ABSTRACT

ANALYSIS OF RELATIONSHIP BETWEEN SOFTWARE METRICS AND PROCESS MODELS

by
Sadia Munir

This thesis studies the correlation between software process models and software metrics. To support our studies we have defined a Process - Metric Evaluation Framework and derived an evaluation template from it. The template served as a basic tool in studding the relationships between various process models, artifacts and software metrics.

We have evaluated a number of process models according to our template and have identified suitable software metrics. We have also recommended a root cause analysis approach at various points of the process models.

The suggested software metrics can be derived from various product and process artifacts. They can be used to curb the risks generated at each phase of the development process, identify issues, and do better planning and project management. The evaluation template can also be used to evaluate other models and identify effective metrics.

ANALYSIS OF RELATIONSHIP BETWEEN SOFTWARE METRICS AND PROCESS MODELS

by
Sadia Munir

A Thesis
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Science

Department of Computer Science

January 2005

APPROVAL PAGE

ANALYSIS OF RELATIONSHIP BETWEEN SOFTWARE METRICS AND PROCESS MODELS

Sadia Munir

Dr. Fadi P. Deek, Thesis Co-Advisor — Date
Professor of Information Systems, NJIT
Acting Dean of College of Science and Liberal Arts, NJIT

Dr. Vassilka Kirova, Thesis Co-Advisor — Date
Research Professor of Information Systems, NJIT

Dr. James McHugh, Committee Member — Date
Professor of Computer Science, NJIT

BIOGRAPHICAL SKETCH

Author: Sadia Munir

Degree: Master of Science

Date: January 2005

Undergraduate and Graduate Education:

- Master of Science in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, 2005

- Bachelor of Science in Computer Science,
  Foundation for Advancement of Science and Technology, Karachi, Pakistan, 2002

Major: Computer Science

To my dear parents and beloved brothers

ACKNOWLEDGMENT

Thanks to Dr. Fadi Deek and Dr. James McHugh for their encouragement and support.

Many thanks to Dr. Vassilka Kirova, who unflinchingly believed in my abilities and incessantly, motivated me at each step. She was there to help me at all times whether it was about writing the thesis, requiring encouragement, or giving advice and suggestions.

Thanks also to my friends who motivated me to work on this thesis and kept my spirits high through out my studies.

Great thanks to my family who encouraged and supported me at every step.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

Chapter **Page**

LIST OF TABLES

LIST OF TABLES
(Continued)

Table **Page**

CHAPTER 1

INTRODUCTION

1.1 Objective

The key purpose of software process models is to provide a structural framework to the development efforts. It drives software development through various phases in an organized and predictable way. A software process model provides insight about

- Which phases will occur, and in which sequence during software development.
- How the transition from one phase to another will occur.

Software metrics are used in conjunction with software process models. They are derived from the various artifacts or work products created as result of activities taking place throughout the process of software production.

The software metrics provide a quantitative measures of the project progress, artifact quality and teams efficiency. They facilitate in discovering any potential problems that can hinder the software development. Thus, software metrics serve as management and quality indicators for software projects. The outcomes of management indicators address concerns and benefit the software process and the outcomes of quality indicators address concerns and benefit the software product. There are numerous metrics that can be used to evaluate the progress during software development. The software metrics if utilized properly can help in delivering a quality product in time and within budget.

There are many questions that arise while employing the software metrics within the context of processes tailored according to different software process models.

In our work we have made an effort to determine if any of the software metrics can be beneficial if availed with any software process model. We have then determined a set of software metrics which provide useful results if coupled with specific software process model. And how this set of software metrics per software process model will help in managing and alleviating the risks faced in various stages of development.

1.2 Problem Statement

The major problem with software projects is that they are over budget, over schedule and have quality problems [43]. The complex project development causes a great threat to completion of project on time and within budget and with required product quality. To achieve these goals software projects have to be well planned and monitored, and software measures are an excellent tool for this. These measures keep a check on the process and the product. These measures show the "real" picture of how the project is progressing in terms of product and process. When stored and maintained, the measures also provide historical markers to compare against and improve the production practices at every next project. But even after the use of these measures, a large number of software projects fail to complete within schedule and under budget.

