerland

ITIL Application Management. (2002). *ITIL application management.* London, England: The Stationary Office.

Janzen, D.S., & Saiedian, H. (2006).On the influence of test-driven development on software design. *Proceedings of 19th Conference on Software Engineering Education and Training,* 141-148. doi: 10.1109/CSEET.2006.25

Jensen, R.N., Platz, N., & Tjornehoj,G. (2008). Developer stories: Improving architecture In agile practice. *Proceedings of 2nd International conference on computer communication and information systems,* 22(2), 172–184. doi: 10.1007/978-3-540-88655-6_13

Jung, H., Kim, S., & Chung, C. (2004). Measuring software product quality: A survey of ISO/IEC 9126. *IEEE Software Journal*, 21(5), 88-92.doi: 10.1109/MS.2004.1331309.

Kafura D., & Reddy, R. (1987). The use of software complexity metrics in software maintenance. *IEEE Transactions on Software Engineering, 13*(3), 335-343. doi: 10.1109/TSE.1987.233164

Kaplan, S. (2002). Now is the time to pull the plug on your legacy apps. *CIO Magazine*. Retrieved from www.cio.com/archive/031502/infrastructure.html

Kajko-Mattsson, M., Lewis, G., Siracusa, D., Chapin, N.,Nocks, J., Sneed, H. et al. (2006). Long-term life cycle impact of Agile methodologies. *22nd IEEE International Conference on Software Maintenance,* 422-425.doi: 10.1109/ICSM.2006.34

Kan, S. H. (2003). *Metric and models in software quality engineering*. (2nd ed.). Boston, MA: Addison-Wesley Publication.

Kanellopoulos, Y., Antonellis, P., Antoniou, D., Makris, C.,Theodoridis, E., Tjortjis, C., Tsirakis, N. (2010). Code quality evaluation methodology using the ISO/IEC 9126 Standard. *International Journal of Software Engineering & Applications* (IJSEA), *1*(3),17-36. doi: 10.5121/ijsea.2010.1302

Kanellopoulos, Y., Heitlager, I., Tjortjis, C., & Visser, J. (2008). Interpretation of source code clusters in terms of the ISO/IEC-9126 maintainability characteristics. *Software maintenance and reengineering IEEE Conference.* 63-72. doi: 10.1109/CSMR.2008.4493301

Karlsson, J. B. R., & Wohlin, C. (1998). An evaluation of methods for prioritizing software requirements. *Journal of Information and Software Technology, 39* (14-15), 993-947. doi: http://dx.doi.org/10.1016/S0950-5849(97)00053-0

Kemerer, C. F., & Slaughter, S. (1999). An empirical approach to studying software evolution. *IEEE Transactions on Software Engineering*, *25*(4), 493-509. doi: 10.1109/32.799945

Kemerer, C. F., & Slaughter, S. (1997). Determinants of software maintenance profiles: An empirical investigation. *Journal of Software Maintenance, 9*(4), 235-251. doi: 10.1002/(SICI)1096-908X(199707/08)9:4<235::AID-SMR153>3.0.CO;2-3

Kendall, K. E., Kong, S., & Kendall, J. (2010). The impact of agile methodologies on the quality of information systems: Factors shaping strategic adoption of agile practices. *International Journal of Strategic Decision Sciences, 1*(1), 41-56. doi: 10.4018/jsds.2010103003

Kelly, D. (2006). A study of design characteristics in evolving software using stability as a criterion. *IEEE Transactions of Software Engineering, 32*(5), 315-329. doi: 10.1109/TSE.2006.42

Khan, K. (2004). *Managing corporate information systems evolution and maintenance.* Sydney, Australia: IGI Global Press.

Kline, R. B. (2005). *Principles and practice of structural equation modeling* (2nd ed.). New York: The Guilford Press.

Knipp, E., Driver, M., Norton, D., Pezzini, M., Murphy, J., Blechar, M. et al. (2010). *Key issues for application development.* Gartner research database.

Kozlov, D., Koskinen, J., Sakkinen, M., & Markkula, J. (2007). Assessing maintainability change over multiple software releases. *Journal of Software Maintenance and Evolution: Research and Practice*, *20*, 31-58. doi: 10.1002/smr.361

Kyte, A. (2011). Estimating the future cost of application maintenance. Gartner Research Database.

Kunstar, J., & Havlice, Z. (2008). Architecture for ease of software systems maintenance. *Applied Machine Intelligence & Informatics.*195-200. doi:10.1109/SAMI.2008.4469163

Kunz, M., Dumke, R. R., & Zenker, N. (2008). Software metrics for Agile software development. IEEE *Software Engineering Conference*, 673-678. doi: 10.1109/ASWEC.2008.4483261

Larson, R., & Farber, B. (2006). *Elementary statistics: Picturing the world* (3rd ed.). Upper Saddle River, NJ: Pearson Education.

Leedy, P. A., & Ormrod, J. E. (2001). *Practical research: Planning and design* (7th ed.). Columbus, OH: Merrill Prentice-Hall.

Law, J., & Rothermel, G. (2003). Whole program path-based dynamic impact analysis. *Proceedings of the 25th International Conference on Software Engineering,* 308-318. doi: 10.1109/ICSE.2003.1201210

Lee, K., & Lee, S. J. (2005). A quantitative software quality evaluation model for the artifacts of component based development. *IEEE Conference in Software Engineering, AI Networking & Parallel/Distributed Computing*, 20-25.doi: 10.1109/SNPD-SAWN.2005.7

Leffingwell, D. (2007). *Scaling software agility–Best practices for large enterprises*. MA: Addison-Wesley.

Lientz, B. P. & Swanson, E. B. (1980). *Software maintenance management.* MA: Addision-Wesley.

Lientz, B. P., & Swanson, E. B. (1978). Characteristics of application software maintenance. *Communications of ACM, 21(6),* 466-471. doi: 10.1145/359511.359522

Lientz, B. P., & Swanson, E. B. (1981). Problems in application software maintenance. *Communications of ACM*, 24(11), 763-769. doi:10.1145/358790.358796

Layman, L. (2004). Empirical investigation of the impact of extreme programming practices on software projects. *ACM Sigplan conference*,328-329. doi.10.1145/1028664.1028787

Lague, B., Proulx, D., Mayrand, J., Merlo, E. & Hudepoh, J. (1997). Assessing the benefits of incorporating function clone detection in a development process. *Proceedings of the International Conference on Software Maintenance*.

Lehman, M. M. (1996). Laws of software evolution revisited. *Proceedings of the 5th European workshop on software process technology*, 1149, 108-124.doi: 10.1007/BFb0017737

Lehman M. M. (1980). Programs, life cycles, and laws of software evolution. *Proceedings of IEEE,* (*68*)9, 1060-1076. doi: 10.1109/PROC.1980.11805

Lehman, M. M., & Belady, L. A. (1985). *Program evolution. Process of software change*. San Diego, CA: Academic Press.

