SCORING A SOFTWARE DEVELOPMENT ORGANIZATION WITH A SINGLE NUMBER

BY

RYAN M. SWANSTROM

A dissertation submitted in partial fulfilment of the requirements for the
Doctor of Philosophy
Major in Computational Science and Statistics
South Dakota State University
2015

SCORING A SOFTWARE DEVELOPMENT ORGANIZATION
WITH A SINGLE NUMBER

This dissertation is approved as a creditable and independent investigation by a candidate for the Doctor of Philosophy degree and is acceptable for meeting the dissertation requirements for this degree. Acceptance of this does not imply that the conclusions reached by the candidates are necessarily the conclusions of the major department.

Dr. Gary Hatfield
Dissertation Advisor Date

Dr. Kurt Cogswell
Head, Math and Statistics Date

Dean, Graduate School Date

ACKNOWLEDGEMENTS

I would like to acknowledge the generous support I received from my family. Thank you to Emily for providing encouragement and the final push to get me to eventually finish. Thank you to Ainsley, Porter, Trey, and Ryker for always providing a smile.

I would also like to acknowledge numerous other people for providing guidance and support during the process. I will only use first names, but you know who you are. Thank you to: Jess, Chris, Clay, Bob, Todd, Leslie, Mike, and Ralph.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

SCORING A SOFTWARE DEVELOPMENT ORGANIZATION WITH A SINGLE NUMBER

RYAN M. SWANSTROM

2015

Nearly every large organization on Earth is involved in software development at some level. Some organizations specialize in software development while other organizations only participate in software development out of necessity. In both cases, the performance of the software development matters. Organizations collect vast amounts of data relating to software development. What do the organizations do with that data? That is the problem. Many organizations fail to do anything meaningful with the data.

Another problem is knowing what data to collect. There are many options, but certain data is more important than others. What data should a software development organization collect?

This paper plans to answer that question and present a framework to gather the right information and provide a score for an organization that produces software. The score is not to be comparative between organizations, but to be comparative for a specific organization over time.

The primary goal of this work is to provide a general framework for what a software development organization should measure and how to report on those measurements. The focus is providing a single number to represnet the entire organization and not just the development efforts. That single number is considered the CRI score. The secondary goal of this work is to provide a software implementation of that framework.

1 INTRODUCTION

Software is becoming a vital part of companies. In 2011 Marc Andreessen, co-founder of the venture capital firm Andreessen-Horowitz, famously claimed, "Software is Eating The World" [1]. His argument was for the ever increasing importance of software in all organizations big and small regardless of the industry. With this important declaration, the production of new software is going to be critical. Just as important will be the effective measurement of how this software is produced.

This work provides a technique for a software development organization to create a single number score which indicates the overall performance of the organization. The score is based upon data collected for 5 result indicators of a software development organization: quality, availability, satisfaction, schedule, and requirements. It is not meant to be comparative between organizations, but to form a historical baseline for a specific organization.

1.1 OVERVIEW

Software development organizations are no different than any other business or organization. There are: tasks to be completed, goals to achieve (or miss), and measurements to be analyzed. One difficulty with software development is the varied number and amount of measurements to be used. It can be difficult to determine the correct activities to measure and the appropriate mechanism to report the measure. This has led organizations to either collect too little information or to collect too much information. Another problem is the inconsistency of the reported measures. It is difficult to compare historical performance if the same measurements are not consistent throughout the recent history of an organization.

Software development organizations need a framework to define what

measurements should be tracked and how those measurements should be reported. The Cumulative Result Indicator (CRI) framework provides a solid foundation for a consistent evaluation. CRI analyzes the historical performance of a Software Development Organization (SDO) to create a baseline in order to provide a broad view of the overall organization. It is common for software development organizations to measure and focus soley on the source code being produced. However, a software development organization does more than just produce source code. There is documentation to be written, testcases to be created, systems to be deployed, and decisions to be made. The framework provides an evaluation of the overall software development organization, not just the source code.

The framework will produce a single number score for each of the five result indicators as well as a single overall score. It will be able to provide a quick evaluation of the organization. The scores will enable performance to be consistently measured and compared.

Other attempts at evaluating a software development organization have been presented, but none produce a single number score for the entire effort of the software development organization. The following are attempts to evaluate all or parts of software development.

1.2 TERMINOLOGY

Like any other business domain, the software engineering field has number of specific terms. Many of these terms will be used throughout the remainder of the document, so definitions are provided.

Application - a software system or a collection of other applications

Release - A collection of projects being put in production on a specified date

Project - A body of work involving zero or more applications in preparation

for a release

SIT (Systems Integration Testing) - The initial step of testing after the development phase of the SDLC. This is typically performed by members of the SDO. It is validation that all the software components function together as expected.

UAT (User Acceptance Testing) - The final step of testing when a select few members of the user group are invited to validate the software system. Once validation has occured for UAT, the software system is ready to proceed to production

PROD (Production) - The software has been released to the final audience.

defect "A software defect is a bug or error that causes software to either stop operating or to produce invalid or unacceptable results" as quoted from Capers Jones [43]. It is important to mention that even though defects are typically found in the computer code, a defect should not be isolated to just code. A poorly written requirement or missed test cases can both be considered a defect. Other common names for a defect are: bug, error, fault, or ticket.

1.3 CMMI

The Capability Maturity Model Integration (CMMI) is one of the most widely acknowledged models for process improvement in software development. CMMI offers a generic guideline and appraisal program for process improvement. It was created and is administered by the Software Engineering Institute at Carnegie Mellon University [15]. While the CMMI is not specific to software development, it is often applied in software development settings. CMMI certification is required for many United States Government and Department of Defense contracts.

CMMI-Dev is a modification of the CMMI specific to the development activities applied to products and services. The practices covered in CMMI-Dev include project management, systems engineering, hardware engineering, process management, software engineering, and other maintenance processes. Five maturity levels are specified, and they include the existence of a number of process areas. The 5 maturity levels and the process areas are specified as follows.

CMMI MATURITY LEVEL 1 - INITIAL A maturity level 1 organization consists of an adhoc and chaotic processes. While working products are still produced, the results are often over budget and behind schedule. A level 1 organization will also have difficulties repeating a process with the same degree of success. These organizations typically rely on the heroic efforts of certain individuals.

CMMI MATURITY LEVEL 2 - MANAGED A maturity level 2 organization has a policy for planning and executing processes. The processes are controlled, monitored, reviewed, and enforced. The practices are even maintained in times of stress. The following process areas should be present at maturity level 2.

- Configuration Management (CM)
- Measurement and Analysis (MA)
- Project Monitoring and Control (PMC)
- Project Planning (PP)
- Process and Product Quality Assurance (PPQA)
- Requirements Management (REQM)
- Supplier Agreement Management (SAM)

CMMI MATURITY LEVEL 3 - DEFINED A maturity level 3 organization has well-understood processes that are described in standards, tools, procedures, and methods. The organization has standard processes that are reviewed and improved over time. The major differentiators between level 2 and level 3 is the existence of standards and process descriptions. A level 2 organization will have processes that are inconsistent across projects. A level 3 organization will tailor a standard process for each project. Also, level 3 processes are described with much more rigor. In addition to the process areas found in level 2, the following process areas should be present at maturity level 3.

- Decision Analysis and Resolution (DAR)
- Integrated Project Management (IPM)
- Organizational Process Definition (OPD)
- Organizational Process Focus (OPF)
- Organizational Training (OT)
- Product Integration (PI)
- Requirements Development (RD)
- Risk Management (RSKM)
- Technical Solution (TS)
- Validation (VAL)
- Verification (VER)

CMMI MATURITY LEVEL 4 - QUANTITATIVELY MANAGED A maturity level 4 organization has quantitative measures for quality and process performance. The measures are based upon customer needs, end users, and process implementers. The quality and process performance are

understood mathematically and managed throughout the life of a project. Level 4 is characterized by the predictability of the process performance. In addition to the process areas found in level 2 and 3, the following additional process areas should be present at maturity level 4.

- Organizational Process Performance (OPP)
- Quantitative Project Management (QPM)

CMMI MATURITY LEVEL 5 - OPTIMIZING The final and pinnacle level of CMMI maturity is level 5. A maturity level 5 organization continually improves processes based upon quantitative measures. The major distinction from level 4 is the constant focus on improving and managing organizational performance. A maturity level 5 organization has well-documented standard processes that are tracked and enforced as well as a focus on continual improvement of the processes based upon quantitative measures. In addition to the process areas of the previous maturity levels, maturity level 5 should contain the following process areas.

- Causal Analysis and Resolution (CAR)
- Organizational Performance Management (OPM)

A visual description of the CMMI maturity levels can be seen in Figure 1. While CMMI-Dev does provide an excellent framework for improving a process, it is wholely focused on process improvement. It does not provide guidelines for evaluating the final product. Also, it does not provide a specify mechanism for evaluating or scoring the progression through the maturity levels. An indicator is still needed to quantify the overall performance of an organization, not just the compliance to standard processes.

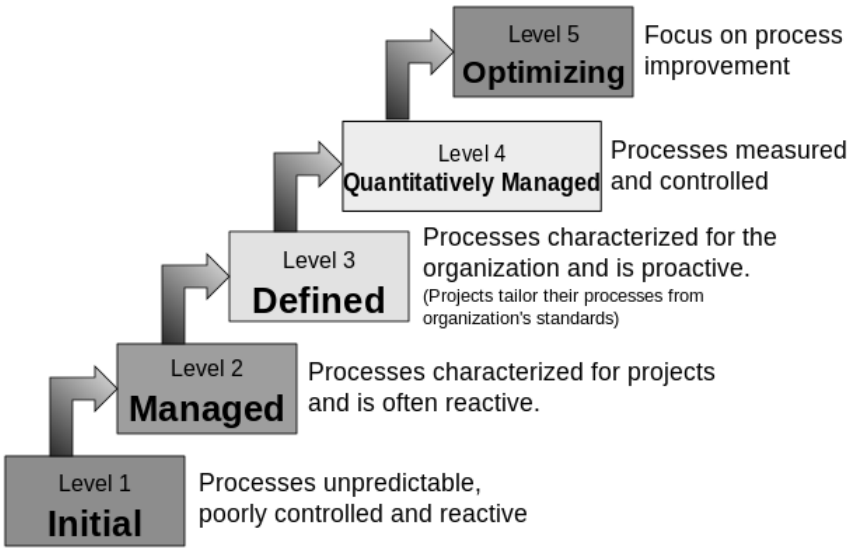

Figure 1: Characteristics of CMMI, image adapted from [29]

1.4 PREVIOUS SOFTWARE EVALUATION WORK

An example of scoring software development is presented by Jones [44]. The methodology looks for the presence of various techniques used in software engineering. The methodology provides a score based upon the productivity and quality increase of the technique being evaluated. Points are positive or negative based upon the presense of various techniques. A couple example techniques are: automated source code analysis and continuous integration. The end result is a score in range $[-10, 10]$. While the result is a single number score, it does not account for the entirety of the software development lifecycle.

Constructive Cost Model (COCOMO) is a software cost estimation model created by Boehm [7]. It combines future project characteristics with historical project data to create a regression model to estimate the cost of a software project. The original version developed in 1981 was focused on mainframe and batch processing. An unpdated version, named COCOMO II, was created by Boehm in 1995 to be more flexible for newer development practices such as desktop development, off the shelf components, and code reuse. COCOMO provides a nice

algorithm for making decisions regarding building or buying software products. It does not provide an algorithm to review and modify past performance based upon estimates. COCOMO II can be a useful tool for estimating the time and costs within an SDO, but it only provides an estimate and not an evaluation of the actual performance.

Sextant is a visualization tool for Java source code [87]. Sextant provides a graphical representation of the information related to a software system. The tool provides the capability to provide custom rules which are specific to the domain or application. However, Sextant only provides metrics and analysis of the software code. It provides no information regarding the rest of the software development lifecycle. Also, the primary output of Sextant is visual graphs. While these graphs do provide useful information, they do not provide a single number to determine the performance of the software.

Another promising research area is *process mining* [84]. As stated by Wil van der Aalst [83], "The idea of process mining is to discover, monitor and improve real processes (i.e., not assumed processes) by extracting knowledge from event logs readily available in today's systems." Creating software involves a vast amount of processes. Numerous logs of raw data are collected. One application of process mining in the area of software development was an algorithm and information for measuring a software engineering process from the Software Configuration Management (SCM) [71]. The technique creates process models to understand the process of developing software and code. It is less focused on the output and results, but it is more focused on adherence to a specified process.

Process mining has also been applied to decision making regarding software upgrades [85]. Historic and current logs can be processed and evaluated. Then small pilot groups can be offered the upgrade, and the new logs will be processed and evaluated. Thus, process mining can be beneficial for new software implementation.

More research needs to be done applying process mining to the other processes involved with software development, such as other documentation, testing, and implementation.

Although process mining can be useful for analyzing software development data, it has not yet been applied to the entirety of a software development organization. It also does not focus on the results of the organization. It focuses on process conformance, which can be very important, so it is limited in the ability to evaluate the entire organization.

Much work has been done to determine metrics for source code, in fact entire books have been written on the topic of software metrics [42], [66]. Yet, organizations still struggle to measure the production of software. Little work exists for scoring the entire software development organization.

1.4.1 SEMAT

Software Engineering Method and Theory (SEMAT) is claiming to be the "new software engineering" [40]. The authors rightfully claim that software engineering lacks an underlying theoretical foundation found in other engineering disciplines. This lack of theory has led software engineering to not be engineering, but rather a craft. The goal of SEMAT is to merge the craftsmanship and engineering to provide a foundation for software engineering. The primary initiative of SEMAT has been the creation of a kernel for software engineering. The kernel is the minimal set of things common to all software development endeavours. The three parts to the kernel are:

Measurement - There must exist a means to determine the health and progress of an endeavour

Categorization - The activities must progress through categories during an endeavour

Competencies - Specific competencies will be required for completing activities

The kernel defines alphas, which are seven dimensions with specific states for measuring progress. The seven dimensions are:

1. Opportunity
2. Stakeholders
3. Requirements
4. Software Systems
5. Work
6. Team
7. Way of Working

Although SEMAT is very promising, the development is not yet complete. Adoption is limited so the technique has not been validated on many actual software engineering endeavours. Although SEMAT does include a part for measurement of progress, it does not specify how the measurement is to be performed.

1.4.2 SOFTWARE QUALITY

Software quality is one of the most well studied aspects of software development. Most of the work focuses on either the problems with the software or the source code. Quality and the number of problems with the software are inversely related, more problems means lower quality. Quality is easy to measure, but that measurement is usually very software specific. It is easy to find that some software has X number of problems, but it is nearly impossible to determine whether the quality of that software is better or worse than some other software with X defects.

One piece of software can be larger[1] or more complex. Thus, finding a value for quality is easier than interpreting that value. No matter the interpretation, the goal is to release the number of problems with the software. Top 10 lists have been created for techniques to remove problems from software [10]. A number of different techniques or best practices for preventing defects have been proposed [26]. These are all strategies to identify or remove the problems before the software is completed and released to users.

Another aspect of software quality is the complexity of the source code. More complex code results in more maintenance efforts and more chances for problems. A couple of numerical measures for the complexity of source code have been created. The most common examples are McCabe [58] and Halstead [32]. However, the measures on source code only explain part of the software development lifecycle.

Another measurement of quality can be the cost per defect, also known as the cost to fix a problem. As seen in [46], this measurement has problems because the lowest cost per defect will occur on software with the most problems. Therefore, the lowest cost per defect is actually the lowest quality as well. Due to this difficulty and others, a number of other models have been created for evaluating the quality of software [61]. While all of the models have merit in certain situations, the measures of quality must be combined with other measures in order to provide an overall evaluation of a software development organization.

1.4.3 SOFTWARE ANALYTICS

One area of research that is focusing on the evaluation of software development organizations is *software analytics*. Software analytics is less focused on evaluation and more so on all sorts of analysis of software data. The earliest variants of

[1] Saying a piece of software is larger can be a rather arbitrary statement. It can mean the software requires more computing time, has more lines of code, more documentation, more hours spent on development, or some other arbitrary measure.

software analytics were disguised as applications of data mining techniques to software engineering data in the late 1990s [24], [30], [72]. Later the field began to emerge more heavily, but still remained primarily methods of data mining applied to software engineering [31], [49], [80], [88]. Finally, The term software intelligence was proposed for the field of study [34], but eventually the term software analytics became the dominant term for referring to the field of study [12], [89].

The goal of software analytics is to extract insights from software artifacts to aide practitioners in making better decisions regarding software development [90]. The three main focus areas of software analytics are:

1. **User Experience** - How can the software enable the user to more easily or quickly accomplish the task at hand?
2. **Quality** - How can the number of problems with the software be decreased?
3. **Development Productivity** - How can the processes or tools be modified to increase the rate at which software is produced?

Later, Martin Shepperd in [33] identified 3 important questions that software analytics must address:

1. "How much better is my model performing than a naive strategy, such as guessing [. . .]?"
2. "How practically significant are the results?"
3. "How sensitive are the results to small changes in one or more of the inputs?"

These are 3 important questions that should be addressed when presenting any results in software analytics. The research needs to demonstrate clear advantages for practitioners. The work presented in this work will address both the 3 main focus areas and the 3 important questions of software analytics.