In an effort to improve the software measures and their application we have added another dimension to the usage of software metrics, namely their relationship to different software models. We have evaluated a number of metrics and have suggested some new once in the context of a representative set of process models such as Waterfall [23] and RUP (Rational Unified Process) [21]. This is the basis of my thesis: Exploring the

software measures and how they can be improved to in return improve the software management.

1.3 Background Information

As part of the researching the problem domain we have reviewed the following references [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19]. A common view supported in the references is that effective software management is critical to resolving the issues known as "software crises", and that this is possible by the means of improved use of software metrics.

Software Metrics is defined as the measurement of the software product and the process by which it is developed [3].

Software Metrics Classifications

Software Metrics are usually divided into two broad categories [2]:

- **Direct Measures:** Measured directly in terms of the observed attribute (usually by counting). Example, length of source code, duration of process, number of defects.

- **Indirect Measures:** Calculated from other direct and indirect measures. Example, Module Defect Density = Number of defects discovered / Length of source.

Software Metrics are also classified as [3]:

- **Product Metrics:** These are the measures of the software product at any stage of its development, from requirements to installed system. Product metrics may measure the complexity of the software design, the size of the final program, or the number of pages of documentation produced.

- **Process Metrics:** These are the measures of the software development process, such as overall development time, type of methodology used or the average level of experience of the programming staff.

Another way to classify software metrics is defined in [22] as:

- **Objective Metrics:** These measures give the same result, no matter how many times they are calculated by any person.

- **Subjective Metrics:** This metric "depends on the collector's judgment; may lead to incoherent and non repeatable measures".

The TAPROOT Framework

TAPROOT stands for Taxonomy PRecis for Object Oriented meTrics. As explained in [22] TAPROOT is defined in two vectors: *category* and *granularity*.

Category represents the attributes of the product and the process model phases with which the software metrics have been effectively utilized. Following are the categories defined in TAPROOT Framework:

- Design
- Size
- Complexity
- Reuse
- Productivity
- Quality
- Generic Approach

The other vector is granularity. It is the appropriate level of abstraction on which object oriented metrics should be calculated. Following are the granules defined in TAPROOT Framework:

- Method
- Class
- System

According to TAPROOT there are various object oriented metrics that can be calculated. The framework for these metrics is presented in Table 1.1.

Table 1.1 TAPROOT Framework

	Method	**Class**	**System**
Design	MD	CD	SD
Size	MS	CS	SS
Complexity	MC	CC	SC
Reuse	MR	CR	SR
Productivity	MP	CP	SP
Quality	MQ	CQ	SQ

Software Metrics and Project Team

Software Metrics help all the project team members:

- Software Developer uses the metric to track his progress. The metrics help in answering the following questions: How many classes have to be written? What is the complexity of the code? How many lines of code have been written?
- System Analyst uses the metrics to keep track of the identified requirements. The metrics help in answering the following questions: How many requirements are stable? How many classes are there? How deep is class hierarchy?

- Project Manager uses the metrics to keep track of the project. The metrics help in answering the following questions: What is the project's current status? How do the estimates compare to the actual values? How much development has been done?

Besides the above mentioned team members, metrics are useful for all the stakeholders (managers, process engineers, customers, testers) of the project.

The complexity of software development is the main cause of failure of current software management. Due to few reliable and well-defined measures, it is difficult to keep a check on the complex software development.

The need of the hour is for such software metrics, which not only describe the current state of the project, but also facilitate the development at every phase of the project.

Core Software Metrics

The *seven core metrics* as identified by [1] are described below.

The metrics have been divided into two types of indicators:

1. **MANAGEMENT INDICATOR**
    - *Work and Progress*
    - *Budgeted cost and expenditures*
    - *Staffing and team dynamics*
2. **QUALITY INDICATOR**
    - *Change traffic and stability*
    - *Breakage and modularity*
    - *Rework and adaptability*

- *Mean time between failures (MTBF) and maturity*

Some specific statistics must be collected over the software life cycle to implement the proposed metrics. These metrics have been listed in Table 1.2, Table 1.3 and Table 1.4.