Lehman, M. M., Ramil, J. F., & Wernick, P. D. (1997). Metrics and laws of software evolution-The nineties view. *Fourth International Software Metrics Symposium,* 20-32. doi: 10.1109/METRIC.1997.637156

Lehman, M. M., Perry, D. E., & Ramil, J. E. (1998). On evidence supporting the FEAST hypothesis and the laws of software evolution. *Fifth International Software Metrics Symposium,* 84-88. doi: 10.1109/METRIC.1998.731229

Lehman, M. M., & Ramil, J. F. (.2001). Evolution in software and related areas, *Proceedings of the 4th International Workshop on Principles of Software Evolution*. ACM. doi:10.1145/602461.602463

Lehman, M. M, Perry, E. E., & Ramil, J. F. C. (1998). Implications of evolution metrics on software maintenance. *ICSM98*, 208-217.doi.10.1109/ICSM.1998.738510

Lehman, M. M., Ramil, J. F., Wernick, P. D., Perry, D. E., & Turski, W. M. (1997). Metrics and laws of software evolution: The nineties view. *IEEE Journal,20-32,* doi: 10.1109/METRIC.1997.637156

Lindvall, M., Muthig, D., Dagnino, A.,Wallin ,C., Stupperich, M., Kiefer, … , J., Kahkonen,T. (2004). Agile software development in large organizations. *IEEE Computer Journal, 37*(12), 26-34. doi: 10.1109/MC.2004.231

Loof de, L. A. (1997). *Information systems outsourcing decision making*. *A managerial approach*. Hershey, PA: IDEA Group Publishing.

Lorenz, M., & Kidd, J. (1994). *Object-oriented software metrics.* NY: Prentice Hall.

Maxwell, S. E., & Delaney, H. D. (2000). *Designing experiments and analyzing data: A model comparison perspective*. Mahwah, NJ. : Lawrence Erlbaum.

Martin, J., & McClure, C. (1983). *Software maintenance: The problem and its solutions.* NY: Prentice-Hall.

McBreen, P. (2003). *Questioning extreme programming.* New York: Addison-Wesley Professional.

McClure, C. L. (1981). *Managing software development and maintenance.* New York: Van Nostrand Reinhold.

McCall, J. A., Richards, P. K., & Walters, G. F. (1977). Factors in software quality. *National Technical Information Service, 1*, 2-3. AN: ADA049014

McConell, S. (1996). *Rapid development*. Redmond, WA: Microsoft Press.

McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering, 2*(4), 308-320. doi: 10.1109/TSE.1976.233837

McCabe,T.J. & Watson, A. H.(1994). Software complexity. *Crosstalk*, *Journal of Defense Software Engineering, 7*(12), 5-9.

Mens, T., & Tourw´e, T. (2004). A survey of software refactoring. *IEEE Transactions on Software Engineering*, *30*(2),126 -139. doi: 10.1109/TSE.2004.1265817

Mens, T., & Demeyer, S. (2001). Future trends in software evolution metrics. *Proceedings of the Fourth International Workshop on Principles of Software Evolution (IWPSE)*,83-86. doi: 10.1145/602461.602476

Michura, J., & Capretz, M. A. M. (2005). Metrics suite for class complexity. In *Proceedings of the International Conference on Information Technology: Coding and Computing*, 2, 404-409. doi: 10.1109/ITCC.2005.193

Mistra, S. C. (2007). *Adopting software development practices: Success factors, changes required, and challenges*. (Unpublished doctoral dissertation). Retrieved from ProQuest research database.

Miranda, E. (2001). Improving subjective estimates using paired comparisons. *IEEE Software Journal*, *18*(1), 87-91. doi:10.1109/52.903173

Mohr, L. B. (1995). *Impact analysis for program evaluation*. Newbury Park, CA: Sage Publications.

Moser, R., Scotto, M., Sillitti, A., & Succi, G. (2007). Does XP deliver quality and maintainable code? *Proceedings of the 8th international conference on Agile processes in software engineering and extreme programming*. Springer-Verlag Publisher, 4536, 105-114. doi: 10.1007/978-3-540-73101-6_15

McClure, C. L. (1981). Managing software development and maintenance. New York: Van Nostrand Reinhold.

Monkevich, O. (1999). SDL-based specification and testing strategy for communication network protocols. In: *Proceedings of the 9th SDL Forum, Montreal, Canada*.

Mogyorodi, G., (2001). Requirements-based testing: An overview. In: *39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS39)*, 286-295, doi: 10.1109/TOOLS.2001.941681

Murphy, T. (2009). *Mix in the right test skills and achieve quality*. Gartner research Database.

Murphy-Hill, E., Parnin, C., & Black, A. P. (2009). How we refactor, and how we know it. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, 287-297. doi: 10.1109/ICSE.2009.5070529

Nair, K. (2010). Design property metrics to maintainability estimation: A virtual method using functional relationship mapping. *ACM Sigsoft Software Engineering, 35*(8). doi.10.1145/1874391.1874404

Nasution, M. F. F., & Weistroffer, H. R. (2009). *Documentation in Systems Development: A Significant Criterion for Project Success*,1-9. doi:10.1109/HICSS.2009.167

Nakatani, T., Tamai, T., Tomoeda, A., & Matsuda, H. (1997). Towards constructing a class evolution model. *Proceedings Asia-Pacific Software Engineering Conference and International Computer Science Conference*,131-138. doi: 10.1109/APSEC.1997.640170

Ohlsson, M. C., Andrews, A. A., & Wohlin, C. (2001). Modeling fault-proneness statistically over a sequence of releases: A case study. *Journal of Software Maintenance and Evolution: Research and Practice*, *13(3)*, 167-199. doi: 10.1002/smr.229

Orso, A., Apiwattanapong, T., & Harrold, M. J. (2003). Leveraging field data for impact analysis and regression testing. In *ESEC/FSE-11: Proceedings of the 9th European Software Engineering Conference*, 28(5), 128-137. doi: 10.1145/949952.940089

Oman, P. & Hagemeister, J. (1992). Metrics for assessing a software system's maintainability. *IEEE Journal, 9*, 337-334. doi.10.1109/ICSM.1992.242525

Osborne, W. (1990). *Software maintenance and computers.* Los Alamitos: IEEE Computer Society Press.

O'Rourke, N., Hatcher, L., & Stepanski, E. J. (2005). *A step-by-step approach to using SAS® for univariate & multivariate statistics* (2nd ed.) Cary, NC: SAS publishing & Wiley.

Paulish, D. J., & Carleton, A. D., (1994). Case studies of software-process-improvement measurement. *Computer Journal, 27*(9), 50-57. doi: 10.1109/2.312039

Paulk, M. (2001). Extreme programming from a CMM perspective. *IEEE Software*, *8*(6), 19-26. doi: 10.1109/52.965798

Parikh, G. (1982). The world of software maintenance. *Techniques of Program and System Maintenance*. Cambridge, MA: Winthrop.