Lavazza, Morasca, Taibi, and Tosi focus specifically on the source code; analyzing the complexity, size, and coupling [81]. They created a theoretical framework for dynamic measurements instead of traditional static measures. Letier and Fitzgerald discuss how to choose the correct tools and techniques to analyze software data [55]. A goal model is produced that matches the data analysis methods with the goals of the software stakeholders. The method does not focus specifically on analysis of the development of software.

Software Development Analytics is a subfield of software analytics [60]. It focuses specifically on the analytics of the development of software, however not the overall performance of the software. Hassan points out in [33] that software analytics needs to go beyond just the developers. Everyone and everything involved in the development of software produces some data and that data can be meaningful. The insights from non-developer data has the potential to yield important results as well. Software development produces many valuable pieces of datum that can be analyzed [57]. Just a few of the pieces of datum are: email communication, bugs, fixes, source code, version control system histories, process information, and test data. Examples of this type of data can be found in the PROMISE Data Repository [59].

All of these techniques are of no use if the correct data is not available. Therefore, it is important to identify the information that is needed to properly perform software development analytics. Unfortunately, there are vast amounts of information that need to be collected to meet the analytic needs of developers and managers [13]. When the data and tools exists, the analytics should help an organization with the following tasks.

- Evaluate a project
- Determine what is and is not working
- Manage Risk

- Anticipate changes
- Evaluate prior decisions

In order to store the data, appropriate tools are needed. Microsoft is working on developing some tools for the analysis of the development of software [18], [90]. Microsoft has developed StackMine, a postmortem tool for performance debugging, and CODEMINE, a tool for collecting and analyzing software development process data. Both tools provide analytical insight for various aspects of the software development process, however neither tool covers all aspects of software development. These tools are currently early in development and the adoption of the tools by practitioners is still unknown. One of the reasons for the slow adoption of new tools is the inherent difficulty of producing new tools for the software development process [78]. A tool that works fantastic for one team might not automatically apply to another team. The people creating the tools need to be acutely aware of the needs of the technical practitioners that will be using the tools.

Iqbal, Ureche, Hausenblas, and Tummarello introduced a methodology named Linked Data Driven Software Development (LD2SD) which is a collection of various software artifacts into linked data [39]. This is one of the original attempts to collect software engineering data. The methodology links version control, discussion forums, and issue tracking data. The result is web-scale integration of data, but the actual benefits are still uncertain.

After the proper tools are in place to collect the necessary data on software development and software analytics are being p roperly implemented, an obvious next step is the application of gamification to software development. Gamification is "the process of making activities more game-like" [86]. Some of the benefits of gamification are higher productivity, competition, and greater enjoyment. Prior attempts at gamification of software development focus only on the computer programming phase [76]. Others focus on defining a framework for gamification

within the software development process [41]. There are even some indications that gamification might help increase software quality [23]. While this work will not focus on gamification, it is important to note that an implementation of an evaluation technique for software development could be implemented simultaneously with a gamification strategy. Both will require new collections of data and new reporting.

Overall, there exist many attempts to evaluate portions of the a software development organization. None of the the attempts provide a single number score for the entirety of the organization. Most of the techniques focus on specific portions of the software development lifecycle, namely the development portion. Plus, there are many tools that need to be created for software analytics to provide all the value that it promises.

1.5 ORGANIZATION OF THE WORK

The remainder of this dissertation is divided into 5 chapters. The next chapter provides an overview of software, software development lifecycles, software engineering, and software development organizations. Chapter 3 introduces what is meant by the term data-driven software engineering. Chapter 4 then provides an explanation of the Cumulative Result Indicator (CRI). It will present the essential elements for calculating the CRI, as well as the formulas, framework, and data necessary to produce the CRI. Chapter 5 provides a technological framework for generating and storing the current and historical CRI values. Chapter 6 demostrates how CRI can be implemented in a software development portion of a large financial institution. Chapter 7 concludes the dissertation with a summary of the results and some possible future directions for further enhancements.

2 A SOFTWARE DEVELOPMENT ORGANIZATION (SDO)

A *Software Development Organization (SDO)* is any organization or subset of an organization that is responsible for the creation, deployment, and maintenance of software. Many times a software development organization is a company that produces software. Other times, a software development organization is contained within the Information Technology department of a larger organization. Some of the job roles with an SDO are: software engineer, system administrator, software quality analyst, programmer, database administrator, and documentation specialist.

2.1 WHAT IS SOFTWARE?

Numerous definitions can be found for the term *software*. Software is more than just computer programs. According to Ian Sommerville [77], "Software is not just the programs but also all associated documentation and configuration data which is needed to make these programs operate correctly." This is the definition used for the remainder of this work.

2.2 THE SOFTWARE DEVELOPMENT LIFE CYCLE

The discipline of software engineering has created a workflow for developing software. This workflow is called the *Software Development Life Cycle (SDLC)*. SDLC can be defined as [73]:

> [...] a conceptual framework or process that considers the structure of the stages involved in the development of an application from its initial feasibility study through to its deployment in the field and maintenance.

While the SDLC states what needs to be done, there are numerous models that formalize exactly how to perform the SDLC. The models contain steps that are commonly referred to as a phases. A few of the popular models are described below.

2.2.1 WATERFALL

The waterfall model is the oldest and most influential of the SDLC models. It was first presented at a Navy Mathematical Computing Advisory Panel in 1956 by Herb Benington [6]. Figure 2 shows the model Benington outlined for producing large software systems. In 1970, Benington's model was modified by Royce [70]. Royce produced an updated version of the diagram seen in Figure 3 which provides some loops to go back to a previous phase in the workflow.

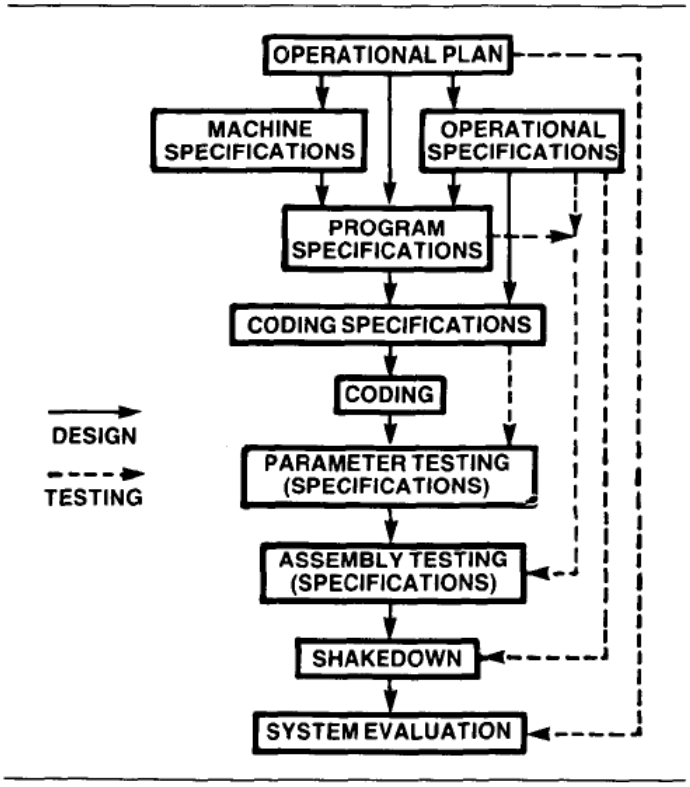

Figure 2: Benington's original diagram for producing large software systems, adapted from [6]

The modern version of the waterfall model specifies that each phase needs to be entirely completed before moving onto the next phase. Some small amount of overlap is permitted and looping occurs but both actions are discouraged and should be limited. A modern diagram of the waterfall model can be seen in Figure 4.

Waterfall has some excellent features such as: simple to understand, easy to

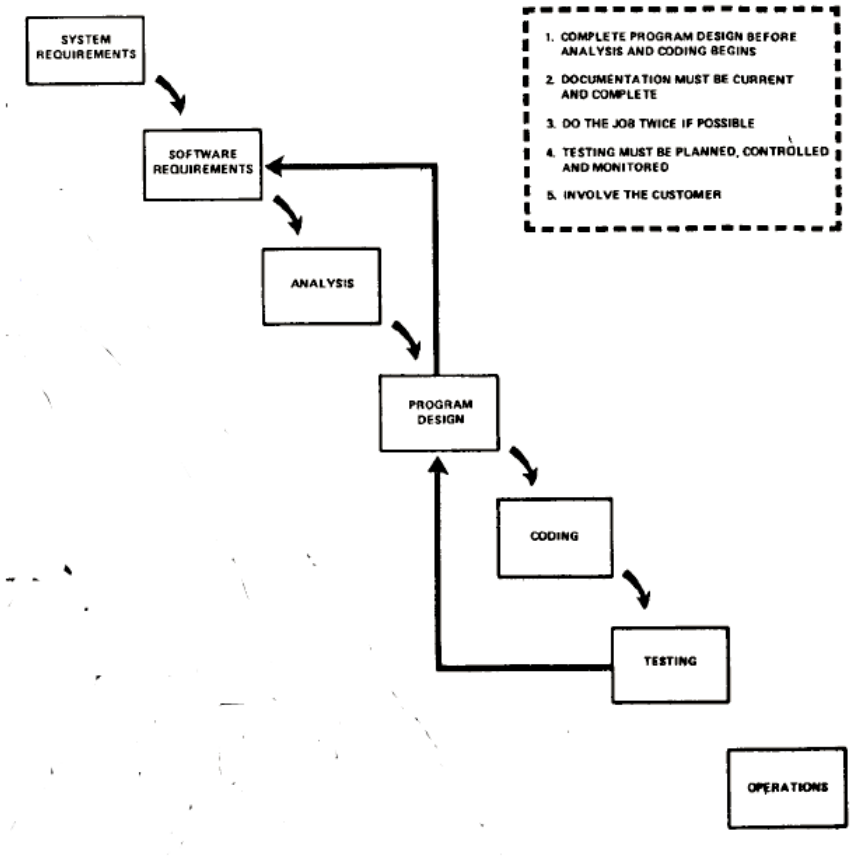

Figure 3: Royce's version of the waterfall model for producing software systems, adapted from [70]

plan, and well-defined phases. However, waterfall lacks the flexibility required of many software systems built today [56]. Due to the fact the phases are so sequential, it makes changes during the life cycle difficult and expensive if not impossible. Therefore, other models of SDLC have been created to address the lack of flexibility of the waterfall model. Notice, the other models are adaptations of waterfall.

2.2.2 SPIRAL

The spiral model for software development was presented by Boehm in 1986 [8], [9]. The goal of the spiral model of software development is very risk-driven. A software project will start with many small and quick iterations. Each iteration will cover

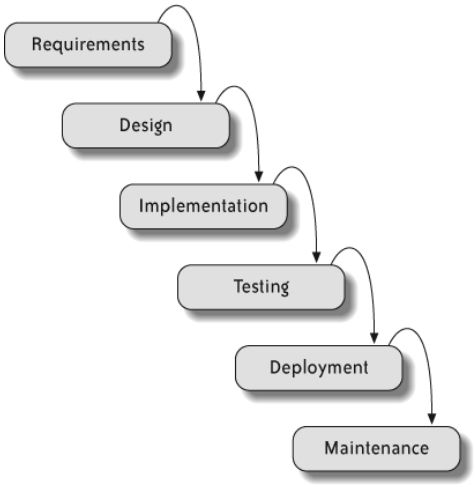

Figure 4: Modern Waterfall Diagram, adapted from [35]

the following 4 basic steps.

1. Determine Objectives
2. Identify Risks
3. Develop and Test
4. Plan Next Iteration

This model allows software to be built over a series of iterations without risking too much time or effort in any single iteration. Sprial requires a very adaptive management approach as well as flexibility of the key stakeholders [73]. It can also be difficult to identify risks that will occur in future iterations. Figure 5 provides a bit more detail on the iterations and the overall process.

2.2.3 AGILE

Agile software development has arisen due to the inability of the waterfall and other models to adjust to changes during the development cycle. Agile software development is a group of SDLC models that operate under the influence of the following four key principles [5].

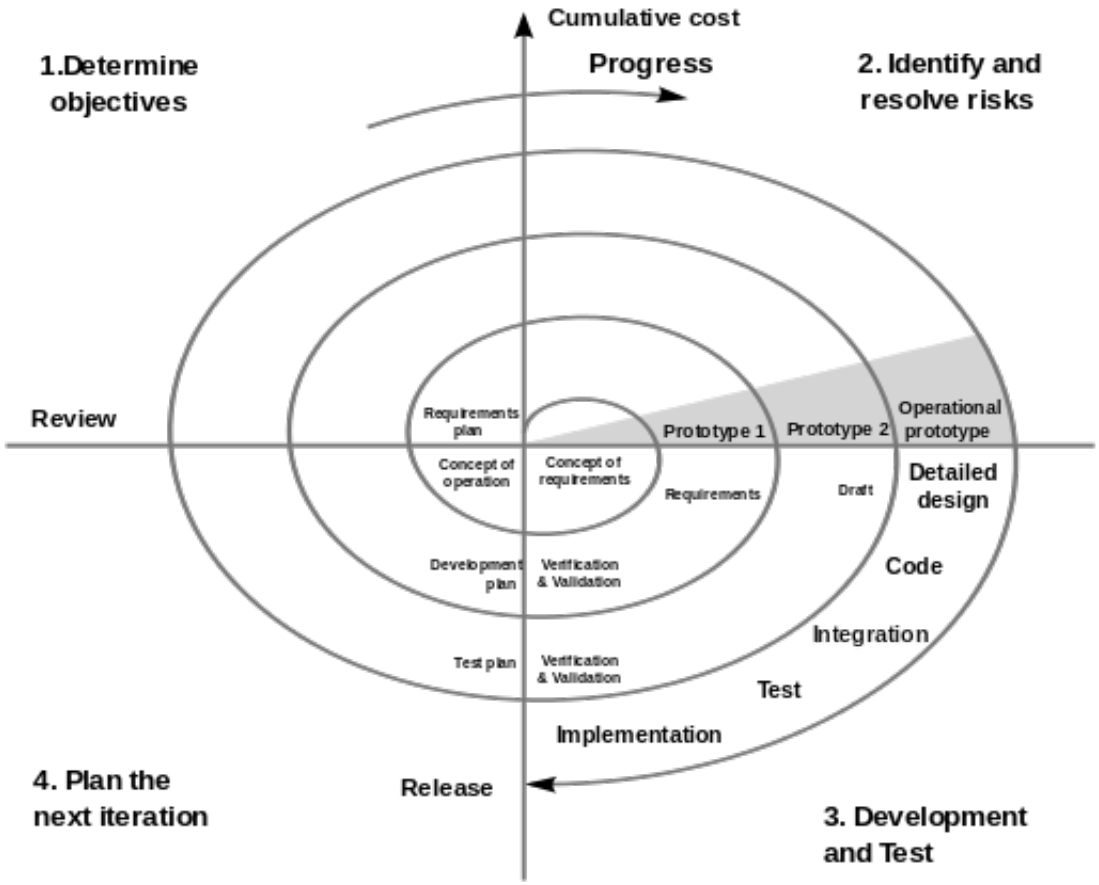

Figure 5: Spiral SDLC Model [11]

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

Agile does not specify an implementation, but some specific models of agile SDLC are: eXtreme Programming, Scrum, Lean, Kanban and others [35], [62]. Agile models are very popular in many of today's software development organizations because the models work well for dynamic quickly changing applications such as web-based applications. Startups have largely adopted the Lean methodology for its

ability to identify a minimum viable product[2] and reduce the time to market [28].

2.2.4 SDLC COMMONALITIES

Even with the large number of SDLC models currently being used by different SDOs, many commonalities exist among the models. The commonalities can be tied back to the steps of waterfall. All of the models exhibit, to some degree, the following phases. The only major difference is the scope, size, and duration of each phase. For example, the spiral model spends less time in each phase. The agile models produce less documentation and focus more on the implementation phase. Here are the common phases in nearly all SDLC models:

1. Requirements

The first phase is involved with defining what the software must do. Each piece of functionality is considered a requirement.

2. Design

Before writing any code, the necessary infrastructure and involved software systems must be identified. This phase can serve as a roadmap for the remaining phases. If done properly, this phase can greatly help the later phases.

3. Implementation

Often the only phase of the SDLC that is measured, this is the phase where the actual computer code is written.

4. Testing

This phase validates the expected functionality. Also, testing attempts to discover unexpected side affects of the software.

[2] A *minimum viable product* is a reduced version of software that contains only the bare minimum functionality required to meet the requirements.

5. Deployment and Maintenance

All software must be correctly deployed and maintained. This phase is the most expensive and lengthy phase of the software development life cycle.

These 5 steps cannot cover everything that needs to be accomplished during a project. They just provide of rough guideline of what needs to be completed in order to ensure a more successful software release. Appendix A contains a more detailed list of the tasks necessary to complete the software life cycle.

2.3 WHAT IS SOFTWARE ENGINEERING?

Software Engineering as a term dates back to the 1968 North Atlantic Treaty Organization (NATO) conference [63], [82]. Over the years many definitions have been provided. IEEE provides a definition that encompasses many of the other definitions. IEEE ISO610.12 defines software engineering as, "The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software" [38].

Software engineering has struggled to determine the correct projects to complete [21]. Software projects are commonly behind schedule and over budget [14], [37], [47], [54] with nearly 20% of projects in the United States still failing [25]. As of 2015, Software engineering is still awaiting the professional status of more established fields such as medicine, law, and general engineering [45]. Organizations need a better technique to understand the past performance so they can better predict the future performance. Proper measurement will be essential to solidifying software engineering as certified profession.