Table 1.2 Collected Statistics and their definitions [1]

COLLECTED STATISTICS	DEFINITON
Total SLOC	$SLOC_T$=total size in SLOC
Configured SLOC	$SLOC_C$=current baseline SLOC
Critical Defects	SCO_0=number of type 0 SCOs
Normal Defects	SCO_1=number of type 1 SCOs
Improvements	SCO_2=number of type 2 SCOs
New Features	SCO_3=number of type 3 SCOs
Number of SCOs	$N=SCO_0+ SCO_1 + SCO_2$
Open Rework (breakage)	B=cumulative broken SLOC due to SCO_0, SCO_1, and SCO_2
Closed Rework (fixes)	F=cumulative fixed SLOC
Rework Effort	E=cumulative effort expended fixing SCO_0, SCO_1, and SCO_2
Usage Time	UT=hours that a given baseline has been operating under realistic usage scenarios

These metrics help in maintainability of the software products with respect to type 0, 1 and 2 SCOs. Type 3 SCOs are not included because they redefine the quality of the system.

Table 1.3 End Product Quality Metrics and their definitions [1]

METRIC	DEFINITION
Scrap Ratio	$B/SLOC_T$
Rework Ratio	E/Development Effort
Modularity	B/N
Adaptability	E/N
Maturity	$UT/(SCO_0 + SCO_1)$
Maintainability	(scrap ratio)/(rework ratio)

Table 1.4 In-Progress Indicators and their definitions [1]

INDICATOR	DEFINITION
Rework Stability	B-F
Rework Backlog	(B-F)/$SLOC_C$
Modularity trend	Modularity plotted over time
Adaptability trend	Adaptability plotted over time
Maturity trend	Maturity plotted over time

1.4 Hypothesis

The following are the hypothesis that we evaluate through our work:

- Not all software metrics are applicable and equally useful with any software process model. Value can be derived from the software metrics only if the correct set of metrics is employed with the software process model. Collecting the measures, calculating the metrics, monitoring it and performing RCA (Root Cause Analysis) consume time, staff, and money – all of which are critical factors in a project. Therefore, wasting project resources on computing ineffective software metrics, which will not provide any actionable data and reflect properly on the progress, team motivation and quality of the product as well as the management of the project, is not advisable.
- The above premise leads to the following:
    - Identifying a set of software metrics for a software process model, which will provide valuable insight into the process and the product.
    - Characteristics of a software process model relevant to a project's phase, product and project management artifacts derived in a particular phase – these all contribute towards the computation of software metrics.

The goal is to define a comprehensible association between the characteristics of a process model, various artifacts produced in a phase, and the software metrics, such that, by varying the "strength" of the characteristics of the process model, we can achieve our desired affect (value) in the software metrics.

- A strong correlation exists between the software metrics indicators and the risks associated with each phase of the project. This excludes any risks that are inherent to the software process model. Analyzing how the software metrics can be used to reduce risks.

CHAPTER 2

LITERATURE REVIEW

2.1 Study of Process Models

A process model differs from a software methodology. The primary function of a process model is to facilitate in determining what should be done next in a phase and how long should a particular phase last. While a software methodology guides how to "navigate through each phase" [23], and how to develop artifacts in a phase.

The process models are essential because they provide guidance on the order in which the tasks should be done in a project plan. [23] elaborates on the evolution of process models as follows:

Code and Fix Model [23]

This was a basic process model used in the premature days of software development. It comprised of two main steps:

- Write a piece of code,
- Fix the problems in the code and proceed.

This process model did not incorporate requirements gathering, analysis, design, testing or maintenance of the project. All these major building blocks were left for later pondering.

The basic structure of the code and fix model caused many problems. After numerous fixes in the code, the code would become so weakly structured that it could not

accommodate any more fixes or the fixes were very expensive. Often the end product will not meet the requirements of the user. The code would be expensive to fix. These problems clearly indicate a need for proper phases before the coding of the software starts.

Stage – Wise Model [23]

After code and fix model, the Stage-Wise model was introduced as an outcome of the problems listed above. Stage-wise model specified that the software be developed in consecutive stages. In this approach after the end of a stage, it was not possible to repeat the same stage.