Perepletchikov, M., Ryan, C., & Tari, Z. (2010). The impact of service cohesion on the analyzability of service-oriented software. *Services Computing, IEEE Transactions, 3*(2), 89-103. doi: 10.1109/TSC.2010.23

Pigoski, T. M. (1996). *Practical software maintenance: Best practices for managing your software investment*. New York, NY: Wiley Publication.

Ping, L. (2010). A quantitative approach to software maintainability prediction. *Information Technology and Applications (IFITA). International Forum*, 1, 105-108. doi. 10.1109/IFITA.2010.294

Pressman, R. S. (1994). *Software engineering: A practitioner's approach*. (6th ed.). New York: McGraw-Hill Publishing Co.

Porter, A. (1997). Fundamental laws and assumptions of software maintenance. *Empirical Software Engineering, 2*(2), 119-131. doi: 10.1023/A:1009793015685

Ravichandar, R., Arthur, J. D., Bohner, S. A., & Tegarden, D.P. (2008). Improving change tolerance through capabilities-based design: An empirical analysis. *Journal of Software Maintenance and Evolution: Research and Practice*, *20(2)*, 135-170, doi: 10.1002/smr.367

Ratzinger, J., Sigmund, T., Vorburger, P., & Gall, H. (2007). Mining software evolution to predict refactoring. *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*, 354-363. doi: 10.1109/ESEM.2007.9

Riaz, M., Mendes, E., & Tempero, E. (2009). *Empirical software engineering and measurement*. *ESEM*, 367- 377. doi. 10.1109/ESEM.2009.5314233

Ruiz, F., Aurora, V., Piattini, M., & Garcia, F.(2004). An ontology for the management of software maintenance projects. *International Journal of Software Engineering and Knowledge Engineering*, *14*(3), 323-349. doi: WSPC/117-ijseke

Rico, D. F. (2008). Effects of Agile methods on website quality for electronic commerce. *Proceedings of the 41st Hawaii International Conference on System Sciences*,463-463. Doi: 10.1109/HICSS.2008.137

Royal Academy of Engineering. (2004). *The challenges of complex IT projects*. London: British Computer Society.

Rosso, D. C. (2006). Continuous evolution through software architecture evaluation: A case study. *Journal of Software Maintenance and Evolution: Research and Practice, 18*, 351-383. doi.10.1002/smr.337

Saaty, T. L (1980). *The analytic hierarchy process, planning, priority setting, resource allocation.* New York: McGraw-Hill.

Sindhgatta, R., Narendra, C. N., & Sengupta, B. (2010). Software evolution in Agile development: A case study. *ACM Journal*, 105-114. doi:10.1145/1869542.1869560

Singleton, R. A., & Straits, B. C. (2005). *Approaches to social research* (4th ed.). New York: Oxford University Press.

Singh, Y., & Goel, B. (2007). A step towards software preventive maintenance. *SigSoft - Software Engineering*, *32*(4), 10. doi: 10.1145/1281421.1281432

Siket, I. (2010). Applying software product metrics in software maintenance. Published Dissertation, University of Szeged, Szeged, Hungary.

Schach, S., Jin, B., Wright, D., Heller, G., & Offutt, A. J. (2003). Determining the distribution of maintenance categories: Survey versus empirical study. *Kluwer's Empirical Software Engineering*, *8*(4), 351-365. doi: 10.1023/A:1025368318006

Schneiderwind, N. F. (1987). The state of software maintenance. *IEEE Transactions on Software Engineering*, *13*(3), 303-310. doi:10.1109/TSE.1987.233161

Schwaber, K., & Beedle, M. (2002). *Agile software development with Scrum*. Upper Saddle River, NJ: Prentice Hall.

Sneed H. (1995). Planning the reengineering of Legacy systems. *IEEE software, 12*(1), 24-34. doi: 10.1109/52.363168

Sommerville, I. (2000). *Software engineering* (6th ed.). New York: Addison-Wesley.

Standish Group International, Inc. (2004). *Third quarter research report*. West Yarmouth, MA: Standish Group.

Stauss, B. (1993). Service Problem Deployment: Transformation of problem information into problem prevention activities. *International Journal of Service Industry Management, 4*(2), 41-62. doi: 10.1108/09564239310037927

Stevens, J. P. (2009). *Applied multivariate statistics for the social sciences* (5th ed.). Mahwah, NJ: Routledge Academic.

Swanson, E. B. (1976). The dimensions of maintenance. *Proceedings of the 2nd international conference on software engineering*, 492-497. IEEE Computer Society Press. Los Alamitos, CA.

Tabachnick, B. G., & Fidell, L. S. (2001). *Using multivariate statistics* (4th ed.). Needham Heights, MA: Allyn and Bacon.

Tsantalis, N., Chatzigeorgiou, A., & Stephanides, G. (2005). Predicting the probability of change in object-oriented systems. *IEEE Transactions on Software Engineering,* 31(7). 601-614. doi: 10.1109/TSE.2005.83

Trochim, W. M. K., & Donnelly, J. P. (2007). *Research methods knowledge base.* (3rd ed.). Mason, OH: Thompson Learning Publications.

Testa, L. (2009). *Growing software: Proven strategies for managing software engineers*. San Francisco, CA: No Starch Press.

Tichy, W. F. (2004). Agile development: Evaluation and experience. *Software Engineering*, 692. doi: 10.1109/ICSE.2004.1317492

Turski, W. M. (1996). Reference model for smooth growth of software systems. *IEEE Transactions on Software Engineering*, *22*(8), 99-600. doi: 10.1109/32.536959

Turski, W. M. (2002). The reference model for smooth growth of software systems revisited. *Software Engineering, IEEE Transactions, 28*(8), 814-815. doi: 10.1109/TSE.2002.1027802

Vogt, P. W. (2007). *Quantitative research methods for professionals*. City: Pearson Publications.

Wiegers, K. E. (2005). *Software requirements.* (2nd ed.). Redmond: MS Press.

Wiener-Ehrlich, W. K., Hamrick, J. R., & Rupolo, V. F. (1984). Modeling software behavior in terms of a formal life cycle curve: Implications for software maintenance. *IEEE Transaction Software Engineering*, *10(4),* 376-383. doi: 10.1109/TSE.1984.5010250

Yeh, D., & Jeng, J. H. (2002). An empirical study of the influence of departmentalization and organizational position on software maintenance. *Journal of Software Maintenance and Evolution*. *Research and Practice, 14*(1), 65-82. doi: 10.1002/smr.246

Yu, L. & Chen, K. (2006). An empirical study of the maintenance effort. *Proceedings in the 8th International Conference on Software Engineering and Knowledge Engineering (SEKE).* San Francisco, CA. 242-245, ISBN 1-891706-18-7

Zhe, M., & Kerong, B. (2010). Research on maintainability evaluation of service-oriented software. 4, *IEEE Software Journal,* 510-512. doi:10.1109/ICCSIT.2010.5563562

Appendix A: Agile practice areas and underlying practices

Table A1

Agile practice areas/Agile characteristics	Key objectives
Customer team member/participation	Represent customer group for the functional, business, and technical requirements. Better and timely interaction with the development team. Better synch up & alignment with developers and IT operations organization.
User stories	User requirement that also serves as planning tool to schedule the implementation of a requirement Every user story has priority and cost. User story can evolve and change!
Short cycles/Iterations	Delivers the working stories in iterative manner typically over 1-2 weeks. Customer agrees to freeze the stories being developed in iteration and review the results at the end of the iteration cycle. Team work according to release plan that delivers prioritized user stories selected by customer that consists of several smaller iterations with agreed order (priority).
Acceptance tests	Mechanism to verify the user story is behaving as specified by customer. User story is always associated with one or more acceptance test(s) that need to be passed and continue to be successful throughout the iteration(s) and release(s).