3 MEASURING AN SDO

Anything can be measured [36]. Thus, an SDO can be measured. Proper measurement is crucial for improvement because without a starting point it is

impossible to determine progress. Also, consistent reporting is essential for tracking historical performance.

The SDLC, like any process, needs to be properly measured. In order to accomplish proper measurement, three activities need to occur [27].

1. Identify Process Issues
2. Select And Define Measures
3. Integrate with Software Process

This work will focus on steps 1 and 2. The process issue is the overall effectiveness of the SDO. Section 4 will cover step 2 as it relates to an SDO. Step 3 will be different for each SDO, but Section 5 provides a bit of guidance for storing the correct information. It is up to the specific SDO to determine how and when the information is being stored.

Many methods have been used in the past to measure and evaluate SDOs. Some of the common methods will be explained in the following sections.

3.1 METRICS

A *metric* can be defined as a means of telling a complete story for the purpose of improving something [52]. Metrics are frequently indirect measurements and are very common in the measurement of SDOs. The following are some examples of metrics that can be collected for an SDO.

- SLOC - The number of Source Lines of Code
- NOM - The Number of Methods per class
- Complexity - A numerical measure of the code complexity, some common examples are McCabe [58] and Halstead [32]

- Design - The amount of coupling and cohesion present in the software code
- Source Code Analysis - Tools that determine whether code adheres to specified set of rules. Common examples are PMD[3] and FindBugs^{TM} [4], [16].

All these metrics are beneficial, but none of them tell the story of the entire SDO. Most of the metrics, as seen in the list above, for an SDO focus on the source code and development phase. Since metrics are indirect, it can be very difficult match SDO performance with a metric or series of metrics. Metrics are great for tracking, but decision making based upon metrics alone is difficult. That is why many of the other techniques build upon metrics to provide a more complete overall picture of an SDO. Metrics are great starting point, but more is needed to properly evaluate performance.

3.2 INDICATORS

Another common measurement technique is indicators. An *indicator* is simply a performance measure. Typically, a number of indicators will be placed together and displayed in some report or on some dashboard. Indicators can be crucial measurements within any business setting, and an SDO is no exception. Determining the correct indicators for an organization can be difficult, and many organizations incorrectly classify the indicators [65]. The differences between the indicators will be explored and possible measures for each indicator in an SDO will be presented. The four types of indicators important to an SDO are shown in Table 1.

Performance Measure	**Description**
RI (Result Indicator)	What has been done?
KRI (Key Result Indicator)	How you have done?

[3]PMD is a source code analysis product. It is not an acronym.

Performance Measure	Description
PI (Performance Indicator)	What to do?
KPI (Key Performance Indicator)	How to dramatically increase performance?

Table 1: INDICATORS

Indicators can be used in just about every organizational setting from businesses to non-profit organizations. They are not unique to SDOs, and the exact indicators to track is very specific to the organization. The indicators chosen by one organization might not be the same as the indicators chosen by another organization. The following sections will explain the type of indicators in more detail and provide some examples for an SDO.

3.2.1 RESULT INDICATORS (RI) FOR AN SDO

Result Indicators are performance measures that summarize activity. All financial performance measures are result indicators. Result indicators are measured on a timely basis (daily, weekly, monthly) and are the result of more than one activity. They do not tell staff what needs to be done to improve the RI. For an SDO, some possible RIs are seen in Table 2.

Result Indicators
Requirements Implemented Per Month
Monthly SLOC
New Weekly Users
Monthly Development Hours
Webpage Views Yesterday
Monthly Development Hours
Monthly Server Uptime

Result Indicators
Quartely Software Sales

Table 2: RESULT INDICATORS FOR AN SDO

3.2.2 KEY RESULT INDICATOR (KRI) FOR AN SDO

Key Result Indicators are measures of multiple activities that give a clear picture of whether the organization is traveling in the right direction. Unfortunately, KRIs are commonly mistaken for KPIs [65]. KRIs do not tell an organization what is needed to improve the results. KRIs are highly benefical for high-level management and not necessarily beneficial for staff working directly on the software. For an SDO, some possible KRIs are seen in Table 3.

Key Result Indicators
Customer Satisfaction
Net Profit
Money Spent on Fixing Software
% of New Features vs. Fixes
Time on Website
% Servers Meeting the Expected Availability

Table 3: KEY RESULT INDICATORS FOR AN SDO

3.2.3 PERFORMANCE INDICATOR (PI) FOR AN SDO

Performance Indicators are nonfinancial performance measures that help a team align themselves with the organizations strategy. These are important to the organizations success, but they are not the key measures that will lead to drastic

improvement. They are specifically tied to a team and all staff understand what actions need to be taken to improve the PI. For an SDO, some possible PIs are seen in Table 4.

Performance Indicators
% Test Coverage[4]
# of Project Defects from Key Customers
Requirements Scheduled for Next Release
# of Missed Requirements
% of Late Projects

Table 4: PERFORMANCE INDICATORS FOR AN SDO

3.2.4 KEY PERFORMANCE INDICATOR (KPI) FOR AN SDO

Key Performance Indicators are performance measures focusing on critical aspects for current and future organizational success. Notice, KPIs are not focused on historical performance, and they clearly indicate how to drastically increase performance. KPIs allow a team to monitor current performance and quickly take action to correct future performance. KPIs cover a shorter time frame than KRIs. KPIs consist of the following 7 characteristics [65].

1. Not financial
2. Measured frequently (hourly, daily, weekly)
3. Acted on by CEO[5] and/or senior management
4. Clearly indicate the action required

[4]Test Coverage is simply the percentage of the code that is being tested. Ideally, this number would be 100%, but higher is better.

[5]Chief Executive Officer

5. Tie responsibility to a particular team
6. Have a significant impact
7. Encourage appropriate action

Antolic in [2] made one of the earliest attempts to identify and measure the KPIs for an SDO. Antolic's strategy focused around 7 KPIs.

1. Schedule adherence
2. Assigned content adherence
3. Cost adherence
4. Fault slip through
5. Trouble report closure rate
6. Cost per defect

However, according to the definition of KPI just presented, they are not really KPIs but instead KRIs. That is because none of the 7 clearly indicate the action required to improve the measure. Also, it is unclear if improving any of the measures will drastically improve performance.

For an SDO, Table 5 identifies some more appropriate KPIs.

Key Performance Indicators
Projects more than 20% behind schedule
Servers currently unavailable
Automated tests failing for more than 24 hours
Projects with test coverage less than 60%
Projects with more than 10 SIT defects
Unfixed, high priority PROD defects older than 1 week

Key Performance Indicators

Table 5: KEY PERFORMANCE INDICATORS FOR AN SDO

3.3 BALANCED SCORECARD

Developed in 1992 by Robert S. Kaplan and David P. Norton, the *balanced scorecard* is a set of measures that give management a quick and comprehensive view of the organization [50]. Originally created as an extension to the already existing financial measures, the balanced scorecard expanded the measures to include: customer focus, internal process, and learning/growth. This gave organizations a more comprehensive view that was not strictly financial. It allows an organization to focus on long-term strategic goals instead of just short-term goals. As a result of the strategic focus, the balance scorecard rapidly gained widespread adoption among businesses [51].

In 2010, David Parmenter [65] added 2 more characteristics to the balanced scorecard: employee satisfaction and environment/community. This results in a total of 6 characteristics for the balanced scorecard.

1. Financial
2. Customer Focus
3. Internal Process
4. Learning and Growth
5. Employee Satisfaction[6]
6. Environment/Community[6]

[6] Added later by Parmenter [65].

Balanced scorecards are great for easily displaying the important information about an organization. The downside is a balanced scorecard does not specify what exactly needs to be tracked. It can be very difficult to determine exactly what PIs, RIs, KRIs and/or KPIs to track in a balanced scorecard. It just specifies 6 broad categories. It is also not specific to an SDO and it does not produce a single number. However, any new measurement technique for an organization should be compared with the balanced scorecard.

3.4 PROJECT MANAGEMENT MEASUREMENT

A final strategy to measure an SDO is focused on the aspect of project management. Project management is the guidance applied to a project to ensure an effective and efficient completion. Proper project management will ensure all steps of the SDLC continue to progress and all obstacles are handled in a timely fashion. According to Putnam and Myers in [66], the 5 core measurements for managing software projects are:

1. **Quantity of function** usually measured in terms of size (such as source lines of code), that ultimately execute on the computer
2. **Productivity** as expressed in terms of the functionality produced for the time and effort expended
3. **Time** the duration of the project in calendar months
4. **Effort** the amount of work expended in person-months
5. **Reliability** as expressed in terms of defect rate (or its reciprocal, mean time to defect)

A *process productivity* number is calculated based entirely off aspects of the SDLC and the 5 core measurements. It is a number targeted at project teams

working on the SDLC.

3.5 A SIMPLER MEASUREMENT

It is important to note that SDOs do not just develop software. An SDO has many other duties including: deploying software, installing server hardware/software, writing documentation, surveying users, research, innovation, education and other common business duties. Thus it is important to measure as many duties as possible.

How can the PIs, RIs, KRIs, and KPIs be combined to form a single value called the CRI (Cumulative Result Indicator)? If the indicators are targeted for upper management to understand performance, then KPIs are not the correct indicators. KPIs are targeted towards immediate action and future performance. RIs and KRIs are the most beneficial for upper management to gauge how an organization is doing. However, with so many possible RIs and even KRIs, it can be tricky to gain a quick understanding. The next section will present and explain a technique to combine KRIs into a single number for immediate and effortless evaluation of an SDO.

4 CUMULATIVE RESULT INDICATOR (CRI)

Software development organizations struggle to measure overall performance. The *Cumulative Result Indicator (CRI)* is an algorithm to provide a single number score to measure the performance of a software development organization. It works by statistically analyzing the past performance of the organization and using that information to score an organization on current performance. CRI is a collection of the following 5 elements, which are actually KRIs, for a software development organization.

1. **Quality**

2. **Availability**
3. **Satisfaction**
4. **Schedule**
5. **Requirements**

A separate CRI score is calculated for each element and then aggregated together to form an overall CRI score. A new CRI score will be calculated based upon the selection of a given time period (weekly, monthly, quarterly). **CRI is not meant to be comparative between organizations, but to measure the amount of increase or decrease a single organization exhibits across elements**. CRI is made to be easily expandable to other elements if desired.

The scores for CRI will range from -1, indicating the worst performance, all the way to +1, indicating perfection. A score of 0 is an indication of meeting the basic expectations. A negative score indicates under-performance and a positive score indicates over-performance. Here are some examples. An CRI score of 0.35 means the organization is performing 35% better than expected. Conversely, a score of -0.15 means an organization is performing 15% worse than expected.

Below are the attributes of the CRI scoring.

- The range of scores must have equal values above and below 0
- The minimum score must equate to the worst possible performance, however that is defined
- Similarily, the maximum score must equate to the best possible performance.
- A score of 0 must be average (or expected) performance
- All individual elements must have the same scoring range

As long as those 5 features are met, the range of scores can be anything. The range of $[-1, 1]$ was chosen because it is easy to scale to a different range such as $[-10, 10]$ or $[-100, 100]$. Thus scaling can be applied to obtain values in any appropriate range. The scaling factor is denoted with the variable k. The scale must be the same for all 5 elements.

4.1 ELEMENTS OF CRI

Each of the 5 elements of CRI has its own set of data that needs to be collected and a formula for calculating a score. These five elements will be outline in the next sections.

4.1.1 QUALITY

Measuring quality is a crucial part of accessing software development results. Poor quality means time, money, and resources are spent fixing the problems. As a result, new features are not being created. One of the key indicators of software quality is defects. It is important to measure the number of defects associated with a software release because industry-wide the current defect removal rate is only about 85% and this value should be increased to about 95% for high quality software [43]. Organizations are leaving too many defects unfixed. If organizations could lower the number of defects, then not as many defects would need to be fixed, which in turn would raise the defect removal rate.

Another aspect of defects is severity levels. A severity level indicates the importance of a defect that has been discovered. Although an organization can choose whatever severity levels they choose, it is common practice to use 5 severity levels [68]. The most severe level for a defect is 1. All other levels drop in severity from that point. Table 6 describes the 5 levels for defect severity.

Level	Description
1	Software is unavailable and no workaround
2	Software performance degraded with no workaround
3	Software performance degraded but workaround exists
4	Software functions but a loss of non-critical functionality
5	Others: minor cosmetic issue, missing documentation

Table 6: SOFTWARE DEFECT SEVERITY LEVELS

QUALITY DATA In order to properly score the quality of an SDO, certain data needs to be obtained in order to measure performance. Table 7 identifies the columns of data that will be used to create a score for the quality element of CRI. Each column is classified as *required* or *optional*. That is to allow some flexibility in the model for organizations that collect varying amounts of data.

Column Name	Data Type	
Application ID	String (factor)	Required
Frequency Date	Date	Required
Development Effort	Integer	Required
Testing Effort	Integer	Optional
SIT Defects	Integer	Optional
UAT Defects	Integer	Optional
PROD Defects	Integer	Required

Table 7: QUALITY DATA NEEDED FOR CRI

The development and testing effort can come from any of the following choices for effort. It is possible that other measures will work for effort.

Actual Time This number is a representation of the total amount of time spent on a project. This number can be measured in any unit of time: hours, days, weeks, etc. Actual time can be applied to development or testing effort.

Estimated Time This number is a representation of the initial estimated amount of time spent on a project. This number can be measured in any unit of time: hours, days, weeks, etc. Estimated time can be applied to development or testing effort. It is common for the estimated and actual times to be diffent.

Source Lines Of Code (SLOC) This number is the count of the total number of lines of source code for a project. Obviously, this item only counts as a level of effort for development unless coding in used to generate automated test cases. [7]

Modified Lines Of Code This number is a count of the number of modified lines of source code. Modified lines is defined as the number of deleted, added, and modified lines of source code. This number is different from above since it does not include all the lines of source code. Similar to above, this number makes more sense for development effort.

Test Cases A test case is a step or series of steps followed to validate some expected outcome of software. Organizations will create a number of test cases to be validated for a software system. The number of such test cases could be used as a level of testing effort.

Notice the data does not include a severity level. The severity level should be handled before being stored. A good technique is to count the defects based upon the weighting scheme in Table 8 [68]. For example, finding 1 defect of severity level

[7] Automated testing is the process of creating software to automatically run tests against other software. The adoption of automated testing is varied and it is not a solution in all cases [67].

5 will result in total count of 1. However, finding 1 defect of severity level 2 will result in a total count of 15. This strategy helps to standardize the number of defects found. An organization can alter the values of Table 8 based upon priorities or use a different technique if desired. It is just important to get a standard, meaningful number for SIT defects, UAT defects, and PROD defects which manages severity appropriately.

Severity Level	**Weight**
1	30
2	15
3	5
4	2
5	1

Table 8: DEFECT SEVERITY LEVEL WEIGHTING

QUALITY FORMULA The first step in creating a score for the quality element is analysis of the historical data to create a baseline function. The historical data is all quality data collected before a given point in time. Some common historical cutoffs are the current date or the end of the previous fiscal year. Then a mathematical model, called the *baseline quality function*, to predict PROD Defects will be produced. In statistical terms, the response is *PROD Defects* and the predictors are: *UAT Defects*, *SIT Defects*, *Testing Effort*, and *Development Effort*. Some of the following strategies will be employed to find a reasonable model.

- Removal of outliers and/or influential points
- Linear Regression
- Stepwise Regression

- Ridge Regression for suspected multicolinearity

Once a model has been found, it will be labeled as f and it will not change. The function f can be the same for all Application IDs or it can be different for each Application ID or any combination of Application IDs. It serves as the quality baseline for CRI. All future quality scores will be dependent upon the original f. Once set, the model will not change.

After the model f has been determined, it is time to calculate the quality score for each application ID within the given time period. The quality score for each Application ID can be calculated as follows.

$$S_{1_i} = \left\{ \begin{array}{ll} \text{where } f_i \geq d_i & : \frac{f_i - d_i}{f_i} \times k \\ \text{where } d_i > f_i & : \frac{f_i - d_i}{6\sigma_i} \times k \end{array} \right] \text{, calculate quality score for each app } i$$

where

- S_{1_i} is the quality score for Application ID i
- n is the number of Application IDs
- d_i is the actual PROD defects
- f_i is the function to predict PROD defects for Application i based upon *UAT Defects*, *SIT Defects*, *Testing Effort*, and *Development Effort*
- σ^2 is the estimated variance

Then the overall quality score is calculated as below.

$$S_1 = \sum_{i=1}^{n} w_i' S_{1_i} \text{ where } \sum_{i=1}^{n} w_i' = 1$$

where

- S_1 is the combined quality score for all Application IDs, a weighted average

Then S_1 represents the quality score for that given time period.

4.1.2 AVAILABILITY

All the new requirements and great quality do not matter if the software is not available. All the new features and great quality do not matter if the software is not available. Thus it is essential to set an expected Service Level Agreement (SLA)[8] and measure performance against that SLA. The following section will outline to data needed to properly calculate an SLA and the data needed to calculate the CRI score for availability.

Special Note: The Service ID for availability does not have to be the same as the Application ID for quality or any of the other elements. Some organizations have a one-to-one mapping between Applications being developed and services being deployed. Others have more complex sceranios that require multiple applications to be combined to form a service. Then the availability of the system is tracked.