Waterfall Model [23]

Waterfall model was an enhancement to the Stage-Wise model. The two main areas where the Stage-Wise model was fine-tuned were as follows:

- A *feedback loop* was created between successive stages. This feedback loop could not be extended to more than two successive stages since it would require steep modifications.
- *Prototyping* was incorporated in the requirements and design stages. These improvements purged many impediments faced by the Stage-Wise model. But the basic structure of Waterfall model had some embedded issues.

Waterfall model, did not allow easy navigation between the various stages of the project. Besides that it required fully structured artifacts at the end of a stage such as a complete

requirements document at the end of the requirements stage. This is can be a major problem for highly end-user interactive systems.

Evolutionary Development Model [23]

The above mentioned problems led to the Evolutionary Development model. The model focuses on development of the software based on a subset of elicited requirements. The users work on the software and clarify the requirements or specify a new set of requirements. Changes are made in the operational software based on the feedback obtained from the users. The quick initial operational experience of the users is a positive point of evolutionary development model. This leads to an early response from the users which lead to product improvements.

The difficulties faced in evolutionary development are not much different from the struggles of the code and fix model. Evolutionary development can also lead to an unplanned, unmanageable spaghetti code. This model also makes an idealistic assumption that all the user's change requests which were not initially planned for can be easily accommodated in the operational software. This can be a major problem if a number of independent applications have to be integrated; if the temporary patches placed on the operational software to correct the defects have caused unalterable constraints; if the new system is going to be an increment of a poorly modularized old system.

Transform Model [23]

The solution to spaghetti code is the Transform Model. The Transform Model assumes that the formal specifications can be automatically converted into a program that satisfies those formal specifications.

The Transform Model stipulates the following: the formal specifications should be the true representation of the initial understanding of the product; the transformation system will be able to improve the product by following specific guidelines provided to it; the product can be adjusted based on the operational experience.

The Transform Model does not produce spaghetti code since changes are not made to the code directly. Changes are made to the formal specifications, which in result are transformed into the modified code through the transformation system. Transform Model seems to save the cost and effort incurred on the design, code and testing stages.

The Transform Model can only be used for small products in very few areas, such as "spreadsheets, small 4GL applications and limited computer science domains". Another issue faced by Transform Model is that it assumes that all the unanticipated change requests can be accommodated in the operational product.

Rational Unified Process [21]

Rational Unified Process is a disciplined approach. The purpose is to make certain a development of a high quality product which is within the boundaries of schedule and cost and also fulfils the requirements set forth by the users. RUP emphasizes on creation of rich models that are representations of the system under development.

The best practices observed in RUP are developing software iteratively, managing requirements, using component based architectures, visually modeling software, verifying software quality and controlling changes to the software. RUP is described along two dimensions. The horizontal dimension represents time and dynamic aspects of the process. The vertical dimension represents the static aspects of the process.

There are four main phases in RUP: Inception, Elaboration, Construction and Transition. The paper proceeds to discuss in detail the four phases, the artifacts developed in each phase, the key characteristics of each phase and the evaluation criteria at the end of each phase. The artifacts listed in the paper are useful in determining the metrics that can be derived from those artifacts.

Spiral Model [32]

Spiral development is an iterative, risk-driven process which continually enhances the software due to its cyclic approach. This cyclic approach towards development reduces the degree of risk. The article focuses on a set of six Spiral Model essentials. These six essentials have been marked as criteria for success while following the Spiral Model approach.

The first essential for Spiral Model has been defined as concurrent determination of key artifacts. To develop a product that matches the expectations of the stakeholders, it is necessary that certain key artifacts are developed concurrently and not sequentially. Some of the key artifacts that should go through concurrent development are operational concept, system and software requirements, project plan, system and software architecture and design, code components, COTS, reused components, prototypes, any critical components for success and algorithms. If these artifacts are sequentially developed then there are fewer chances of satisfying the stakeholders and hence hampering the project.

For a critical system based on low technology emphasis will be given on requirements artifact. While for a high technology system, the emphasis is on intensive prototyping.

If this Essential is not taken into account during spiral model development then it can result in premature commitments which will be difficult to materialize as the project proceeds. As the article suggests, such premature commitments can include requirements for hardware platforms, incompatible combinations of COTS components, and requirements whose achievability has not been validated. In cases where such requirements exist which seem difficult to achieve, then concurrent prototyping is highly recommended. This will cut down the development costs, save project time and effort.