Table A1 *(Continued)*

Agile practice areas/Agile characteristics	Key objectives
Pair programming	Technique to promote the quality (reduced defect rate) as well the knowledge across the development team Promote cross skill development opportunities among the team members.
Test driven development	Tests are written before writing code encouraging the quality checkpoints on modular level. Test must pass with every change in the code Technique that helps refactoring and decoupling within the code in longer run.
Collective ownership	Frequent code check in/check-out with collective ownership of the code among the developers No single authority over code promoting team environment and discipline over the code management through the development cycle.
Continuous integration/Automated Deployments	System build several times a day with newly integrated code. At the end of the integration, all the tests must pass and final state of the system is working, promoting greater quality checkpoints.

Table A1 *(Continued)*

Agile practice areas/Agile characteristics	Key objectives
The planning game	Discussion that yields clear responsibility and understanding on the decisions about functional features and required efforts from the development team allowing customer and project team to grasp the overall project timelines This planning effort enlists and agrees with number of iterations and releases within given resources for the project
Simple Design	Focus on the design within iteration period with simplicity in mind Code duplication is avoided. Only justified and needed infrastructural components are considered per iteration/user story basis, rather than looking for distant requirements
Refactoring	Continuous exercise to transform the code to simpler, cleaner, and scalable state Refactoring need not wait for release or iteration to end but can be done every hour within the code development phase to ensure the clean design

Appendix B: AHP Scoring Protocol

The participant that is also an Agile software development model expert use values from 1 to 5 to score the importance of the software source code attributes for software maintainability sub-characteristics viz. software analyzability, changeability, stability, and testability.

Table B1

AHP Scoring Protocol with explanation

Value	For	Explanation
1	Equal importance	The two system properties contribute equally to software maintainability sub-characteristics (Software analyzability, changeability, stability, and testability).
2	Moderate importance	One system attribute moderately contribute than other system attribute.
3	Strong importance	One system attribute strongly contribute than other attribute.
4	Very strong importance	A system attribute is favored very strongly over another.
5	Extreme importance	A system attribute or property is absolutely important than other system attribute.

Q	Questions to facilitate pair-wise comparison based on subjective assessment	**Score (1-5)**
	Scoring for software code attribute importance for Analyzability	
Q1	In your experience, how important is code complexity than code coupling when evaluating software analyzability?	2
Q2	In your experience, how important is code complexity than code duplication when evaluating software analyzability?	2
Q3	In your experience, how important is code complexity than unit test efforts when evaluating software analyzability?	5
Q4	In your experience, how important is code coupling than code duplication when evaluating software analyzability?	1
Q5	In your experience, how important is code coupling than unit test efforts when evaluating software analyzability?	5
Q6	In your experience, how important is code duplication than unit test efforts when evaluating software analyzability?	5
	Scoring for software code attribute importance for Changeability	
Q7	In your experience, how important is code complexity than code coupling when evaluating software changeability?	3

Q8	In your experience, how important is code complexity than code duplication when evaluating software changeability?	3
Q9	In your experience, how important is code complexity than unit test efforts when evaluating software changeability?	1
	(Continued table)	
Q10	In your experience, how important is code coupling than code duplication when evaluating software changeability?	1
Q11	In your experience, how important is code coupling than unit test efforts when evaluating software changeability?	1
Q12	In your experience, how important is code duplication than unit test efforts when evaluating software changeability?	1
	Scoring for software code attribute importance for Stability	
Q13	In your experience, how important is code complexity than code coupling when evaluating software stability?	3
Q14	In your experience, how important is code complexity than code duplication when evaluating software stability?	3
Q15	In your experience, how important is code complexity than unit test efforts when evaluating software stability?	1
Q16	In your experience, how important is code coupling than code duplication when evaluating software stability?	1

Q17	In your experience, how important is code coupling than unit test efforts when evaluating software stability?	4
Q18	In your experience, how important is code duplication than unit test efforts when evaluating software stability?	4
	Scoring for software code attribute importance for Testability	
Q19	In your experience, how important is code complexity than code coupling when evaluating software testability?	2
Q20	In your experience, how important is code complexity than code duplication when evaluating software testability?	1
Q21	In your experience, how important is code complexity than unit test efforts when evaluating software testability?	5
Q22	In your experience, how important is code coupling than code duplication when evaluating software testability?	2
Q23	In your experience, how important is code coupling than unit test efforts when evaluating software testability?	4
Q24	In your experience, how important is code duplication than unit test efforts when evaluating software testability?	4

Additional Information to Aid scoring

(Maintainability Sub-Characteristics definitions and Software attribute measures)

Analyzability: It is a software capability to allow identification for parts that should be modified. ISO 9126 defines it as attributes of software that bear on the effort needed for diagnosis of deficiencies or causes of failures, or for identification of parts to be modified.

Changeability: It is a software capability to enable a specified modification to software system to be implemented. ISO 9126 defines it as attributes of software that bear on the effort needed for modification, fault removal or for environmental change.

Stability: It is a capability of the software product to avoid unexpected effects from modifications of the software. ISO 9126 defines it as attributes of software that bear on the risk of unexpected effect of modifications.

Testability: It is a capability of the software product to enable modified software to be validated. ISO 9126 defines it as attributes of software that bear on the effort needed for validating or testing the modified software.