AVAILABILITY DATA Table 9 identifies the necessary data to calculate the CRI element score for availability. Notice the three optional fields: *Uptime*, *Scheduled Downtime*, and *Unscheduled Downtime*; they are optional because they can be used to calculated the *Percent Uptime*. The *Percent Uptime* is the important value for the CRI schedule score. Here are the two common approaches for calculating percent uptime:

The preferred method:

$$\text{Percent Uptime} = \frac{\text{Uptime}}{\text{Uptime} + \text{Scheduled Downtime} + \text{Unscheduled Downtime}}$$

[8]For an SDO, the SLA is a contract specifying the amount of time software will be available during a given time period.

and the alternative method:

$$\text{Percent Uptime} = \frac{\text{Uptime}}{\text{Uptime + Unscheduled Downtime}}$$

The only difference is the removal of scheduled downtime from the calculation. The calculation approach is typically specified in the contract associated with the SLA. Thus, the Percent Uptime is important and it can either be supplied in the data or calculated from the optional fields.

Column Name	**Data Type**	
Service ID	String	Required
Frequency Date	Date	Required
Uptime	Float	Optional
Scheduled Downtime	Float	Optional
Unscheduled Downtime	Float	Optional
Percent Uptime	Float	Required
Expected Percent Uptime	Float	Required

Table 9: AVAILABILITY DATA NEEDED FOR CRI

AVAILABILITY FORMULA The formula for availability is more straightforward than the quality formula. It does not include any analysis of the historic data. That lack of historical analysis is avoided since the SLA provides an existing baseline to measure against. The following formula is simply a percentage the SLA was exceeded or missed.

$$S_{2_i} = \begin{cases} \text{where } A_{a_i} \leq A_{e_i} & : \left[\frac{A_{a_i} - A_{e_i}}{A_{e_i}} \times k\right] \\ \text{where } A_{a_i} > A_{e_i} & : \left[\frac{A_{a_i} - A_{e_i}}{1 - A_{e_i}} \times k\right] \end{cases}, \text{ calculate availability score for each app } i$$

where

- A_{a_i} actual availability for System ID i
- A_{e_i} expected availability for System ID i

Then the overall availability score is calculated as below.

$$S_2 = \sum_{i=1}^{n} w_i' S_{2_i} \text{ where } \sum_{i=1}^{n} w_i' = 1$$

4.1.3 SATISFACTION

The satisfaction of users, customers, and/or business partners is the third element to be measured. This element is important because in an established business, retaining customers is less expensive than attracting new customers [3]. Depending upon the type of SDO, the customers may be internal or external to the organization. For the remainder of this section, the term customer will be used to represent any person who is responsible for guidance, decision-making or use of the software. The term customer can refer to a: user, paying or nonpaying customer, internal or external business partner, or any other person deemed influential to the development of the software.

If one element of CRI was to be rated as the most important, satisfaction would be it. Without satisfied customers, the rest of the measures do not matter. For example, having a quality application that is always available does not matter if the application is not what the customer wants.

Surveys are used to measure satisfaction for CRI. A series of statements will be presented to all or a subset of the customers. Any customer that chooses to respond to the survey is considered a respondent. A respondent can rate statements based upon a Likert Scale[9] with a numerical response where the minimum value

[9]"The Likert Scale presents respondents with a series of (attitude) dimensions, which fall along a continuum." [17]

indicates maximum disagreement and the maximum value indicates the maximum agreement. Common rating scales would be from 1 to 5 or from 1 to 3. An example survey can be seen in Table 10.

ID	Statement	Disagree	Neutral	Agree
1	I find the software easy to use.			
2	I would recommend this software to others.			
3	The software makes me more productive.			
4	I am happy with this software.			

Table 10: SAMPLE SURVEY FOR SATISFACTION

ISSUES WITH SURVEYS Surveys present a number of challenges that need to be presented and briefly discussed. Here are some of the issues that need to be addressed when using surveys.

Text

The specific text used in the questions or statments is very important. The text cannot be too vague. Also, the text must be clear enough to eliminate misinterpretation. Survey questions must be complete and not include gaps. For example, if an age range is presented, it must include all possible ages. These are just some of the difficulties with getting the text correct in surveys.

Number and Ordering

The number of questions is important. Too many questions and the respondents will lose interest and begin responding without the adequate attention needed. Plus, if the survey is too long there is the risk of quitting before completion. A short survey might not cover the adequate amount of material. Both short and long surveys run the risk of providing inaccurate

responses. After determining the best number of questions, the ordering of the questions is important. Previous survey questions can have an unintended impact on responses. Thus, the ordering of questions needs to be addressed.

Sampling

Next is the issue of sampling. Not every customer can be surveyed, so sample sets os customers need to be presented with a survery. In the case of CRI, there are 2 possible scenarios for sampling.

First When an SDO is part of a larger organization, there typically is a small number of business partners that help to guide and direct the work performed by the SDO. In this case, the business partners might be the the ones offering survey responses and they should all be willing to respond. Thus, those business partners represent the entire population, and the surveys should result in a 100% response rate which is technically a census. The only bias that will be present here is the bias of the business partners and sampling cannot control for that.

Second End-users will be surveyed for satisfaction. Obviously, the entire population cannot be surveyed, so a probability sample should be randomly created. Even then, bias will be present.

- Not all users will respond. This is because survey respondents tend to sit at the extremes of either satisfied or dissatisfied. Thus the results will tend to indicate that separation.
- Even with probability sampling it is possible to miss entire groups of population members. For example consider a banking application such as a savings account, a survey would most likely be presented online and it would have a coverage bias due the exclusion of savings account holders that do not bank online.

- A selection bias can occur when some members of the population have a higher probability of inclusion in the sampling frame than others. One example could be a user with multiple savings accounts. The selection bias is typically easy to avoid if the bias is identified. Weighting is a common solution for selection bias.

For more on creating appropriate surveys, see [75] by Snijkers, Haraldsen, Jones, and Willimack. In the book, Snijkers et al. present a framework named generic statistical business process model (GSBPM) for conducting surveys in a business or organizational setting. GSBPM covers the issues above as well as a few more issues such as response storage and risks. Also, Cowles and Nelson provide another good resource for preparing and conducting surveys [17]. They even include an entire chapters on both writing survey questions and survey errors.

SATISFACTION DATA Once the surveys have been distributed and the results collected, Table 11 displays the data that needs to be collected in order to calculate the satisfaction element score for CRI.

Column Name	**Data Type**	
Question ID	String	Required
Question Text	String	Optional
Respondent ID	String	Optional
Frequency Date	Date	Required
Response	Integer	Required
Response Date	Date	Optional
Application ID	String	Optional

Table 11: SATISFACTION DATA NEEDED FOR CRI

SATISFACTION FORMULA After collecting the necessary survey data from Table 11, calculating the score is rather straight forward. The scores for each question are averaged and then those values are averaged together. If some survey questions are more important than others, the formula could be easily modified to include weighting.

First the score for each question needs to be calculated.

$$S_{3_i} = \sum_{i=1}^{n} \left(k \times \frac{\sum_{j=1}^{m} a_{ij} - \frac{min+max}{2}}{m} \right)$$

- a_{ij} answer to question i for respondent j
- n number of questions
- m number of respondents
- min minimum score for a question
- max maximum score for a question

Then the satisfaction score is calculated as below. Use a weighted average to combine the question scores.

$$S_3 = \sum_{i=1}^{n} w_i' S_{3_i} \text{ where } \sum_{i=1}^{n} w_i' = 1$$

4.1.4 SCHEDULE

Delivery of software in a timely manner is an essential part of being a successful SDO. Being able to meet scheduled deadlines is a sign of accurate estimation and planning. Drastically missing deadlines is a sign of an SDO with a process that needs refinement. Studies have shown that software projects exceed the estimates by an average of 30% [48]. Thus it is important to score SDOs on accurate schedule adherence. Without tracking and measuring schedule adherence, it will not improve.

The CRI schedule score provides a numeric value to indicate the amount schedules are missed or exceeded. The score provides a cumulative measure of the performance as compared to other months. The score is based upon the historical deviance of estimates for projects. Projects completing on time will be given a score of 0. Projects finishing early will be rewarded with positive scores increasing toward k. Alternatively, late projects will be given negative scores that approach $-k$ as the projects become more late.

SCHEDULE DATA In order to calculate the schedule score, certain dates need to be present. Table 12 outlines the necessary data for schedules. One date is considered optional as it is not used in the CRI calculation, but it is an important date that could be useful for future enhancements to CRI.

Column Name	**Data Type**	
Project ID	String	Required
Frequency Date	Date	Required
Scheduled Start Date	Date	Required
Scheduled Finish Date	Date	Required
Actual Start Date	Date	Optional
Actual Finish Date	Date	Required

Table 12: SCHEDULE DATA NEEDED FOR CRI

SCHEDULE FORMULA Schedule has a clear date for finishing on-time, however there are not clear bounds as to how early or late a project can be delivered. Thus, the formula for schedule is more involved than availability or satisfaction. It requires some analysis of the historical data. The first step of the formula is determining how often projects are early or late, and by how much a project is early or late.

This can be accomplished by looking at the distribution of the data. Specifically, look at what percentage of the entire project duration the schedule was missed.

$$\Delta_i = \frac{F_{a_i} - F_{s_i}}{F_{s_i} - B_{s_i} + 1}$$

where

- F_{a_i} the actual finish date of project i
- F_{s_i} the scheduled finish date of project i
- B_{s_i} the scheduled beginning date of project i
- Δ_i the percent the schedule was missed for project i

Once all the Δ_i's have been determined, a distribution must be fit to the data. There are several techniques for testing the fit of a distribution: histograms, chi-square, Kolmogorov-Smirnov, Shapiro-Wilk, or Anderson-Darling [19], [53]. The distribution is needed for the Cumulative Distribution Function (C.D.F.). The C.D.F. maps the values to a percentile rank within the distribution [22]. The C.D.F. will be transformed to create the schedule score for CRI. Since all C.D.F. functions fall within the range $[0, 1]$, the function needs to be shifted to center around 0, and then doubled to fill the desired range of $[-1, 1]$. Thus the CRI schedule score for each project becomes the following.

$$S_{4_i} = 2k \cdot (C.D.F.(\Delta_i) - \frac{1}{2})$$

Then the overall schedule score is calculated as below.

$$S_4 = \sum_{i=1}^{n} w_i' S_{4_i} \text{ where } \sum_{i=1}^{n} w_i' = 1$$

ALTERNATE APPROACH An alternative approach for scoring schedule goes as follows. The best possible score should be achieved when meeting the estimated date exactly. The maximum score should come from the best estimate. Then given historical release data, it is easy to determine an average Δ between the actual and the estimated. Finishing a project within that Δ should result in a positive score. Outside the Δ results in negative scores. For example, a project releasing one day early or one day late would receive the same score because in both cases the estimate was missed by one day.

The first step of the formula is finding the percentage the schedules were missed for historical projects. The calculation treats over- and under-estimating the schedule the same. The same penalty is applied in both cases. For example, being 15% late will result in the same score as being 15% early. Perform this calculation only for projects that did not exactly meet the estimated finish date.

$$\Delta_i = \left| \frac{F_{a_i} - F_{s_i}}{F_{s_i} - B_{s_i} + 1} \right|$$

Find the average of the Δ_i's. This is the average percent of a missed schedule.

$$\bar{\Delta} = \frac{\sum_{i=1}^{n} \Delta_i}{n}$$

The formula for schedule is then a percentage above or below the Δ. The number is calculated for each project, and then averaged to form the schedule score.

After the $\bar{\Delta}$ is calculated, the following formulas are used to create the schedule scores for each project and then the averaged schedule score.

$$S_{4_i} = \left\{ \begin{array}{ll} \text{where } \Delta_i \geq 1 & : -1 \times k \\ \text{where } \Delta_i \leq \bar{\Delta} & : \frac{\bar{\Delta} - \Delta_i}{\bar{\Delta}} \times k \\ \text{where } \Delta_i > \bar{\Delta} & : \frac{\bar{\Delta} - \Delta_i}{1 - \bar{\Delta}} \times k \end{array} \right] \text{, calculate schedule score for each project } i$$

$$S_4 = \sum_{i=1}^{n} w_i' S_{4_i} \text{ where } \sum_{i=1}^{n} w_i' = 1 \text{ , average the schedule scores}$$

where

- n is the number of projects
- F_{a_i} the actual finish date of project i
- F_{s_i} the scheduled finish date of project i
- B_{s_i} the scheduled beginning date of project i
- Δ_i the percent the schedule was missed
- $\bar{\Delta}$ is the average percent schedules are missed
- S_{4_i} is the schedule score for project i

Again, this was just an alternate approach to scoring schedule. It will not be used in the case studies.

4.1.5 REQUIREMENTS

The requirements of a software development organization are important.

Function Points find a definition and a comparison with story points

Story Points Look in some agile book

Column Name	Data Type	
Project ID	String	Required
Frequency Date	Date	Required
Requirements Scheduled	Integer	Required
Actual Requirements Released	Integer	Required

Table 13: REQUIREMENTS DATA NEEDED FOR CRI

REQUIREMENTS DATA

REQUIREMENTS FORMULA Requirements are desired new features or enhancements to a software product. It is important to know how many requirements were scheduled to be completed versus how many actually got completed.

$$S_5 = \frac{\sum_{i=1}^{n} (R_{a_i} - R_{e_i})}{n}$$

- R_a actual requirements completed for application i
- R_e expected requirements completed for application i

4.1.6 OVERALL CRI SCORE

In order to accomplish the single number score that CRI requires, the 5 element scores must be combined. The combination of the scores is a weighted average. The weights can be set based upon the priority of the SDO. Thus, the overall CRI score is calculated as below.

$$CRI = \sum_{i=1}^{n} w_i S_i \text{ where } \sum_{i=1}^{n} w_i = 1$$

4.2 CORRELATIONS IN CRI

It is possible that 2 or more of the 5 elements of CRI will be correlated. This means that one of the elements can be predicted based upon the values of the other elements. Although it is possible for correlation to occur between any of the elements, the satisfaction element is an obvious element which deserves attention due to the human involvement of the surveys. If a schedule is missed or an important requirement dropped, that could have a large negative effect on the satisfaction surveys. The same could be said of quality or availability with regard to the satisfaction. However, satisfaction is not the only potentially correlated element. It is also possible that a decrease in quality could result in unexpected downtime which could have a negative result on availability. Similarly, if requirements are added, it is possible the schedule will be negatively impacted. Also, if requirements are dropped, the quality might suffer due to missing functionality.

It is impossible to know which or if correlations will always exist. Thus it is necessary to check for correlations after determining element and overall CRI scores. If correlation is determined, the element should not be dropped but rather be weighted less than the other elements. This technique keeps the most data available but lessens the importance of the correlated element.

4.3 SENSITIVITY OF CRI

Some simulations will be run with data from each given distribution. Then the CRI scores will be analyzed. For more on sensitivity analysis in statistical modeling, see [74].

4.4 CRI COMPARED

CRI is one way to evaluate an SDO, and it is also a technique of software analytics. Therefore, it is beneficial to compare CRI with some of the other techniques and

guidelines available. The next sections will provide those comparisons.

4.4.1 CRI VS. FOCUS AREAS OF SOFTWARE ANALYTICS

Earlier, in the introduction section 1.4.3, 3 main focus areas for software analytics were presented. Table 14 provides a explanation of how CRI addresses each focus area. As can be seen, CRI clearly addresses the 3 main focus areas of software analytics. CRI does not provide any mechanisms for improving the focus areas, but it provides a consistent mechanism to measure the focus areas.

Focus Area	**Why CRI?**
User Experience	One of the 5 elements of CRI is satisfaction. While not all of the questions focus solely on the user experience, the entire purpose of the survey is to determine if the user is satisfied with the software product. Does it have the correct features? Are new features added in a timely manner? Of course, specific survey questions can be created to focus solely on a certain user experience.
Quality	Again, one of the 5 elements specifically focuses on quality. CRI provides a single number to measure quality. Therefore, it is easy to track changes in quality over time. CRI does not address how to improve the quality, but without a consistent measurement, it would be impossible to determine the change in quality.
Development Productivity	The combination of CRI elements, schedule and requirements, provide an indication of development productivity. The schedule element measures the productivity related to estimated schedule. Similarly, the requirement element measures the amount of work actually being completed.

Focus Area	Why CRI?

Table 14: SOFTWARE ANALYTICS FOCUS AREAS AND CRI

4.4.2 CRI VS. FOCUS AREAS OF SOFTWARE ANALYTICS

Also, section 1.4.3 mentions 3 important questions that software analytics must address. Table 15 presents the 3 questions and a description of how CRI addresses that specific question. It is clear that CRI addresses the questions. CRI is a beneficial technique of software analytics when applied to software development organizations.

Question	Why CRI?
How much better is my model performing than a naive strategy, such as guessing?	CRI provides consistency which may not exist without it. Therefore, CRI removes the guesswork of measuring a software development organization.
How practically significant are the results?	The CRI score is consistent and easy to comprehend. Thus comparison with past performance is quick and simple. This is a significant advantage for software development organizations.
How sensitive are the results to small changes in one or more of the inputs?	The question was extensively addressed in section 4.3. It appears CRI is not overly sensitive to small changes in the inputs.

Table 15: IMPORTANT QUESTIONS FOR SOFTWARE ANALYTICS AND CRI

4.4.3 CRI VS. BALANCED SCORECARD

Section 3.3 discusses the characteristics of the balanced scorecard. Table 16 presents a comparison of the characteristics of a balanced scorecard versus CRI. As the newer 2 characteristics of a balanced scorecard have only existed since 2010 and the adoption is limited, the comparison will only be against the original 4 balanced scorecard characteristics.