Spiral Model is based on the belief that requirements are not pre specifiable, especially for innovative user interface systems. Spiral Model takes into account rapidly changing requirements for such projects where the technology is volatile and the marketplace is high.

In the second Essential for Spiral Model, each cycle concentrates on critical stake holder objectives and constraints, alternatives, risks, reviews and commitment to proceed. Taking into account, any assessments of the project and any process substitutions that will ease in meeting the objectives that are marked with constraints and risks. This Essential element does not provide directive to the level of effort required for each activity in the model. Any development alternatives should be given due consideration before any choice of technology is made. By ignoring the alternatives, a lot of effort can be wasted on basing development on an alternative which could have earlier been proved to be unsatisfactory based on the objectives and constraints set forth by the stake holders. A balance should be maintained between the risk of knowing too little about various process and development alternatives and wasting time and effort on gathering too much information which will be of less value. This Essential also emphasizes on due participation of stakeholders at various stages in the project.

The third Essential for Spiral Model, helps you decide what level of effort is required for an activity based on the risks associated with it. Risk Exposure can be calculated as Probability (Loss) into Size (Loss). The other risks are of putting little effort in the project that can lead to project error, and the other one due to putting too much effort in the project leading to project delays. This Essential does not dictate the choice of method to be followed in each activity. It also does not lay down any rules for the degree of detail in the artifacts produced during each activity.

The fourth Essential drives the degree for detail driven by risk considerations. The article suggests that if it is risky to not specify precisely, do specify. And if it is risky to specify

precisely, do not specify. In Spiral Model, it will be too risky to have all the requirements pre specified.

The fifth Essential emphasizes on deliverance of intermediate milestones that will serve as progress and commitment checkpoints. The article suggests the development of Life Cycle Objectives, Life Cycle Architecture and Initial Operating Capability. These milestones avoid analysis paralysis, unrealistic expectations, requirements creep, architectural drift, COTS shortfalls and incompatibilities, unsustainable architectures, traumatic cutovers, and useless systems. This Essential provides the ability to merge the milestones in cases where the framework covers the deliverables of a milestone in another milestone.

The sixth Essential emphasizes on system and life cycle activities and artifacts. The following questions need to be answered, if the product satisfies the stakeholders? Will it meet cost and performance goals? Will it integrate with existing business practices?

Agile processes have been described to have a high capability of adapting changing requirements with time. A main characteristic of agile process is to develop the software rapidly such that it also meets the requirements of the user. Agile processes are mostly used with small projects but in theory have been recommended for projects of all sizes.

Agile Models

This paper discussed the various characteristics of agile process model and the different factors associated with its success.

The main characteristics of agile models have been defined as the following:

- The agile processes have a tight feedback iteration that keeps the factor of constant improvement in the software at a high level. The iterations can span from a day to couple of weeks depending upon the agile process chosen (example, XP or Scrum).

  The non-agile process models also have iterations for ensuring the development of a high quality product. But the paper suggests that these iterations span a long period of time, and reviews for these iterations are quite long.

- The other main characteristics of agile processes are working in groups and effective communication between the team members. The groups play a vital role in agile processes by following the practice of collective ownership.

The paper goes further to discuss the various benefits and drawbacks of agile process models.

SCRUM

SCRUM is an enhancement of incremental process model. SCRUM is based on the principle that system analysis, design and development activities are considered as a "controlled black box" [24]. Controls are employed to manage the loose software development activities. SCRUM gives an end product definition in the initial plan, and allows the product to evolve in the sprints.

As described in [24], SCRUM accommodates the variables in its product releases. The variables are customer requirements, time pressure, competition, product quality, system vision and resource availability. These variables are initially defined in the project plan.

SCRUM has been described in [24] to have the following phases: Planning, Architecture/Design Phase, Sprint and Closure. Planning and Closure phases consist of "defined processes".

"SCRUM approaches software development from a patterns perspective" [25]. The patterns used in SCRUM are "Backlog, Sprints, Scrum Meetings, and Demos" [25].