Table B2

Software Properties or Source Code Attribute with Applicable Measures

Attribute	**Operationalization**
Unit Size	Number of lines of code/statements in a method
Complexity	CC (McCabe Metric) = number of linearly independent paths through a source code (method/class)
Coupling	Count of the number of classes to which a class is coupled (CBO)
Duplication	The % of all code that occurs more than once in equal code blocks of at least 6 lines
Unit Testing	Percent of Test/code coverage and count of Assert within class files

Appendix C: AHP Pair-Wise Comparison & Weight Tabulation

Table C1

*Pair-wise Comparison with Respect to **Analyzability** Sub-Variable Yielding the Weight for Complexity, Coupling, Duplication, and Unit Test Effort.*

Analyzability	Complexity	Coupling	Duplication	Unit Test Effort
Complexity	1			
Coupling		1		
Duplication			1	
Unit Test Effort				1
Normalized Eigenvalues	$W_{AHP\text{-}Complexity\ For\ Analyzability}$	$W_{AHP\text{-}Coupling\ For\ Analyzability}$	$W_{AHP\text{-}Duplication\ For\ Analyzability}$	$W_{AHP\text{-}UnitTestEfforts\ For\ Analyzability}$

Table C2

*Pair-Wise Comparison with Respect to **Changeability** Sub-Variable Yielding the Weight for Complexity, Coupling, Duplication, and Unit Test Effort*

Changeability	Complexity	Coupling	Duplication	Unit Test Effort
Complexity	1			
Coupling		1		
Duplication			1	
Unit Test Effort				1
Normalized Eigenvalues	$W_{AHP\text{-}Complexity\ For\ Changeability}$	$W_{AHP\text{-}Coupling\ For\ Changeability}$	$W_{AHP\text{-}Duplication\ For\ Changeability}$	$W_{AHP\text{-}UnitTestEfforts\ For\ Changeability}$

Table C3

*Pair-Wise Comparison with Respect to **Stability** Sub-Variable Yielding the Weight for Complexity, Coupling, Duplication, and Unit Test Effort*

Stability	Complexity	Coupling	Duplication	Unit Test Effort
Complexity	1			
Coupling		1		
Duplication			1	
Unit Test Effort				1
Normalized Eigenvalues	W*AHP-Complexity For Stability*	W*AHP-Coupling For Stability*	W*AHP-Duplication For Stability*	W*AHP-UnitTestEfforts For Stability*

Table C4

*Pair-Wise Comparison with Respect to **Testability** Sub-Variable Yielding the Weight for Complexity, Coupling, Duplication, and Unit Test Effort*

Testability	Complexity	Coupling	Duplication	Unit Test Effort
Complexity	1			
Coupling		1		
Duplication			1	
Unit Test Effort				1
Normalized Eigenvalues	W*AHP-Complexity For Testability*	W*AHP-Coupling For Testability*	W*AHP-Duplication For Testability*	W*AHP-UnitTestEfforts For Testability*

Table C5

Final Source Code Property Weight Based on Above Pair-Comparison Tables

Source Code Property Weight	Analyzability	Changeability	Stability	Testability
Complexity	W*AHP-Complexity For Analyzability*	W*AHP-Complexity For Changeability*	W*AHP-Complexity For Stability*	W*AHP-Complexity For Testability*
Coupling	W*AHP-Coupling For Analyzability*	W*AHP-Coupling For Changeability*	W*AHP-Coupling For Stability*	W*AHP-Coupling For Testability*
Duplication	W*AHP-Duplication For Analyzability*	W*AHP-Duplication For Changeability*	W*AHP-Duplication For Stability*	W*AHP-Duplication For Testability*
Unit Test Effort	W*AHP-UnitTestEfforts For Analyzability*	W*AHP-UnitTestEfforts For Changeability*	W*AHP-UnitTestEfforts For Stability*	W*AHP-UnitTestEfforts For Testability*

Table C6

Tools Used for Metrics Measurement

Purpose	Tool Name	URL
Source Code management and download	TortoiseSVN Ver. 1.6.2.16344	http://tortoisesvn.net (Open source)
Unit Size measurement	Source Monitor Ver. 2.6.8.123	http://campwoodsw.com (Student License)
Source code metrics collection (CC, CBO, Assert).	Understand Ver. 2.6	http://www.scitools.com (Student License)
Test Coverage and Continuous Integration log.	CruiseControl 1.3.0.2918	http://confluence.public.thoughtworks.org/display/CCNET/ (Open Source)
Code cloning	Solid SDD 1.3	www.solidsourceit.com

Appendix D: Raw Data Collection Template shown for 20 iterations with Actual Measure

Table D1

Software Attribute, Their Measure, and the Actual Data Collection Template Sheet

Measure	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10
Iteration	(0 < McCabe CC <10)	1 /(11 < McCabe CC < 20)	1 /(21 < McCabe CC < 50)	1/ Unit Size	0 < CBO < 6	7 < CB0 < 14	1/ (15 < CBO < 50)	Assert/Class	Test Coverage	1/Cloning
I1										
I2										
I3										
I4										
I5										
I6										
I7										
I8										
I9										
I10										
I11										
I12										
I13										
I14										
I15										
I16										
I17										
I18										
I19										
I20										

Note that, M2, M3, M4,and M7 are negatively correlated to maintainability characteristics as discussed in literature. In other words, increase in complexity, unit size, coupling, and duplication reduces the maintainability and its sub-characteristics. Note that programmer attempts to lower code complexity, lower the coupling between objects reduce the method size, and the code duplication in the program to improve the maintainability traits during the software development iterations.

Appendix E: NIH Certificate of Completion

Certificate of Completion

The National Institutes of Health (NIH) Office of Extramural Research certifies that **Ajay Gawali** successfully completed the NIH Web-based training course “Protecting Human Research Participants”.

Date of completion: 09/14/2011

Certification Number: 757322

Appendix F: Final Calculation of AHP Weights for analyzability characteristics as an example

Weights for source code attributes for analyzability characteristic

	Complexity	Coupling	Cloning	Unit Test Efforts	Weights
Complexity	1.00	2.00	2.00	5.00	0.4299
Coupling	0.50	1.00	1.00	5.00	0.2540
Cloning	0.50	1.00	1.00	5.00	0.2540
Unit Test Effort	0.20	0.20	0.20	1.00	0.0622

Appendix G: Raw data for first 5 iterations shown as example

Measure	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10
Iteration	(0 < McCabe CC <10)	1 /(11 < McCabe CC < 20)	1 /(21 < McCabe CC < 50)	1/ Unit Size	0 < CBO < 6	7 < CB0 < 14	1/ (15 < CBO < 50)	Assert/Class	Test Coverage	1/Cloning
I1	0.71	7.12	9.58	0.31	0.61	0.25	7.10	5.12	0.2774	10.267
I2	0.71	7.07	9.74	0.31	0.61	0.25	7.50	5.45	0.2898	10.799
I3	0.71	7.08	9.79	0.31	0.61	0.25	7.55	5.48	0.2908	10.811
I4	0.72	7.12	9.90	0.31	0.62	0.26	8.06	5.56	0.2937	11.325
I5	0.72	7.04	10.16	0.31	0.62	0.26	8.58	5.79	0.295	11.351

Appendix H: Nonstandardized Data for All Variables

	X1	X2	X3	Y1	Y2	Y3	Y4	Y
Iteration	Test Driven Development/TDD	Refactoring (REFR)	Continuous Integration (CI)	Analyzability	Changeability	Stability	Testability	Maintainability
1	0.1666	6.784260516	20	12.49776035	11.42597383	8.494733345	7.781915577	472.9946676
2	0.1741	6.877579092	45	12.80024847	11.70830642	8.840181761	8.131200738	487.4023402
3	0.1766	6.920415225	57	12.85053862	11.75647821	8.877304131	8.166367366	489.3984575
4	0.1773	6.954102921	19	13.1777447	12.01064431	9.132417303	8.384927619	501.6762613
5	0.1819	7.087172218	21	13.40658272	12.22930398	9.365866594	8.614213836	512.038094