Balanced Scorecard	**CRI?**	**Explanation**
Financial	No	CRI does not address financial as it is best suited for an organization that treats software development as a fixed, budgeted expense. If the budget is fixed, CRI provides a number to indicate the amount of value for that fixed budget.
Customer Focus	Yes	CRI includes a customer survey which is completely customer focused.
Internal Process	Yes	CRI is highly focused on internal processes. The CRI elements of schedule and requirement are completely focused on how reality meets the expected process. CRI is negatively impacted when internal processes are followed.
Learning/Growth	No	CRI does not address this characteristic.

Table 16: BALANCED SCORECARD VERSUS CRI

4.4.4 CRI VS. PROJECT MANAGEMENT MEASUREMENT

Previously in Section 3.4, the project management measurement was presented. Its greatest limitation is the lack of focus on the entire SDO. Project management measurement says nothing about the availability of the software infrastructure or the satisfaction of the users. It is not near as comprehensive as either the balanced scorecard or CRI. Actually, CRI incorporates the 5 core measurements from project management. Plus, the process productivity number is less suitable for upper management and more suitable for project teams. Although it does provide a single process productivity number, it is does not have the same focus as CRI.

5 SDLC ANALYTIC ENGINE

In order for an SDO to properly track the elements of CRI, a data storage system should be available to store the appropriate data. A consistent storage system should help to avoid the problem of inaccurate data caused by numerous manipulations of the existing data [64]. Plus, if the system is implemented correctly by allowing limited changes to existing data, it will be able to alleviate some of the dishonesty that is currently present in software projects [69]. This storage system will be named the SDLC Analytic Engine (SDLC-AE).

Once all the SDLC data is collected into a single place, there are many possible applications. Software analytics will be much easier to create and gamification will be much more easily attainable. CRI is just one possible application of the SDLC-AE. Figure 6 provides an overview of the data that could potentially be stored in the SDLC-AE as it relates to CRI. The SDLC-AE does not specify how the data is entered, just how the data is stored.

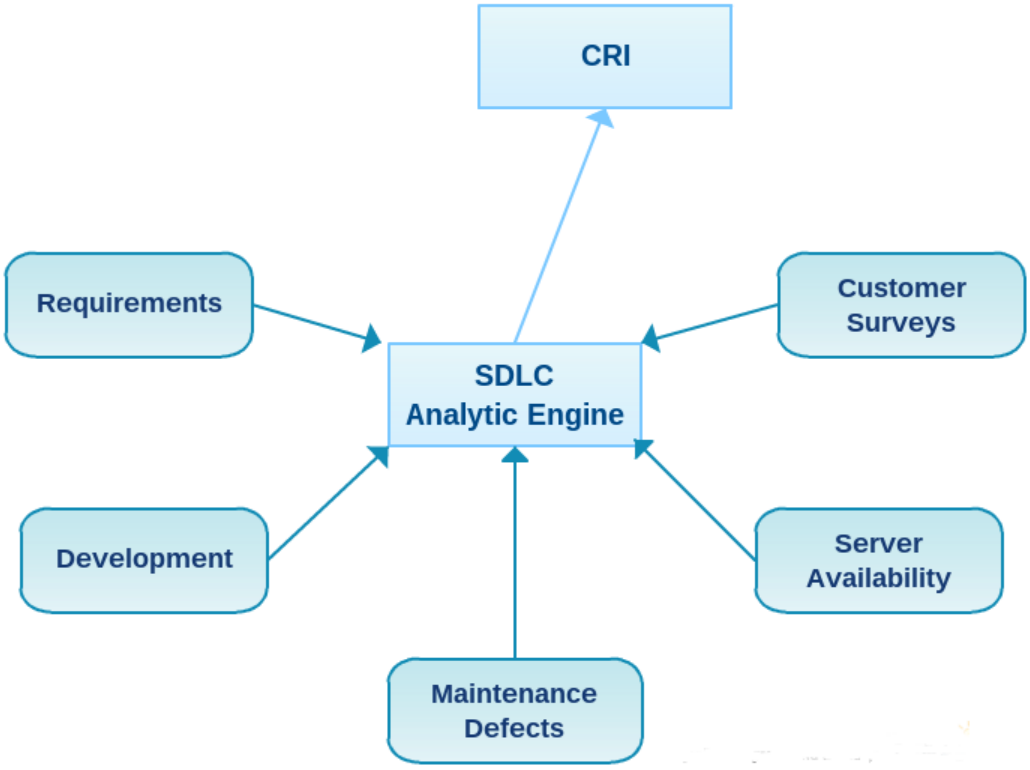

Figure 6: SDLC ANALYTIC ENGINE

5.1 DATABASE STRUCTURE

All the data necessary to compute CRI needs to be stored in a database. This section will lay out the structure of tables and relationships necessary to the store all the data within a relational database. The SQL[10] is written for an Oracle database, but the scripts can be modified to work with another database such as: PostgreSQL, SQL Server, or MySQL.

5.1.1 TABLES FOR RAW CRI DATA

The first set of tables that need to be created are tables to store the raw data that is collected. These tables will line up with the necessary data for each of the 5 elements of CRI. Figure 7 provides a visual description of the tables that are

[10]Structured Query Language (SQL) is a programming language designed for managing data in relational database management system.

needed. The tables have no relationship with each other since they are raw data. The primary goal of these tables is to store the raw data in a single database. The 5 table names are:

1. QUALITY_RAW
2. AVAILABILITY_RAW
3. SATISFACTION_RAW
4. SCHEDULE_RAW
5. REQUIREMENTS_RAW

Notice these table names match with the 5 elements of CRI.

Appendix B.1 provides the necessary SQL statements to create the database tables for storing the raw CRI data.

5.1.2 INTERMEDIATE SCORE TABLES FOR CRI

The next set of tables that need to be created are for the intermediate level element scores. These tables hold the element scores at the application_id, service_id, and project_id level. Figure 8 provides a visual overview of the necessary tables. Furthermore, these tables also lack relationships between one another because at this point, all the element scores are still being treated independently. The five table names are:

1. QUALITY_SCORE
2. AVAILABILITY_SCORE
3. SATISFACTION_SCORE
4. SCHEDULE_SCORE

Figure 7: TABLES FOR RAW CRI DATA

5. REQUIREMENTS_SCORE

Again, these table names match very closely with the 5 elements of CRI.

Appendix B.2 provides the necessary SQL statements to create the database tables for storing the intermediate CRI scores.

5.1.3 FINAL SCORE TABLES FOR CRI

The final set of tables consists of only two tables.

1. ELEMENT

Figure 8: TABLES FOR INTERMEDIATE CRI SCORES

2. CRI_SCORE

The first table is the ELEMENT table. It simply stores the CRI element (Quality, Availability,Satisfaction,Schedule,Requirements,Overall) and optional description. The second table is the CRI_SCORE table. It stores all the final element scores and the final overall CRI score. It is related to the ELEMENT table. Figure 9 provides a visual representation of the relationship between the 2 tables.

Appendix B.3 provides the necessary SQL statements to create the database tables for storing the final scores for each element and the final overall CRI scores for each time frequency.

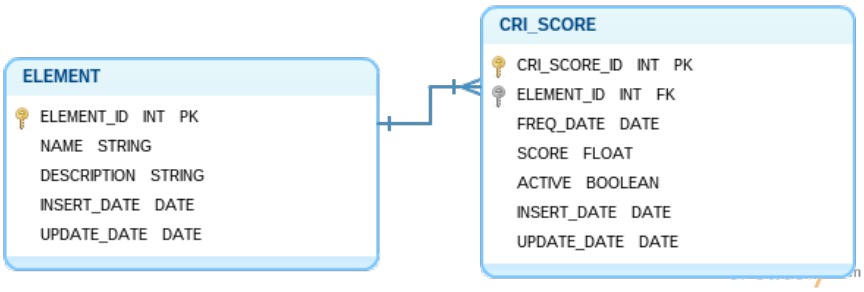

Figure 9: TABLES FOR FINAL CRI SCORES

6 CASE STUDY: SCORING AN SDO OF A LARGE FINANCIAL INSTITUTION

Data has been collected from the software development processes of an SDO within a large financial institution[11]. The data collection was from 2007 to January 2015. Only 4 elements had available data, and not all elements had data available for the entire time period. The elements to be used are: quality, availability, schedule, and requirements. CRI is still effective when not all data is available. The overall CRI score will then be a weighted average of the available elements. This section will serve as a guide to preparing the CRI score based upon the available data.

6.1 WITH HISTORICAL DATA

After the data has been collected, the raw data can be stored in the SDLC if desired. Then scores for each of the 4 elements can be calculated. For this example, the value of k will be 100 and the time frequency will be monthly. Thus, scores will fall in the range $[-100, 100]$. Also, equal weighting is applied to all elements and all Project IDs, Application IDs, and System IDs.

[11] All the raw data files are available at [79]

6.1.1 QUALITY

The first step to dealing with the quality data is a quick analysis of the data. Table 17 provides some descriptive statistics for the quality data. Testing hours were not capture but they are optional, so the analysis can continue.

Column	Min	Max	Median	Mean	Variance
Development Effort	0	26937	300	1637	18383464
Testing Effort	NA	NA	NA	NA	NA
SIT Defects	0	1106	1	45.86	24528
UAT Defects	0	277	0	10.28	1306
PROD Defects	0	1216	5	51.5	20311

985 obs. from 23 Application IDs from 2007 to 2015

Table 17: QUALITY DATA DESCRIPTIVE STATISTICS

Notice, the data for quality goes from October 2007 to January 2015 and contains 985 observations. This is important because historical data can be used to create the baseline quality function. All quality for the years 2007 through the end of 2013 will be used as historical data for the purposes of creating the baseline quality function. Once the historical data is separated, it results in 799 observations to be used for creating the baseline quality function. Figure 10 shows scatterplots of PROD_DFTS versus the independent variables of DEV_EFF, SIT_DFTS, and UAT_DFTS. It can be seen that some correlations exist between the variables.

Figure 10 also shows the presence of a possible outlier with 1216 PROD_DFTS. That data point is dropped for the remaining analysis. At this point, a simple linear regression model can be fit to the remaining 798 observations. At this point, no transformations have been performed on the data. The linear regression model yields all 3 independent variables as significant and an overall

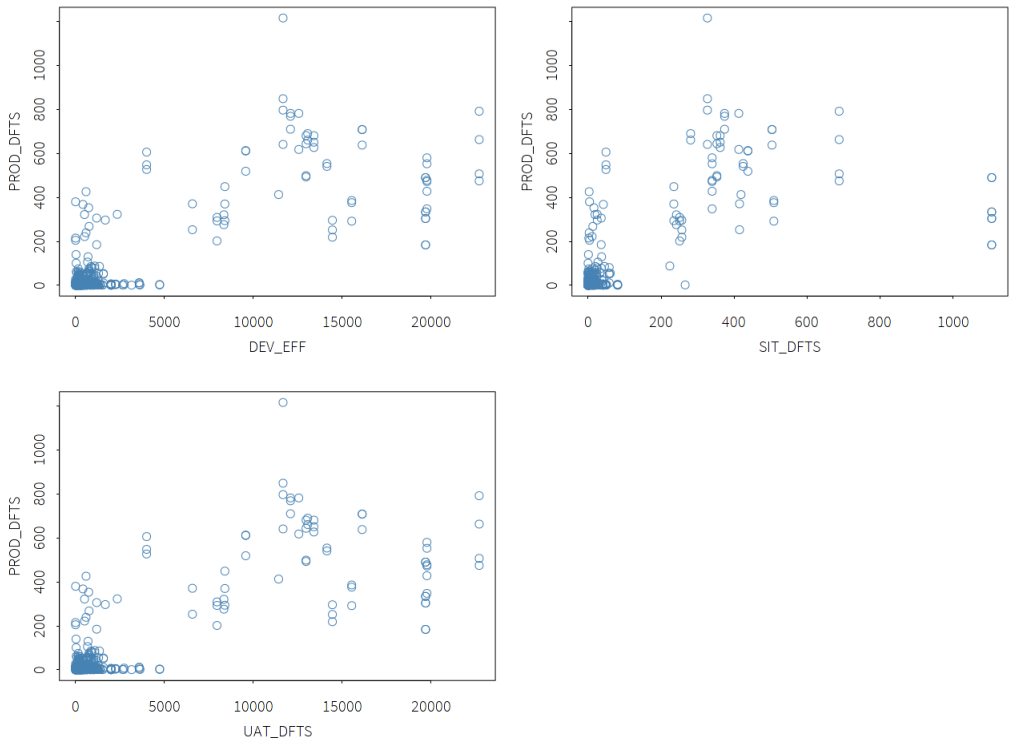

Figure 10: QUALITY DATA PLOTS: DEPENDENT VS. INDEPENDENT VARIABLES

$R^2 = .72$. The source code and some further analysis can be found in Appendix C.1. It appears to be a good fit and thus all Application IDs will have the same baseline quality function. Therefore, f does not need a subscript, and f can be seen below.

$$f = 5.92 + 0.035 \cdot DEV\_EFF - 0.36 \cdot SIT\_DFTS + 1.05 \cdot UAT\_DFTS$$

Now that f has been determined, the quality scores for each application can be calculated for all the months beyond 2013. Appendix C.3 provides the necessary R code to perform the calculations. Figure 11 shows the quality scores for 2014 and beyond. As can be seen, the scores are greatly above expectations. That is an indication of improved quality over historical performance.

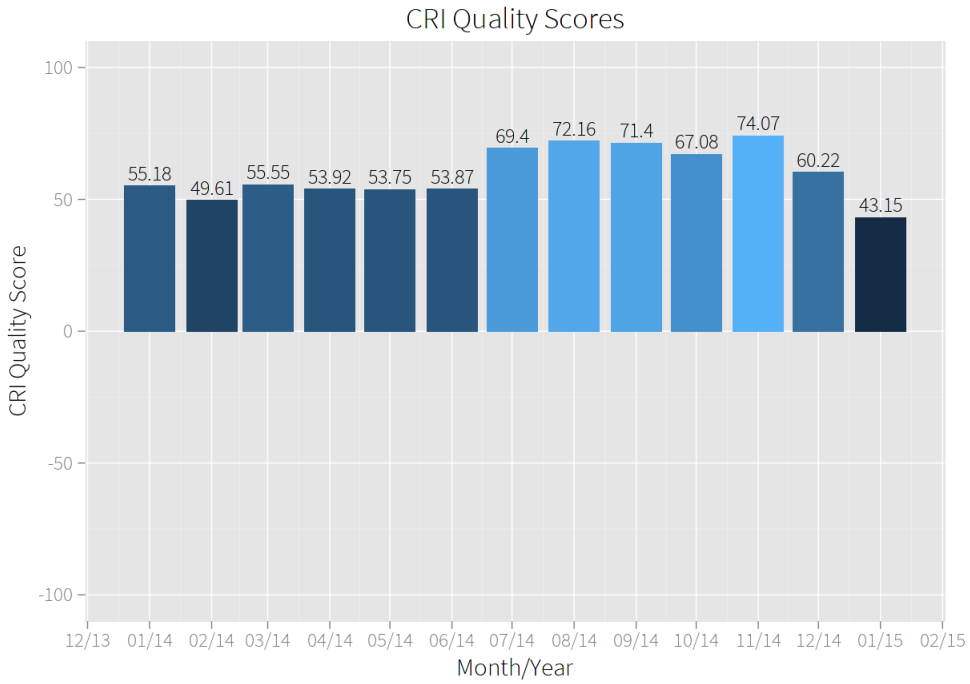

Figure 11: CRI QUALITY SCORES

6.1.2 AVAILABILITY

For the availability data, some descriptive statistics can be seen in Table 18. The percent uptime was precalculated for this data so uptime, scheduled downtime, and unscheduled downtime are not needed. Availability does not rely upon analysis of historical data since a known upper and lower bound exist, 1 and 0 respectively.

Column	**Min**	**Max**	**Median**	**Mean**	**Variance**
Percent Uptime	0	1.0	1.0	0.9745	0.023
Expected Percent Uptime	0.93	1.0	0.98	0.9769	0.977

7522 obs. from 83 Application IDs from 2008 to 2015

Table 18: AVAILABILITY DATA DESCRIPTIVE STATISTICS

Since percent uptime has already been calculated and no historical analysis needs to be performed, the calculation of the scores can be performed. R code to compute the CRI availability scores can be found in Appendix C.4. Figure 12 displays the CRI availability scores. As can be seen the scores are all above 70.