Appendix I: Standardized Z Scores for 5 iterations for Independent (X) Variables & Final Y Values, and weighted values as an example

Software Maintainability Computation Table

Iteration	TDD (X1)	REFR (X2)	CI (X3)	Analyzability (Y1)	Changeability (Y2)	Stability (Y3)	Testability (Y4)	Maintainability (Y)
1	-1.86	-1.44	-0.67	12.5	11.43	8.49	7.78	472.99
2	-1.72	-1.12	0.61	12.8	11.71	8.84	8.13	487.4
3	-1.67	-0.98	1.22	12.85	11.76	8.88	8.17	489.4
4	-1.65	-0.87	-0.72	13.18	12.01	9.13	8.38	501.68
5	-1.57	-0.42	-0.62	13.41	12.23	9.37	8.61	512.04

Example of Weighted software analyzability

Analyzability (Y1)	M = Sum of T / 61	Occurrence	T = Y1 X Occurrence	Weighted Y1 = Y1 X M
12.50	13.71	1	12.49776035	171.3879341
12.80	13.71	1	12.80024847	175.5361024

Example of Weighted software changeability

Changeability (Y2)	M = Sum of T / 61	Occurrence	T = Y2 X Occurrence	Weighted Y2 = Y2 X M
11.43	12.67	1	11.4259738	144.7285024
11.71	12.67	1	11.7083064	148.3047028

Example of Weighted software stability

Software Stability (Y3)	M = Sum of T / 61	Occurrence	T = Y3 X Occurrence	Weighted Y3 = Y3 X M
8.49	9.992803279	1	8.49473334	84.88619922
8.84	9.992803279	1	8.84018176	88.33819728

Example of Weighted software testability (Y4)

Software Testability (Y4)	M = Sum of T / 61	Occurrence	T = Y4 X Occurrence	Weighted Y4 = Y4 X M
7.78	9.25	1	7.781915577	71.99203187
8.13	9.25	1	8.131200738	75.22333761

Example of Weighted Y or final weighted maintainability

Weighted Analyzability W_{Y1}	Weighted Changeability W_{Y2}	Weighted Stability W_{Y3}	Weighted Testability W_{Y4}	Weighted Maintainability (Y = (W_{Y1}+ W_{Y2}+ W_{Y3}+ W_{Y4})
171.3879341	144.7285024	84.8861992	71.9920319	472.9946676
175.5361024	148.3047028	88.3381973	75.2233376	487.4023402

Appendix J: Descriptive Statistics Summary

Sample Descriptive Statistics for selected variables

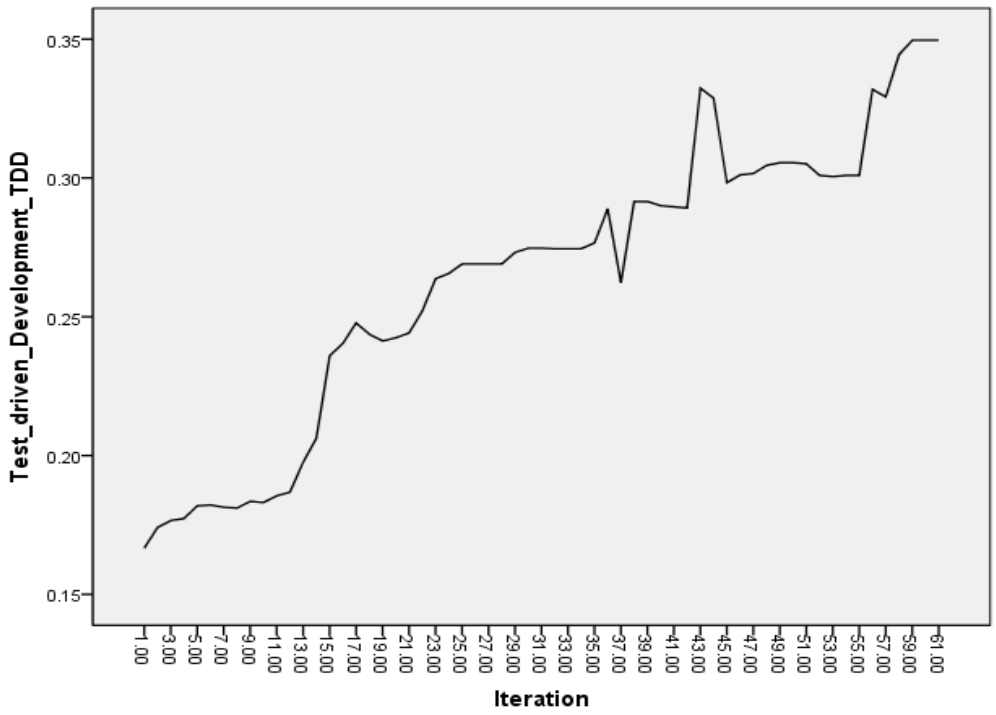
	N	Minimum	Maximum	Mean	Std. Deviation
Test_driven_Development_TDD	61	.17	.35	.2637	.05222
Standard_TDD	61	-1.86	1.65	.0000	1.00000
Analyzability	61	12.50	14.56	13.7144	.39124
Maintainability	61	472.99	563.28	537.8634	20.93317
Valid N (listwise)	61				