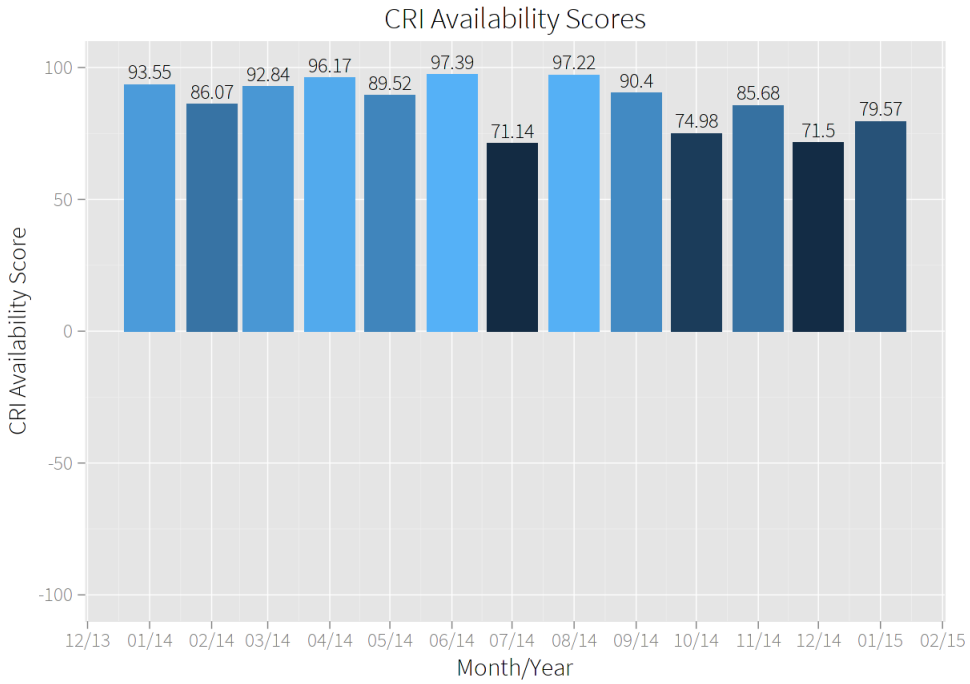

Figure 12: CRI AVAILABILITY SCORES

6.1.3 SCHEDULE

6.1.4 REQUIREMENTS

6.1.5 OVERALL

All of the months do not include the same elements.

An CRI calculation analyzing historical performance.

7 FUTURE WORK

Due to the number of issues with surveys. One area of future work would be indentifying a general set of questions that would best fit CRI. This set would have to include the best number of questions, order of questions and wording of questions. Therefore, any new organization would not have to determine their own survey, but rather just use the predetermined set of questions. It would even be

advantageous to build a software system to handle the survey distribution and collection. Ideally, the results would automatically be inserted into the appropriate database tables for CRI scoring.

As shown in Table 16, CRI is currently not addressing 2 characteristics of the balanced scorecard. If the SDO does not operate on a fixed budget, CRI could be expanded to include another element for financial data. The challenge arises when determining how to relate the SDO performance to finances. The most likely scenario would be a method to either select or predicted the expected software sales for a month. Then every month create an CRI element score that reflects how much the expected sales were exceeded or missed. The other missing balanced scorecard characteristic is learning and growth. Those are very difficult to quantitatively measure. In this case some possible data points might be: hours of training, number of training courses, number of employees receiving training, number of promotions, or another measure centered around training courses and career growth. Again, once data exists, the problem becomes finding a baseline and measuring with respect to that baseline.

Another area of future work is the expansion of the SDLC-AE to include more artifacts of the SDLC. The more artifacts and processes that can be collected, the deeper the understanding of the SDLC. All of the data collection combined with better software analytics could lead to true *data-driven software engineering*. Data-driven software engineering is the application of collecting and analyzing historical information about software engineering artifacts in order to accurately predict the outcomes of software engineering projects. This will lead to more informed decisions about software engineering. Figure 13 shows an expansion of the previous SDLC-AE diagram. The new elements are in the unshaded boxes, and they are not exhaustive. Data-driven software engineering should not to be confused with data-driven programming, in which the computer code describes the data

instead of the sequence of opertions [20].

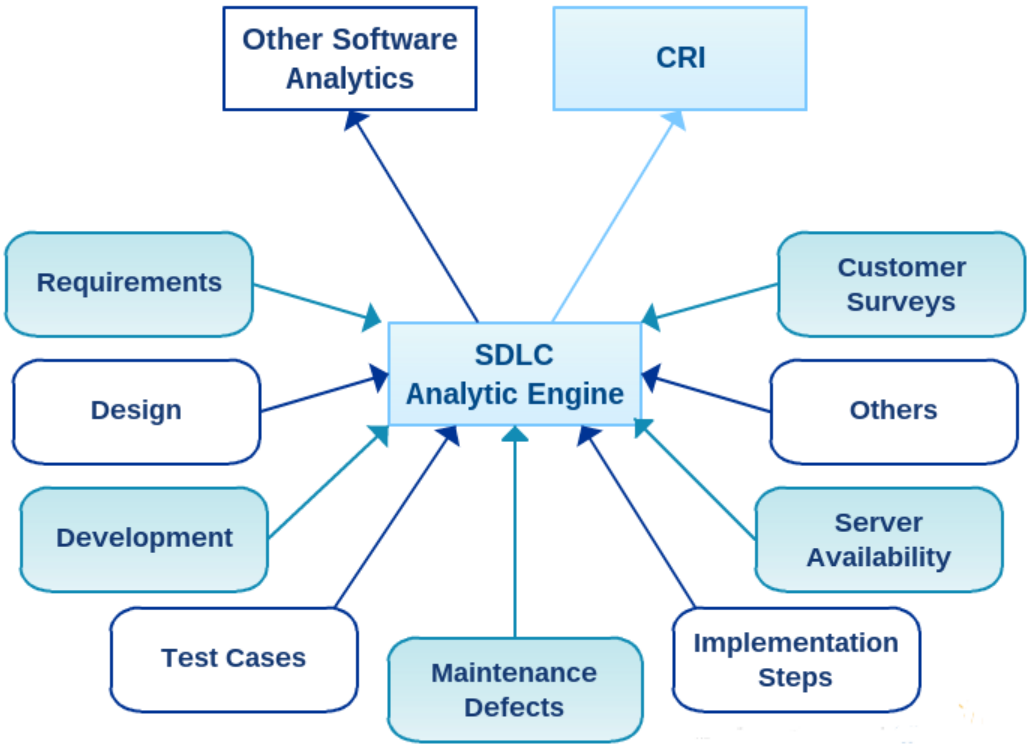

Figure 13: SDLC ANALYTIC ENGINE EXPANSION

Currently, CRI provides a single method to evaluate past performance, but it does not provide any guidance around making more informed future decisions. Some phases are not properly tracked with the initial SDLC-AE and more data can be tracked for the existing phases. Schedules for each of the individual phases of the SDLC need to be tracked, not just the entire project. Teams need to know how much time is spent in design versus testing. Also, how much time is required to generate proper test cases? These are just a couple of examples of expansions to the SDLC-AE. These and other advancements could lead to greater insights about SDLC phases that are struggling and need improvement. The SDLC-AE could be expanded to have predictive capabilities.

8 CONCLUSION

There are many metrics that can be used to evaluate a Software Development Organization (SDO). Knowing which metrics to use and what they all mean can be a daunting task. CRI is a proposed solution to the difficulty of measuring an SDO.

Upon completion, this work shall identify:

- Define **What** characteristics should be measured for an SDO
- Define **How** to measure those characteristics
- Map the relationship or lack of relationship with a Balanced Scorecard
- Create a Framework to store the necessary data for CRI
- Outline a Process to generate the CRI score
- Provide an example CRI score with real data

An entire software development organization (SDO) needs to be measured and analyzed properly, not just the development portion. This document has provided an overview of what indicators need to be measured for an SDO, and how those indicators can be combined to form a single number indicating the performance score of a software development organization. Also, a framework to store this data was discussed.

APPENDIX

A DETAILED STEPS OF THE SDLC

Often times the SDLC contains more than the 5 basic steps of requirements, design, implementation, testing, and deployment/maintenance. Those are the high-level phases, but many steps are required to complete each phase. The following list provides a more detailed list of what needs to be accomplished in the entire life cycle of software development. These steps do not need to occur in a sequential fashion.

- Identify the Work/Task/Project
  - Get Initial Idea
  - Obtain Details
- Estimate
  - Create an Estimate (What is included? What is the output? days/dollars/hours/reqs)
  - Obtain Approval
  - Quit or Go Forward
- Document Requirements
  - Identify the Requirements
  - Detail the Requirements
- Design The Software
  - Find System Integrations
  - Identify Functional Specs
  - Detail the Functional Specs
- Development of all the tasks in Design and Requirements

  - Identify the Coding Tasks
  - Write the Code/Develop the solution
  - Write the Unit Tests
- Test
  - Create Test Plans and Cases
  - Run Test Plans and Cases
- Deployment
  - Create Deployment Steps
  - Run Deployment Steps
- Maintenance
  - Capture Bugs
  - Survey Users
- Start Again

B SDLC-AE SOURCE CODE

B.1 SQL CODE - DATA TABLES

```
-- create the raw data tables

CREATE TABLE QUALITY_RAW (
    QUALITY_RAW_ID RAW(16) NOT NULL PRIMARY KEY,
    APPLICATION_ID VARCHAR2(64) NOT NULL,
    FREQ_DATE DATE NOT NULL,
    DEV_EFFORT NUMBER(10,0) NOT NULL,
    TEST_EFFORT NUMBER(10,0),
    SIT_DEFECTS NUMBER(10,0),
    UAT_DEFECTS NUMBER(10,0),
    PROD_DEFECTS NUMBER(10,0) NOT NULL,
    INSERT_DATE DATE DEFAULT SYSDATE NOT NULL,
    UPDATE_DATE DATE DEFAULT SYSDATE NOT NULL
);

CREATE TABLE AVAILABILITY_RAW (
```

```
17      AVAILABILITY_RAW_ID RAW(16) NOT NULL PRIMARY KEY,
18      SERVICE_ID VARCHAR2(64) NOT NULL,
19      FREQ_DATE DATE NOT NULL,
20      UPTIME NUMBER(10,5),
21      SCHED_DOWNTIME NUMBER(10,5),
22      UNSCHED_DOWNTIME NUMBER(10,5),
23      PERCENT_UPTIME NUMBER(10,5) NOT NULL,
24      EXPECT_PERCENT_UPTIME NUMBER(10,5) NOT NULL,
25      INSERT_DATE DATE DEFAULT SYSDATE NOT NULL,
26      UPDATE_DATE DATE DEFAULT SYSDATE NOT NULL
27  );
28
29  CREATE TABLE SATISFACTION_RAW (
30      SATISFACTION_RAW_ID RAW(16) NOT NULL PRIMARY KEY,
31      QUESTION_ID VARCHAR2(64) NOT NULL,
32      FREQ_DATE DATE NOT NULL,
33      QUESTION_TEXT VARCHAR2(1024),
34      RESPONDENT_ID VARCHAR2(128),
35      RESPONSE NUMBER(5,0) NOT NULL,
36      RESPONSE_DATE DATE,
37      APPLICATION_ID VARCHAR2(64),
38      INSERT_DATE DATE DEFAULT SYSDATE NOT NULL,
39      UPDATE_DATE DATE DEFAULT SYSDATE NOT NULL
40  );
41
42  CREATE TABLE SCHEDULE_RAW (
43      SCHEDULE_RAW_ID RAW(16) NOT NULL PRIMARY KEY,
44      PROJECT_ID VARCHAR2(64),
45      FREQ_DATE DATE NOT NULL,
46      SCHED_START_DATE DATE NOT NULL,
47      SCHED_FINISH_DATE DATE NOT NULL,
48      ACTUAL_START_DATE DATE,
49      ACTUAL_FINISH_DATE DATE NOT NULL,
50      INSERT_DATE DATE DEFAULT SYSDATE NOT NULL,
51      UPDATE_DATE DATE DEFAULT SYSDATE NOT NULL
52  );
53
54  CREATE TABLE REQUIREMENTS_RAW (
55      REQUIREMENTS_RAW_ID RAW(16) NOT NULL PRIMARY KEY,
56      PROJECT_ID VARCHAR2(64),
57      FREQ_DATE DATE NOT NULL,
58      SCHED_REQ NUMBER(10,0) NOT NULL,
59      ACTUAL_REQ NUMBER(10,0) NOT NULL,
60      INSERT_DATE DATE DEFAULT SYSDATE NOT NULL,
61      UPDATE_DATE DATE DEFAULT SYSDATE NOT NULL
62  );
63
```

```
-- remove the raw tables
--DROP TABLE QUALITY_RAW;
--DROP TABLE AVAILABILITY_RAW;
--DROP TABLE SATISFACTION_RAW;
--DROP TABLE SCHEDULE_RAW;
--DROP TABLE REQUIREMENTS_RAW;
```

B.2 SQL CODE - SCORE TABLES

```
-- Create the scoring tables

CREATE TABLE QUALITY_SCORE (
     QUALITY_SCORE_ID RAW(16) NOT NULL PRIMARY KEY,
     APPLICATION_ID VARCHAR2(64) NOT NULL,
     FREQ_DATE DATE NOT NULL,
     SCORE NUMBER(10,5) NOT NULL,
     ACTIVE CHAR DEFAULT 'Y' NOT NULL,
     INSERT_DATE DATE DEFAULT SYSDATE NOT NULL,
     UPDATE_DATE DATE DEFAULT SYSDATE NOT NULL
);

CREATE TABLE AVAILABILITY_SCORE (
     AVAILABILITY_SCORE_ID RAW(16) NOT NULL PRIMARY KEY,
     SERVICE_ID VARCHAR2(64) NOT NULL,
     FREQ_DATE DATE NOT NULL,
     SCORE NUMBER(10,5) NOT NULL,
     ACTIVE CHAR DEFAULT 'Y' NOT NULL,
     INSERT_DATE DATE DEFAULT SYSDATE NOT NULL,
     UPDATE_DATE DATE DEFAULT SYSDATE NOT NULL
);

CREATE TABLE SATISFACTION_SCORE (
     AVAILABILITY_SCORE_ID RAW(16) NOT NULL PRIMARY KEY,
     QUESTION_ID VARCHAR2(64) NOT NULL,
     FREQ_DATE DATE NOT NULL,
     SCORE NUMBER(10,5) NOT NULL,
     ACTIVE CHAR DEFAULT 'Y' NOT NULL,
     INSERT_DATE DATE DEFAULT SYSDATE NOT NULL,
     UPDATE_DATE DATE DEFAULT SYSDATE NOT NULL
);

CREATE TABLE SCHEDULE_SCORE (
     QUALITY_SCORE_ID RAW(16) NOT NULL PRIMARY KEY,
     PROJECT_ID VARCHAR2(64) NOT NULL,
     FREQ_DATE DATE NOT NULL,
     SCORE NUMBER(10,5) NOT NULL,
     ACTIVE CHAR DEFAULT 'Y' NOT NULL,
```

```
39      INSERT_DATE DATE DEFAULT SYSDATE NOT NULL,
40      UPDATE_DATE DATE DEFAULT SYSDATE NOT NULL
41  );
42
43  CREATE TABLE REQUIREMENTS_SCORE (
44      REQUIREMENTS_SCORE_ID RAW(16) NOT NULL PRIMARY KEY,
45      PROJECT_ID VARCHAR2(64) NOT NULL,
46      FREQ_DATE DATE NOT NULL,
47      SCORE NUMBER(10,5) NOT NULL,
48      ACTIVE CHAR DEFAULT 'Y' NOT NULL,
49      INSERT_DATE DATE DEFAULT SYSDATE NOT NULL,
50      UPDATE_DATE DATE DEFAULT SYSDATE NOT NULL
51  );
52
53  -- remove the scoring tables
54  --DROP TABLE QUALITY_SCORE;
55  --DROP TABLE AVAILABILITY_SCORE;
56  --DROP TABLE SATISFACTION_SCORE;
57  --DROP TABLE SCHEDULE_SCORE;
58  --DROP TABLE REQUIREMENTS_SCORE;
```

B.3 SQL CODE - FINAL SCORE TABLES

```
1   -- create the ELEMENT table
2
3   CREATE TABLE ELEMENT (
4       ELEMENT_ID NUMBER(10,0) NOT NULL PRIMARY KEY,
5       NAME VARCHAR2(64) NOT NULL,
6       DESCRIPTION VARCHAR(255),
7       INSERT_DATE DATE DEFAULT SYSDATE NOT NULL,
8       UPDATE_DATE DATE DEFAULT SYSDATE NOT NULL
9   );
10
11  -- Add the CRI score types to the table
12  INSERT INTO ELEMENT (ELEMENT_ID,NAME)
13      VALUES (1,'QUALITY');
14  INSERT INTO ELEMENT (ELEMENT_ID,NAME)
15      VALUES (2,'AVAILABILITY');
16  INSERT INTO ELEMENT (ELEMENT_ID,NAME)
17      VALUES (3,'SATISFACTION');
18  INSERT INTO ELEMENT (ELEMENT_ID,NAME)
19      VALUES (4,'SCHEDULE');
20  INSERT INTO ELEMENT (ELEMENT_ID,NAME)
21      VALUES (5,'REQUIREMENTS');
22  INSERT INTO ELEMENT (ELEMENT_ID,NAME)
23      VALUES (6,'OVERALL');
24
```

```
25
26 -- create the overall score table
27 CREATE TABLE CRI_SCORE (
28      CRI_SCORE_ID RAW(16) NOT NULL PRIMARY KEY,
29      ELEMENT_ID NUMBER(10,0) NOT NULL FOREIGN KEY
30           REFERENCES ELEMENT(ELEMENT_Id),
31      FREQ_DATE DATE NOT NULL,
32      SCORE NUMBER(10,5) NOT NULL,
33      ACTIVE CHAR DEFAULT 'Y' NOT NULL,
34      INSERT_DATE DATE DEFAULT SYSDATE NOT NULL,
35      UPDATE_DATE DATE DEFAULT SYSDATE NOT NULL
36 );
37
38 -- remove ELEMENT table
39 --DROP TABLE ELEMENT;
40 --DROP TABLE CRI_SCORE;
```