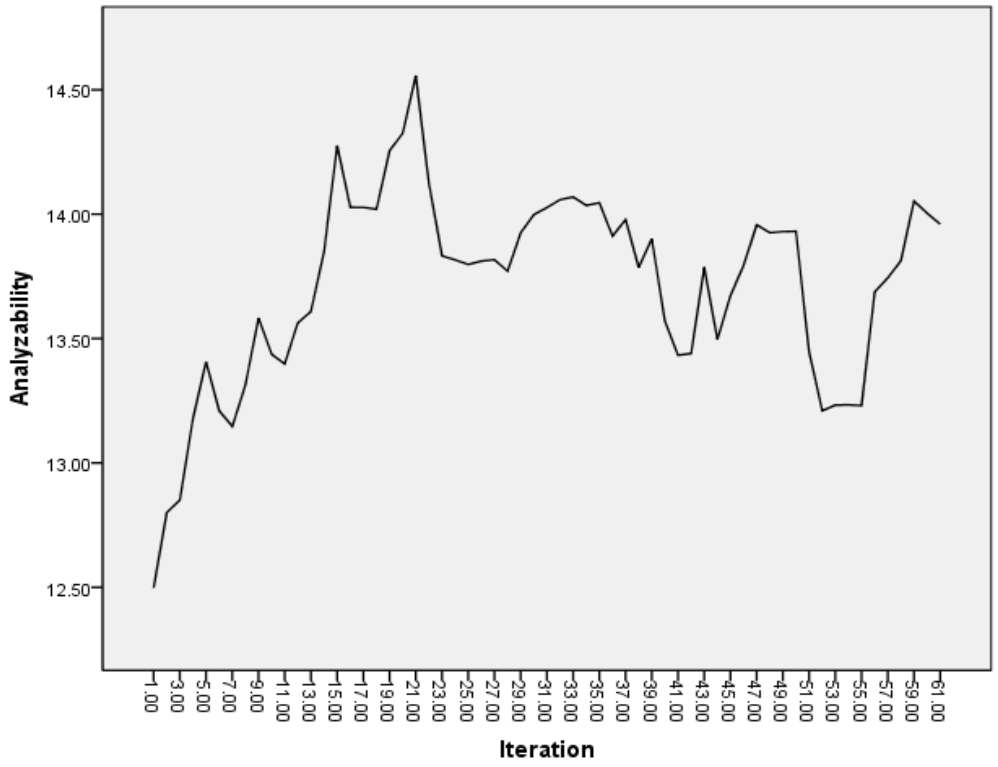
Correlations

		Standard_TDD	Standard_REFR	Standard_CI	Analyzability
Standard_TDD	Pearson Correlation	1	-.116	.262*	.411**
	Sig. (2-tailed)		.373	.042	.001
	N	61	61	61	61
Standard_REFR	Pearson Correlation	-.116	1	-.347**	.703**
	Sig. (2-tailed)	.373		.006	.000
	N	61	61	61	61
Standard_CI	Pearson Correlation	.262*	-.347**	1	-.098
	Sig. (2-tailed)	.042	.006		.454
	N	61	61	61	61
Analyzability	Pearson Correlation	.411**	.703**	-.098	1
	Sig. (2-tailed)	.001	.000	.454	
	N	61	61	61	61
Changeability	Pearson Correlation	.778**	.420**	.057	.885**
	Sig. (2-tailed)	.000	.001	.662	.000
	N	61	61	61	61
Stability	Pearson Correlation	.811**	.276*	.104	.798**
	Sig. (2-tailed)	.000	.031	.423	.000
	N	61	61	61	61
Testability	Pearson Correlation	.877**	.158	.144	.701**
	Sig. (2-tailed)	.000	.223	.267	.000
	N	61	61	61	61

Maintainability	Pearson Correlation	.759**	.411**	.050	.888**
	Sig. (2-tailed)	.000	.001	.705	.000
	N	61	61	61	61

Additional sample sequence charts for independent and dependent variable




Analyzability
14.50
14.00
13.50
13.00
12.50
1.00
3.00
5.00
7.00
9.00
11.00
13.00
15.00
17.00
19.00
21.00
23.00
25.00
27.00
29.00
31.00
33.00
35.00
37.00
39.00
41.00
43.00
45.00
47.00
49.00
51.00
53.00
55.00
57.00
59.00
61.00
Iteration

Appendix K: Regression Model example for software analyzability and Charts

Model Summary[b]

Model	R	R Square	Adjusted R Square	Std. Error of the Estimate
1	.861[a]	.741	.728	.20417

a. Predictors: (Constant), Standard_CI, Standard_TDD, Standard_REFR

b. Dependent Variable: Analyzability

ANOVA[b]

Model		Sum of Squares	df	Mean Square	F	Sig.
1	Regression	6.808	3	2.269	54.440	.000[a]
	Residual	2.376	57	.042		
	Total	9.184	60			

a. Predictors: (Constant), Standard_CI, Standard_TDD, Standard_REFR

b. Dependent Variable: Analyzability

Coefficients[a]

Model		Unstandardized Coefficients	
		B	Std. Error
1	(Constant)	13.714	.026
	Standard_TDD	.191	.027
	Standard_REFR	.303	.028
	Standard_CI	.017	.029

Coefficients[a]

Model		Standardized Coefficients	t	Sig.	Collinearity Statistics	
		Beta			Tolerance	VIF
1	(Constant)		524.628	.000		
	Standard_TDD	.489	7.006	.000	.931	1.074
	Standard_REFR	.775	10.781	.000	.879	1.138
	Standard_CI	.043	.585	.561	.830	1.205

Coefficient Correlations[a]

Model			Standard_CI	Standard_TDD	Standard_REFR
1	Correlations	Standard_CI	1.000	-.238	.330
		Standard_TDD	-.238	1.000	.028
		Standard_REFR	.330	.028	1.000
	Covariances	Standard_CI	.001	.000	.000
		Standard_TDD	.000	.001	2.148E-5
		Standard_REFR	.000	2.148E-5	.001

a. Dependent Variable: Analyzability

Collinearity Diagnostics[a]

Model	Dimension	Eigenvalue	Condition Index
1	1	1.495	1.000
	2	1.000	1.223
	3	.889	1.297
	4	.616	1.558

Collinearity Diagnostics[a]

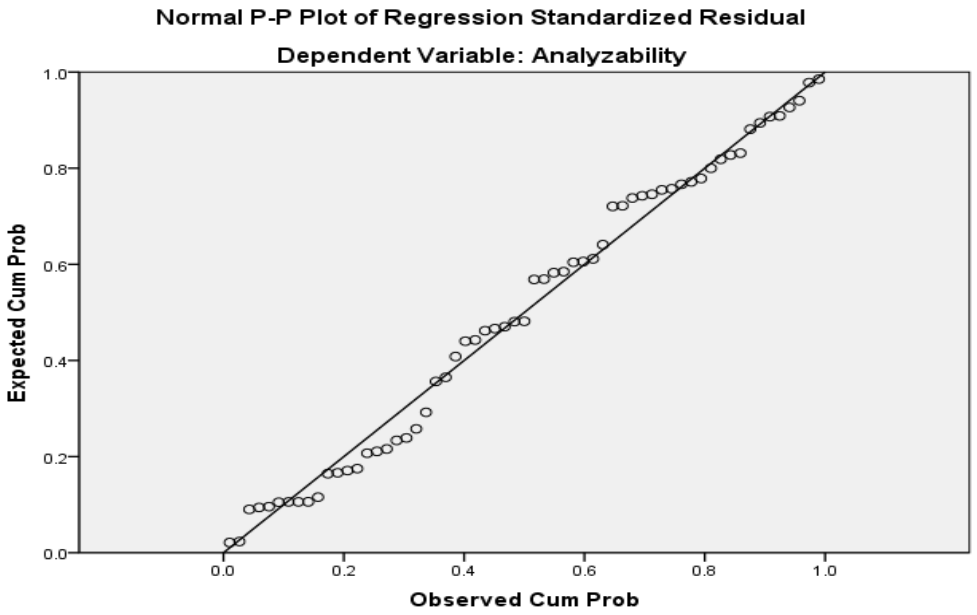
Model	Dimension	Variance Proportions			
		(Constant)	Standard_TDD	Standard_REFR	Standard_CI
1	1	.00	.15	.19	.24
	2	1.00	.00	.00	.00
	3	.00	.68	.34	.01
	4	.00	.17	.47	.75