C CASE STUDY SOURCE CODE

C.1 QUALITY HISTORICAL R CODE AND ANALYSIS

```
1  ### Load the necessary libraries
2  library('sense')
3  library('ggplot2')
4  source('barchart.R')
5
6  ### Load Raw Quality Data
7  setAs("character","myDate",
8        function(from) {as.Date(from, format="%m/%d/%Y")} )
9  setClass('myDate')
10
11 quality_raw <- read.csv('data/quality_raw.csv',
12                          colClasses=c('factor',
13                                       'myDate',
14                                       'numeric',
15                                       'numeric',
16                                       'numeric',
17                                       'numeric') )
18
19 ### Get descriptive statistics
20 str(quality_raw)
21 summary(quality_raw)
22 var(quality_raw)
23
24 ### Find the Baseline Quality Function
25 #### Use data prior to 2014
26 history_quality_raw =
```

```
27          quality_raw[quality_raw$MONTH_DT
28                      <= as.Date('2013-12-31')  ,]
29
30 #### Create some plots of the historical quality data
31 par(mfrow=c(2,2))
32 plot(history_quality_raw$DEV_EFF,
33      history_quality_raw$PROD_DFTS,
34       xlab='DEV_EFF', ylab='PROD_DFTS', col='steelblue')
35 plot(history_quality_raw$SIT_DFTS,
36      history_quality_raw$PROD_DFTS,
37       xlab='SIT_DFTS', ylab='PROD_DFTS', col='steelblue')
38 plot(history_quality_raw$DEV_EFF,
39      history_quality_raw$PROD_DFTS,
40       xlab='UAT_DFTS', ylab='PROD_DFTS', col='steelblue')
41
42 ##### Remove the outlier data point with 1216 PROD_DFTS
43 history_quality_clean = history_quality_raw[history_quality_raw$
   PROD_DFTS < 1000,]
44
45 ##### create the model after dropping the
46 ##### data point with over 1000 PROD_DFTS
47 baseline_quality_function = lm(PROD_DFTS ~ DEV_EFF + SIT_DFTS +
   UAT_DFTS,
48             data=history_quality_clean )
49 summary(baseline_quality_function)
50
51 par(mfrow=c(1,2))
52 qqnorm(baseline_quality_function$resid,col='steelblue')
53 qqline(baseline_quality_function$resid,col='steelblue')
54 summary(baseline_quality_function)$sigma
55 plot(baseline_quality_function$fitted,abs(baseline_quality_
   function$resid),
56       col='steelblue',
57      main='Fitted vs Resid',
58      xlab='Fitted',
59      ylab='Absolute Value of Residuals')
60 pairs(history_quality_clean[  ,c('DEV_EFF','SIT_DFTS','UAT_DFTS')
   ],col='steelblue')
61
62 #### source code for ridge regression
63 #### did not yield better results
64 install.packages('ridge')
65 library('ridge')
66 rd = linearRidge(PROD_DFTS ~ DEV_EFF
67             + SIT_DFTS + UAT_DFTS  ,
68             data=history_quality_clean, nPCs=1)
69 summary(rd)
```

The source code for the selected baseline quality function can be seen above. The output for the linear model can be seen below.

```
Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)  5.917433   2.959339   2.000   0.0459 *
DEV_EFF      0.034942   0.001773  19.709  < 2e-16 ***
SIT_DFTS    -0.362998   0.048068  -7.552 1.18e-13 ***
UAT_DFTS     1.048225   0.143276   7.316 6.25e-13 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 77.43 on 794 degrees of freedom
Multiple R-squared:  0.7188,    Adjusted R-squared:  0.7178
F-statistic: 676.6 on 3 and 794 DF,  p-value: < 2.2e-16
```

Some diagnostic plots for the baseline quality function can be seen in Figure 14. The normal probability plot, a.k.a. QQ Plot, shows the errors are not exactly normally distributed, but the baseline quality function had good predictive power as shown by the high R^2. The fitted versus residuals plot indicates a lack of heteroscedasticity.

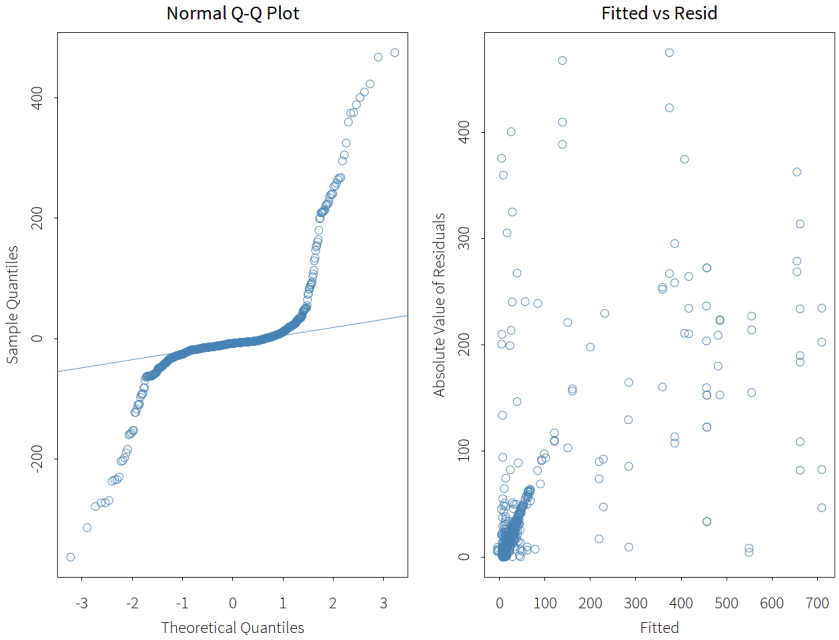

Figure 14: QUALITY DIAGNOSTIC PLOTS

Figure 15 shows the presence of some possible multicolinearity. As a result, ridge regression was used to create a model, but the results were very similar to the original baseline quality function. Therefore, ridge regression was not chosen.

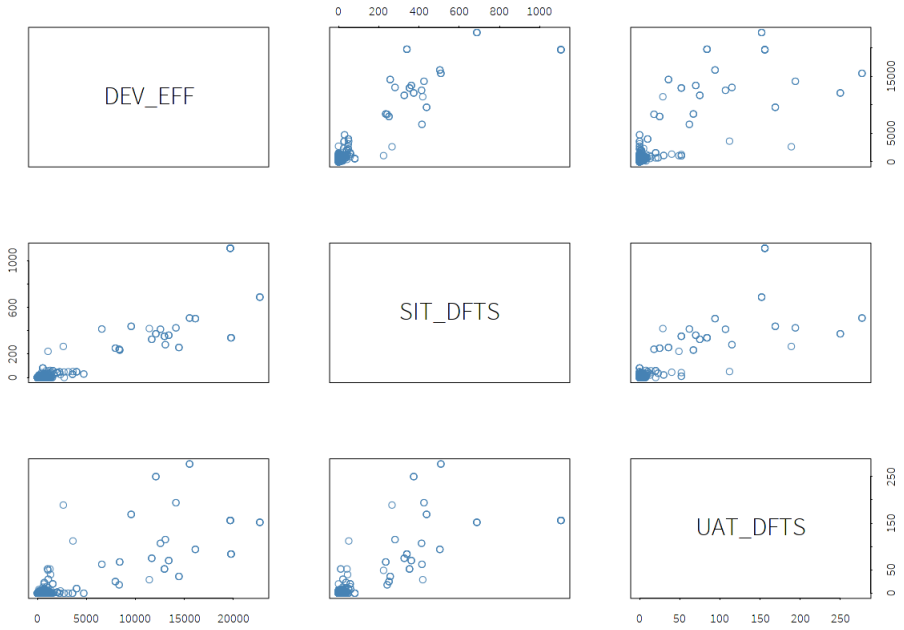

Figure 15: QUALITY PAIRS PLOT OF INDEPENDENT VARIABLES

C.2 BAR CHART - R CODE

This is a code for generating a bar chart. It is used for reporting the scores of all the elements.

```
library('ggplot2')
install.packages('zoo')
library('zoo')
library('scales')

barchart <- function(data, main='CRI Scores', xlab='Month', ylab=
    'MPI Score') {

    ggplot(data, aes(x=as.Date(MONTH_DT), y=SCORE, fill=SCORE)) +
      geom_bar(stat = "identity") +
      geom_text(aes(label=round(SCORE, 2), vjust=ifelse(sign(
    SCORE)>=0, -.4, 1.4)), size=9 ) +
```

```
        guides(fill=FALSE) +
        ylim(-100,100) +
        scale_x_date(labels = date_format("%m/%y"), breaks = date_
    breaks("month")) +
        ggtitle(main) +
        ylab(ylab) +
        xlab(xlab)
}
```

C.3 QUALITY SCORES - R CODE

```
### Load the necessary libraries
library('sense')
library('ggplot2')

### Load Raw Quality Data
setAs("character","myDate",
        function(from) {as.Date(from, format="%m/%d/%Y")} )
setClass('myDate')

quality_raw <- read.csv('data/quality_raw.csv',
                        colClasses=c('factor',
                                     'myDate',
                                     'numeric',
                                     'numeric',
                                     'numeric',
                                     'numeric') )

### Get descriptive statistics
str(quality_raw)
summary(quality_raw)
var(quality_raw)

### Find the Baseline Quality Function
#### Use data prior to 2014
history_quality_raw =
        quality_raw[quality_raw$MONTH_DT
                    <= as.Date('2013-12-31') ,]

#### Create some plots of the historical quality data
par(mfrow=c(2,2))
plot(history_quality_raw$DEV_EFF,
     history_quality_raw$PROD_DFTS,
      xlab='DEV_EFF', ylab='PROD_DFTS', col='steelblue')
plot(history_quality_raw$SIT_DFTS,
     history_quality_raw$PROD_DFTS,
      xlab='SIT_DFTS', ylab='PROD_DFTS', col='steelblue')
```

```
plot(history_quality_raw$DEV_EFF,
     history_quality_raw$PROD_DFTS,
      xlab='UAT_DFTS', ylab='PROD_DFTS', col='steelblue')


##### Remove the outlier data point with 1216 PROD_DFTS
history_quality_clean = history_quality_raw[history_quality_raw$
   PROD_DFTS < 1000,]

##### create the model after dropping the
##### data point with over 1000 PROD_DFTS
baseline_quality_function = lm(PROD_DFTS ~ DEV_EFF + SIT_DFTS +
   UAT_DFTS,
            data=history_quality_clean )
summary(baseline_quality_function)

par(mfrow=c(1,2))
qqnorm(baseline_quality_function$resid,col='steelblue')
qqline(baseline_quality_function$resid,col='steelblue')
summary(baseline_quality_function)$sigma
plot(baseline_quality_function$fitted,abs(baseline_quality_
   function$resid),
      col='steelblue',
     main='Fitted vs Resid',
     xlab='Fitted',
     ylab='Absolute Value of Residuals')
pairs(history_quality_clean[ ,c('DEV_EFF','SIT_DFTS','UAT_DFTS')
   ],col='steelblue')

#### source code for ridge regression
#### did not yield better results
install.packages('ridge')
library('ridge')
rd = linearRidge(PROD_DFTS ~ DEV_EFF
            + SIT_DFTS + UAT_DFTS ,
            data=history_quality_clean, nPCs=1)
summary(rd)
```

C.4 AVAILABILITY SCORES - R CODE

```
library('sense')
source('barchart.R')

# Load Data
# ---------
#
# Load the Availability data.
```

```
setAs("character","myDate",
      function(from) {as.Date(from, format="%m/%d/%Y")} )
setClass('myDate')

avail_raw <- read.csv('data/availability_raw.csv',
                      colClasses=c('factor',
                                   'myDate',
                                   'numeric',
                                   'numeric'))

summary(avail_raw)
str(avail_raw)

# trim to only 2014 and newer
avail_data =
        avail_data[avail_data$SLA_DATE >= as.Date('2014-01-01')
    ,]

avail_data$SCORE =  ifelse(
  avail_data$ACTUAL <= avail_data$EXPECTED,
  (avail_data$ACTUAL - avail_data$EXPECTED)/avail_data$EXPECTED,
  (avail_data$ACTUAL - avail_data$EXPECTED)/(1 - avail_data$
   EXPECTED)
  )

avail_scores = aggregate(SCORE ~ SLA_DATE, avail_data, mean )
avail_scores$SLA_DATE = as.yearmon(avail_scores$SLA_DATE, "%Y-%B")
    )
avail_scores$SCORE = round(100* avail_scores$SCORE, 2)

barchart(avail_scores, main='CRI Availability Scores',
         ylab='CRI Quality Score',xlab='Month/Year')
```

D ADDITIONAL SDLC DATA NEEDS

D.1 ESTIMATION

- Change to Database Structure
- Modify Database Data
- Create a New Database, number of new databases
- Server Configuration Changes Required

- New Servers Required
- Number of Team Members Involved
- Number of (sub)Systems Involved
- Estimation Date
- Number of Days Allowed
- List of Other Attributes
- Number of Screens Involved
- Actual Value (hours, days, dollars)

Estimated Dev Hours - The number of development hours estimated for a project, this is just developer hours

Estimated Doc Hours - The number of documentation hours estimated for a project

Estimated Test Hours - The number of testing hours estimated for a project

Estimated Deployment Hours - The number of estimated hours required to deploy the project

D.2 REQUIREMENTS

List of:

Title

Descrition

author

projectID

date

Comments list of

date

comment Text

author

D.3 MAINTENANCE (DEFECTS)

List of:

Project

Release

description

date Entered

Date fixed

Comments list of

date

comment Text

author

D.4 TESTING

List of:

Project

Release

title

description

author

Date Started

Date Executed

Status (Pending, pass, fail)

Comments list of

date

comment Text

author

D.5 DEVELOPMENT

List of Coding Tasks:

Project

Release

List of Files

author

Date Started

Completion Date

Number of Unit Tests

Lines of Code

Others See [43], [44], [71], [76]

D.6 IMPLEMENTATION

List of:

Project

Release

date Entered

Date Scheduled

Date Executed

Ordering/Prerequisites

Comments list of

date

comment Text

author

REFERENCES

[1] M. Andreessen, "Why software is eating the world," *The Wall Street Journal*, Aug. 2011. [Online]. Available: http://goo.gl/MXk4TS.

[2] Ž. Antolić, "An example of using key performance indicators for software development process efficiency evaluation," in *MIPRO 2008: 31st International Convention on Information and Communication Technology, Electronics and Microelectronics, May 26-30, 2008, Opatija Croatia. Microelectronics, electronics and electronic technologies, MEET.. Grid and visualization systems, GVS*, P. Biljanović, K. Skala, and G. . V. Systems, Eds., vol. 1, MIPRO, 2008, ISBN: 9789532330366.

[3] B. Aulet, "Disciplined entrepreneurship: 24 steps to a successful startup," in. 2013, ch. STEP 19: Calculate the Cost of Customer Acquisition (COCA), ISBN: 1-118692-28-4.

[4] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou, "Evaluating static analysis defect warnings on production software," in *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, ser. PASTE '07, New York, NY, USA: ACM, 2007, pp. 1–8, ISBN: 978-1-59593-595-3. DOI: 10.1145/1251535.1251536.

[5] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas. (2001). Manifesto for agile software development, [Online]. Available: http://www.agilemanifesto.org/.

[6] H. D. Benington, "Production of large computer programs," in *Proceedings of the 9th International Conference on Software Engineering*, ser. ICSE '87, Los Alamitos, CA, USA: IEEE Computer Society Press, 1987, pp. 299–310, ISBN: 0-89791-216-0.

[7] B. W. Boehm, *Software Engineering Economics*, 1st. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1981, ISBN: 0138221227.

[8] B. W. Boehm, "A spiral model of software development and enhancement," *SIGSOFT Software Engineering Notes*, vol. 11, no. 4, pp. 14–24, Aug. 1986, ISSN: 0163-5948. DOI: 10.1145/12944.12948.

[9] B. W. Boehm, "A spiral model of software development and enhancement," *Computer*, vol. 21, no. 5, pp. 61–72, May 1988, ISSN: 0018-9162. DOI: 10.1109/2.59.

[10] B. W. Boehm and V. R. Basili, "Software defect reduction top 10 list," *Computer*, vol. 34, no. 1, pp. 135–137, Jan. 2001, ISSN: 0018-9162. DOI: 10.1109/2.962984.

[11] B. W. Boehm and W. J. Hansen, "Spiral development: experience, principles, and refinements," Carnegie Mellon Software Engineering Institute, Tech. Rep., Jul. 2000, Special Report CMU/SEI-2000-SR-008.

[12] R. P. Buse and T. Zimmermann, "Analytics for software development," in *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, ser. FoSER '10, New York, NY, USA: ACM, 2010, pp. 77–80, ISBN: 978-1-4503-0427-6. DOI: 10.1145/1882362.1882379. [Online]. Available: http://research.microsoft.com/pubs/136301/MSR-TR-2010-111.pdf.

[13] R. P. Buse and T. Zimmermann, "Information needs for software development analytics," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12, Piscataway, NJ, USA: IEEE Press, 2012, pp. 987–996, ISBN: 978-1-4673-1067-3.

[14] R. Charette, "Why software fails [software failure]," *IEEE Spectrum*, vol. 42, no. 9, pp. 42–49, Sep. 2005, ISSN: 0018-9235. DOI: 10.1109/MSPEC.2005.1502528.

[15] CMMI Product Team, "CMMI for Development, Version 1.3," Carnegie Mellon Software Engineering Institute (SEI), http://goo.gl/MBESq0, Tech. Rep., Nov. 2010.