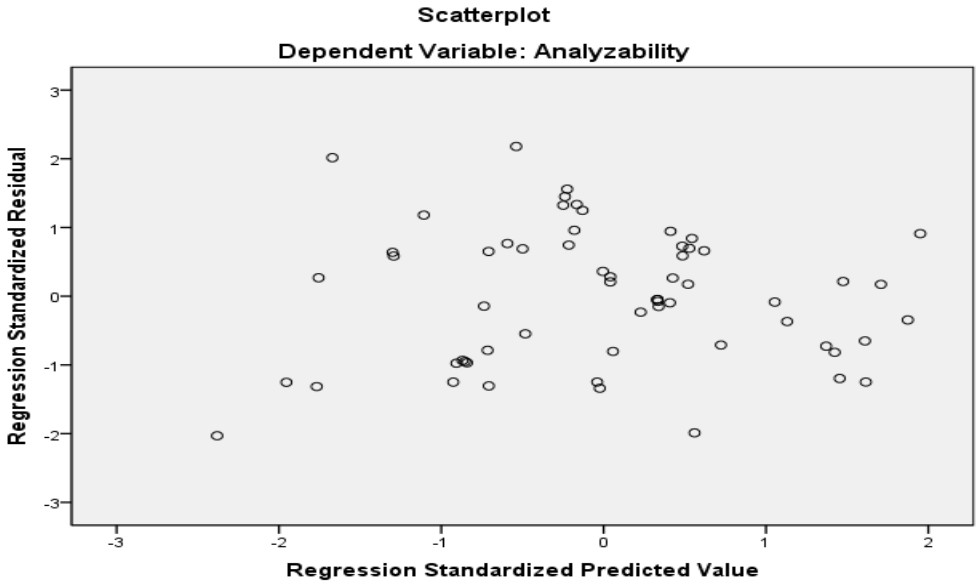
Residuals Statistics[a]

	Minimum	Maximum	Mean	Std. Deviation	N
Predicted Value	12.9124	14.3713	13.7144	.33685	61
Residual	-.41460	.44492	.00000	.19900	61
Std. Predicted Value	-2.381	1.950	.000	1.000	61
Std. Residual	-2.031	2.179	.000	.975	61

a. Dependent Variable: Analyzability

Residual Chart and scatter plot for software analyzability (SA) or Y1 variable





Appendix L: Adjusted Regression Model without CI variable

Analysis of Variance for software maintainability (SM)

Adjusted model without CI(X3) variable included in the regression model

Analysis of Variance

Source	DF	Sum of squares	Mean square	F value	Pr > F
Model	2	21768	10884	139.51	<.0001
Error	58	4525	78		
		R-square	82.8%		
		Adj. R-sq	82.2%		

Predictor	Coef	SE Coef	T	P	VIF
Constant	537.865	1.131	475.61	0.000	
TDD (X1)	17.089	1.147	14.90	0.000	1.014
REFR (X2)	10.585	1.148	9.22	0.000	1.014

The regression equation is: SM (Y) = 538 + 17.1 TDD (X1) + 10.6 REFR (X2)

Appendix M: Curriculum Vitae

Ajay R Gawali, Arizona USA
ajay.gawali@waldenu.edu

Education:

Doctor of Philosophy – Applied Management & Decision Sciences 2012
Walden University, Minneapolis, Minnesota

Master of Science – Physics 1991
Pune University, Maharashtra, India

Bachelor of Science – Electronics 1989
Pune University, Maharashtra, India

Certifications: CISA, MCITP, MCDBA

Professional Affiliations:

Professional Member, ISACA (Information Systems Audit & Control Association)
Member, IEEE (Institute of Electrical & Electronics Engineers)
Member, ACM (Association for Computing Machinery)
Member, ASQ (American Society for Quality)
Lead Assessor for ISO 9000/9001 Quality Management System

Honors and Awards:

Division Recognition Awards: Intel's Information Technology, Facilities & Materials
Excellence Award: Intel's Corporate Services Finance

Research support:

Intel Corporation, Arizona.
Current Research: "Impact of Agile Software Development Model on Software Maintainability and its sub-characteristics"

Relevant Professional Experience:

Sr. System Analyst 2000 - Present
IT Supply Network Capability, Production Services, Intel Corporation, Arizona, US

Lead and represent as a key strategic operation resource on several application developments, integration, & upgrade projects within supply network capability organization in IT. Integrate and manage SAP/Business Objects Reporting application environment for BI (Business Intelligence) platform, MS SQL (2005/2008) Clusters, IIS 6/7 web farms, DFS environment, Agile deployment toolsets and infrastructure. Lead the technical integration, architecture review, and deployment of IP compliance solutions, streamline and maintain the technology, platform, & application life cycle, and serve as 4th level escalation contact for tactical operation team. Work as strategic operation analyst

on several cross functional teams, projects, and IT initiatives critical for Intel's business. Delivered several internal training sessions to improve efficacy of strategic and tactical organizations within production service team that is driven by ITIL service framework.

Sr. System Administrator 2000 - 2003
Intel Online Services, Intel Japan, Tokyo

Integrated, deployed, and sustained several MS AD servers, Wins, DHCP, DFS servers, MS SQL DB Clusters with IIS 5 web farms, administered & supported several data center operation management tools for managed and hosting services for business critical application infrastructure, consulted, designed, and integrated technical solutions for highly available, secured, and reliable application platform for 24 X7 data center operational needs, and served as 3rd level escalation contact for global tactical technical team. Led and delivered several technical training sessions for the operation as well technical project delivery team to improve its strategic capability.

System Administrator 1999 – 2000
Interlogic K.K., Tokyo, Japan

Deployed and upgraded global application toolsets for Goldman Sachs, Japan for business critical applications. Integrated and administered MS NT, SQL Server, and Exchange server infrastructure and applications. Designed and developed IT security, administration policies and procedures. Led several internal audit and training sessions.

System Administrator 1997 - 1998
Electronics Data System, Kobe, Japan

Administered MS Windows NT, Lotus Notes, Novell Netware 30 + servers supporting 3000 + user base for P&G Asia Pacific Head office. Managed and supported Arc Serve backup software system. Technical lead for Novell Servers located across P&G's Japan-wide factory locations. Upgraded and deployed enterprise workstation platform for 1500+ user base.

System Engineer 1992 - 1997
Author Systems, India

Integrated, administered and supported Novell Netware 2.2, 3.12, 4.X servers, supported wide range of IT customer base including training and manufacturing. Integrated and supported several applications and Novell Netware clients. Managed vendor, contract & license support. Administered and supported training organizational user base, designed several training courses for IT training division. Served as Novell expert on several consulting projects for industrial and educational institutions.