[16] T. Copeland, *PMD Applied: An Easy-to-use Guide for Developers*, ser. An easy-to-use guide for developers. Centennial Books, 2005, ISBN: 9780976221418.

[17] E. Cowles and E. Nelson, *An Introduction to Survey Research*, 1st. New York, NY, USA: Business Expert Press, Jan. 2015, ISBN: 978-1-60649-818-7.

[18] J. Czerwonka, N. Nagappan, W. Schulte, and B. Murphy, "Codemine: building a software development data analytics platform at microsoft," *IEEE Software*, vol. 30, no. 4, pp. 64–71, 2013, ISSN: 0740-7459. DOI: 10.1109/MS.2013.68.

[19] A. Damodaran. (Feb. 2015). Statistical distributions, New York University, [Online]. Available: http://goo.gl/Jp7mtn.

[20] (Mar. 2015). Data-driven programming, Wikipedia, [Online]. Available: http://en.wikipedia.org/wiki/Data-driven_programming.

[21] T. DeMarco, "Software engineering: an idea whose time has come and gone?" *IEEE Software*, vol. 26, no. 4, pp. 96–96, Jul. 2009, ISSN: 0740-7459. DOI: 10.1109/MS.2009.101.

[22] A. B. Downey, *Think Stats: Probability and Statistics for Programmers*. O'Reilly Media, 2011, p. 138. [Online]. Available: http://greenteapress.com/thinkstats/.

[23] D. J. Dubois and G. Tamburrelli, "Understanding gamification mechanisms for software development," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ACM, 2013, pp. 659–662.

[24] C. Ebert, T. Liedtke, and E. Baisch, "Improving reliability of large software systems," *Annals of Software Engineering*, vol. 8, no. 1-4, pp. 3–51, Aug. 1999, ISSN: 1022-7091. DOI: 10.1023/A:1018971212809.

[25] K. E. Emam and A. G. Koru, "A replicated survey of it software project failures," *IEEE Software*, vol. 25, no. 5, pp. 84–90, Sep. 2008, ISSN: 0740-7459. DOI: 10.1109/MS.2008.107.

[26] M. Faizan, M. N. A. Khan, and S. Ulhaq, "Contemporary trends in defect prevention: a survey report," *International Journal of Modern Education and Computer Science (IJMECS)*, vol. 4, no. 3, p. 14, 2012.

[27] W. A. Florac and A. D. Carleton, *Measuring the Software Process*. Boston: Addison Wesley, 1999.

[28] C. Giardino, M. Unterkalmsteiner, N. Paternoster, T. Gorschek, and P. Abrahamsson, "What do we know about software development in startups?" *IEEE Software*, vol. 31, no. 5, pp. 28–32, Sep. 2014, ISSN: 0740-7459. DOI: 10.1109/MS.2014.129.

[29] S. Godfrey. (Oct. 2011). Characteristics of capability maturity model, [Online]. Available: http://goo.gl/MpNx9b.

[30] A. L. Goel and M. Shin, "Software engineering data analysis techniques (tutorial)," in *Proceedings of the 19th International Conference on Software Engineering*, ser. ICSE '97, New York, NY, USA: ACM, 1997, pp. 667–668, ISBN: 0-89791-914-9. DOI: 10.1145/253228.253816.

[31] M. Halkidi, D. Spinellis, G. Tsatsaronis, and M. Vazirgiannis, "Data mining in software engineering," *Intelligent Data Analysis*, vol. 15, no. 3, pp. 413–441, Aug. 2011, ISSN: 1088-467X.

[32] M. H. Halstead, *Elements of Software Science (Operating and Programming Systems Series)*. New York, NY, USA: Elsevier Science Inc., 1977, ISBN: 0444002057.

[33] A. E. Hassan, A. Hindle, P. Runeson, M. Shepperd, P. Devanbu, and S. Kim, "Roundtable: what's next in software analytics," *IEEE Software*, vol. 30, no. 4, pp. 53–56, Jul. 2013, ISSN: 0740-7459. DOI: 10.1109/MS.2013.85.

[34] A. E. Hassan and T. Xie, "Software intelligence: the future of mining software engineering data," in *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, ser. FoSER '10, New York, NY, USA: ACM, 2010, pp. 161–166, ISBN: 978-1-4503-0427-6. DOI: 10.1145/1882362.1882397.

[35] C. Hibbs, S. Jewett, and M. Sullivan, *The Art of Lean Software Development: A Practical and Incremental Approach*, 1st. O'Reilly Media, Inc., 2009, ISBN: 0596517319, 9780596517311.

[36] D. Hubbard, *How to Measure Anything: Finding the Value of Intangibles in Business*, 2nd. Wiley, 2010, ISBN: 9780470625699.

[37] E. A. Ichu, *The Role of Quality Assurance in Software Development Projects: Software Project Failures and Business Performance*. Germany: LAP Lambert Academic Publishing, 2012, ISBN: 3659169609, 9783659169601.

[38] "IEEE Standard Glossary of Software Engineering Terminology," *IEEE Std 610.12-1990*, pp. 1–84, Dec. 1990. DOI: 10.1109/IEEESTD.1990.101064.

[39] A. Iqbal, O. Ureche, M. Hausenblas, and G. Tummarello, "Ld2sd: linked data driven software development," in *In 21st International Conference on Software Engineering and Knowledge Engineering (SEKE 09*, 2009.

[40] I. Jacobson and E. Seidewitz, "A new software engineering," *Communications of the ACM*, vol. 57, no. 12, pp. 49–54, Nov. 2014, ISSN: 0001-0782. DOI: 10.1145/2677034.

[41] A. Jain and S. Angadi, "Gamifiying software development process," *Infosys Labs Briefings*, vol. 11, no. 3, pp. 21–28, 2013, http://goo.gl/T9PB96 accessed 01-Jan-2015.

[42] C. Jones, *Applied Software Measurement: Assuring Productivity and Quality*, 2nd. Hightstown, NJ, USA: McGraw-Hill, Inc., 1997, ISBN: 0-07-032826-9.

[43] C. Jones, *Software Engineering Best Practices*, 1st ed. New York, NY, USA: McGraw-Hill, Inc., 2010, ISBN: 007162161X, 9780071621618.

[44] C. Jones. (Jun. 2012). Scoring and evaluating software methods, practices, and results. Namecook Analytics Blog, [Online]. Available: http://goo.gl/3i06pN.

[45] C. Jones, "The technical and social history of software engineering," in, 1st. Addison-Wesley Professional, 2013, ch. 10, ISBN: 0321903420, 9780321903426.

[46] C. Jones. (Jul. 2013). WHY "COST PER DEFECT" IS HARMFUL FOR SOFTWARE QUALITY. Namecook Analytics Blog, [Online]. Available: http://goo.gl/QUsvlw.

[47] M. Jørgensen, "A strong focus on low price when selecting software providers increases the likelihood of failure in software outsourcing projects," in *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '13, New York, NY, USA: ACM, 2013, pp. 220–227, ISBN: 978-1-4503-1848-8. DOI: 10.1145/2460999.2461033.

[48] M. Jorgensen, "What we do and don't know about software development effort estimation," *IEEE Software*, vol. 31, no. 2, pp. 37–40, Mar. 2014, ISSN: 0740-7459. DOI: 10.1109/MS.2014.49.

[49] C. Kaner and W. P. Bond, "Software engineering metrics: what do they measure and how do we know?" In *METRICS 2004*, IEEE CS Press, 2004.

[50] R. S. Kaplan and D. P. Norton, "The balanced scorecard: measures that drive performance," *Harvard Business Review*, pp. 71–80, Jan. 1992.

[51] R. S. Kaplan and D. P. Norton, "Using the balanced scorecard as a strategic management system," *Harvard Business Review*, Jul. 2007. [Online]. Available: http://goo.gl/4v871V.

[52] M. Klubeck, *Metrics: How to Improve Key Business Results*, 1st. Berkely, CA, USA: Apress, 2011, ISBN: 1430237260, 9781430237266.

[53] M. Kutner, C. Nachtsheim, and J. Neter, *Applied Linear Regression Models*, 4th, ser. The McGraw-Hill/Irwin Series Operations and Decision Sciences. McGraw-Hill Higher Education, 2003, ISBN: 9780072955675.

[54] T. O. A. Lehtinen, M. V. Mäntylä, J. Vanhanen, J. Itkonen, and C. Lassenius, "Perceived causes of software project failures - an analysis of their relationships," *Journal Information and Software Technology*, vol. 56, no. 6, pp. 623–643, Jun. 2014, ISSN: 0950-5849. DOI: 10.1016/j.infsof.2014.01.015.

[55] E. Letier and C. Fitzgerald, "Measure what counts: an evaluation pattern for software data analysis," in *2013 1st International Workshop on Data Analysis Patterns in Software Engineering (DAPSE)*, IEEE, 2013, pp. 20–22.

[56] S. Maheshwari and D. C. Jain, "A comparative analysis of different types of models in software development life cycle," *International Journal of Advanced Research in Computer Science and Software Engineering*, vol. 2, no. 5, pp. 285–290, May 2012, ISSN: 2277-128X.

[57] A. Marcus and T. Menzies, "Software is data too," in *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, ser. FoSER '10, New York, NY, USA: ACM, 2010, pp. 229–232, ISBN: 978-1-4503-0427-6. DOI: 10.1145/1882362.1882410.

[58] T. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, Dec. 1976, ISSN: 0098-5589. DOI: 10.1109/TSE.1976.233837.

[59] T. Menzies, B. Caglayan, Z. He, E. Kocaguneli, J. Krall, F. Peters, and B. Turhan. (Jun. 2012). The promise repository of empirical software engineering data, [Online]. Available: http://promisedata.googlecode.com.

[60] T. Menzies and T. Zimmermann, "Goldfish bowl panel: software development analytics," in *2012 34th International Conference on Software Engineering (ICSE)*, Jun. 2012, pp. 1032–1033.

[61] J. P. Miguel, D. Mauricio, and G. Rodríguez, "A review of software quality models for the evaluation of software products," *International Journal of Software Engineering & Applications (IJSEA)*, vol. 5, no. 6, pp. 31–54, Nov. 2014, http://www.airccse.org/journal/ijsea/papers/5614ijsea03.pdf.

[62] A. B. M. Moniruzzaman and S. A. Hossain, "Comparative study on agile software development methodologies," *Global Journal of Computer Science and Technology*, vol. 13, no. 7, 2013. [Online]. Available: http://arxiv.org/abs/1307.3356.

[63] P. Naur and B. Randell, Eds., *Software Engineering: Report of a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO*. 1969, http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF.

[64] J. Olson, *Data Quality: The Accuracy Dimension*, ser. The Morgan Kaufmann Series in Data Management Systems. Elsevier Science, 2003, ISBN: 9780080503691.

[65] D. Parmenter, *Key Performance Indicators: developing, implementing, and using winning KPIs*. Hoboken, New Jersey: John Wiley and Sons, Inc., 2010.

[66] L. H. Putnam and W. Myers, *Five Core Metrics: the Intelligence Behind Successful Software Management*. New York, New York: Addison-Wesley Professional, 2013.

[67] R. Ramler and K. Wolfmaier, "Economic perspectives in test automation: balancing automated and manual testing with opportunity cost," in *Proceedings of the 2006 International Workshop on Automation of Software Test*, ser. AST '06, New York, NY, USA: ACM, 2006, pp. 85–91, ISBN: 1-59593-408-1. DOI: 10.1145/1138929.1138946.

[68] J. Raynus, *Software Process Improvement with CMM*. Norwood, MA, USA: Artech House, Inc., 1999, ISBN: 0-89006-644-2. [Online]. Available: http://goo.gl/Jc7Krf.

[69] J. Rost and R. Glass, "The dark side of software engineering: evil on computing projects," in. Wiley, 2011, ch. 2 Lying, ISBN: 9780470922873.

[70] W. W. Royce, "Managing the development of large software systems: concepts and techniques," in *Proceedings of the 9th International Conference on Software Engineering*, ser. ICSE '87, Los Alamitos, CA, USA: IEEE Computer Society Press, 1987, pp. 328–338, ISBN: 0-89791-216-0.

[71] V. Rubin, C. W. Günther, W. van der Aalst, E. Kindler, B. F. Van Dongen, and W. Schäfer, "Process mining framework for software processes," in *Proceedings of the 2007 International Conference on Software Process*, ser. ICSP'07, Berlin, Heidelberg: Springer-Verlag, 2007, pp. 169–181, ISBN: 978-3-540-72425-4.

[72] G. Ruhe and F. Gesellschaft, "Knowledge discovery from software engineering data: rough set analysis and its interaction with goal-oriented measurement," English, in *Principles of Data Mining and Knowledge Discovery*, ser. Lecture Notes in Computer Science, J. Komorowski and J. Zytkow, Eds., vol. 1263, Springer Berlin Heidelberg, 1997, pp. 167–177, ISBN: 978-3-540-63223-8. DOI: 10.1007/3-540-63223-9_116.

[73] N. B. Ruparelia, "Software development lifecycle models," *SIGSOFT Softw. Eng. Notes*, vol. 35, no. 3, pp. 8–13, May 2010, ISSN: 0163-5948. DOI: 10.1145/1764810.1764814.

[74] A. Saltelli, S. Tarantola, and F. Campolongo, "Sensitivity analysis as an ingredient of modeling," *Statistical Software*, vol. 15, no. 4, pp. 377–395, 2000. [Online]. Available: http://projecteuclid.org/euclid.ss/1009213004.

[75] G. Snijkers, G. Haraldsen, J. Jones, and D. Willimack, *Designing and Conducting Business Surveys*, 2nd ed., ser. Wiley Series in Survey Methodology. John Wiley & Sons, 2013, ISBN: 9781118447918.

[76] W. B. Snipes, "Evaluating developer responses to gamification of software development practices.," Master's thesis, North Carolina State University, 2013. [Online]. Available: http://repository.lib.ncsu.edu/ir/handle/1840.16/9199.

[77] I. Sommerville, *Software Engineering*, 6th ed. Harlow, England: Addison-Wesley, 2001.

[78] S. L. Spraragen, "The challenges in creating tools for improving the software development lifecycle," in *Proceedings of the 2005 Workshop on Human and Social Factors of Software Engineering*, ser. HSSE '05, New York, NY, USA: ACM, 2005, pp. 1–3, ISBN: 1-59593-120-1. DOI: 10.1145/1082983.1083118.

[79] R. Swanstrom. (Mar. 2015). Dissertation-scoring-sdo, [Online]. Available: https://github.com/ryanswanstrom/dissertation-scoring-sdo.

[80] Q. Taylor and C. Giraud-Carrier, "Applications of data mining in software engineering," *International Journal of Data Analysis Techniques and Strategies*, vol. 2, no. 3, pp. 243–257, Jul. 2010, ISSN: 1755-8050.

[81] D. Tosi, L. Lavazza, S. Morasca, and D. Taibi, "On the definition of dynamic software measures," in *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '12, New York, NY, USA: ACM, 2012, pp. 39–48, ISBN: 978-1-4503-1056-7. DOI: 10.1145/2372251.2372259.

[82] F. F. Tsui, *Essentials of software engineering*, 3rd. Jones & Bartlett Publishers, 2013.

[83] W. van der Aalst, *Process Mining : Discovery, Conformance and Enhancement of Business Processes*. Heidelberg: Springer, 2011.

[84] W. van der Aalst, "Process mining: overview and opportunities," *ACM Transactions on Management Information Systems*, vol. 3, no. 2, 7:1–7:17, Jul. 2012, ISSN: 2158-656X. DOI: 10.1145/2229156.2229157.

[85] M. van Genuchten, R. Mans, H. Reijers, and D. Wismeijer, "Is your upgrade worth it? process mining can tell," *IEEE Software*, vol. 31, no. 5, pp. 94–100, Sep. 2014, ISSN: 0740-7459. DOI: 10.1109/MS.2014.20.

[86] K. Werbach, "(re)defining gamification: a process approach," English, in *Persuasive Technology*, ser. Lecture Notes in Computer Science, A. Spagnolli, L. Chittaro, and L. Gamberini, Eds., vol. 8462, Springer International Publishing, 2014, pp. 266–272, ISBN: 978-3-319-07126-8. DOI: 10.1007/978-3-319-07127-5_23.

[87] V. Winter, C. Reinke, and J. Guerrero, "Sextant: a tool to specify and visualize software metrics for java source-code," in *Emerging Trends in Software Metrics (WETSoM), 2013 4th International Workshop on*, May 2013, pp. 49–55. DOI: 10.1109/WETSoM.2013.6619336.

[88] T. Xie, S. Thummalapenta, D. Lo, and C. Liu, "Data mining for software engineering," *Computer*, vol. 42, no. 8, pp. 55–62, 2009, ISSN: 0018-9162. DOI: 10.1109/MC.2009.256.

[89] D. Zhang, Y. Dang, J.-G. Lou, S. Han, H. Zhang, and T. Xie, "Software analytics as a learning case in practice: approaches and experiences," in *Proceedings of the International Workshop on Machine Learning Technologies in Software Engineering*, ser. MALETS '11, New York, NY, USA: ACM, 2011, pp. 55–58, ISBN: 978-1-4503-1022-2. DOI: 10.1145/2070821.2070829.

[90] D. Zhang, S. Han, Y. Dang, J.-G. Lou, H. Zhang, and T. Xie, "Software analytics in practice," *IEEE Software*, vol. 30, no. 5, pp. 30–37, Sep. 2013, ISSN: 0740-7459. DOI: 10.1109/MS.2013.94.