Two types of backlogs are maintained in SCRUM. Product Backlog consists of a list of all the requirements associated with the system. Sprint Backlog is a subset of Product Backlog. It is a list of requirements which are to be implemented in a specific Sprint. Sprint is a 7 to 30 day activity. Sprint Backlog is used for the development in the Sprint. While a Sprint is going on, no changes are allowed to the Sprint Backlog. Due to this static nature of Sprint Backlog it is advised to keep a reasonable length of Sprint, which will not affect the accommodation of changing requirements adversely.

There are three types of Scrum Meetings: Sprint Planning Meeting, Daily Scrum and Sprint Review Meeting.

Sprint Planning Meeting is held at the beginning of the Sprint. The Product Backlog is discussed along with "existing product, business and technology conditions" [7]. The Sprint Backlog along with Sprint Goal is created. A Sprint Goal is a brief summary of the Sprint.

Daily Scrum Meeting is held everyday during the Sprint. It is attended by all the team members and stakeholders. The Sprint progress is discussed in the meeting. Any road blocks are identified and registered in the Blocks List. Changes might be made to the Sprint Backlog list as progress is made in the Sprint.

Sprint Review Meeting is held at the end of the Sprint. A demo of the release is given to the team and stakeholders. An informal discussion is held over the activities of the Sprint. "Scrum does not focus on a large list of specific practices, artifacts, phases, or milestones that would define an entire product lifecycle. It focuses more on defining some enabling practices and patterns that allow you to move quickly and deftly while minimizing the risk that it all will end in chaos"[25].

Scrum is a good approach for those organizations that do not require extensive monitoring of the development team. Direct observation plays an important role in monitoring the progress of the team. Since Scrum does not create a number of artifacts therefore less metrics can be derived from the Scrum process model. Much of the judgment about the team and the project progress has to be gained through direct observation. Extra artifacts can be attached to Scrum to monitor progress and to produce metrics but this will hinder the progress of the team rather than speeding them up in any way.

Extreme Programming - XP

XP has been defined as a light weight methodology that defies the standard software methodology practices. It is recommended for small teams working on small projects. XP does not have separate phases for planning, requirements, design, coding, testing and maintenance. Rather it does a bit of all these activities throughout the life cycle.

[26] has stated three positive points of XP:

- XP helps if the requirements are unclear in the beginning to the customers. Due to XP the customers get to see a rapidly developed system which is open for corrections after getting customer feedback.
- XP emphasizes on keeping the design simple but functional. This helps in scaling the system and in making any future changes to the system.
- In XP developers do the testing along with coding.

[26] goes on and describes a few "possible problems with XP":

- XP does not take into consideration the Problem Analysis phase. Therefore XP assumes that the customers have a clear understanding of the problem domain.
- A thorough Requirements Engineering Phase is lacking in XP. By avoiding this phase the team loses the opportunity to plan the system.
- XP has been recommended for small size and small duration projects. XP believes in little or no documentation. In a long project, if a team member leaves the project then understanding the code can be a problem. Here one of the practices of XP comes to the rescue: Keep the design simple. This is a controversial issue, as one school of thought believes that documentation is essential no matter how simple the code is. While XP argues that the design and code should be simple enough for anyone to understand it. And secondly due to pair programming approach, two people are always working on the code. So even if one team member has to leave the other team member should know how the code works.

- If the system has a large user interface, then creating automated tests can be a problem. To overcome this manual tests have to be done, which can slow down the development process.

[27] talks about XP and CMM. "The conclusion is that lightweight methodologies such as XP advocate many good engineering practices, although some practices may be controversial and counter-productive outside a narrow domain. For those interested in process improvement, the ideas in XP should be carefully considered for adoption where appropriate in an organization's business environment since XP can be used to address many of the CMM Level 2 and 3 practices. In turn, organizations using XP should carefully consider the management and infrastructure issues described in the CMM."

2.2 Objectives of Software Metrics

There are various objectives of software metrics which should be kept in mind while calculating the metrics. [28] and [22] discuss the objectives. The objectives have been defined on the following terms: "

1. To collect objective information about the current state of a software product, project or process.
2. To allow managers and practitioners to make timely, data-driven decisions.
3. To track your organization's progress toward its improvement goals.
4. To assess the impact of process changes. "

Software Metrics give a deeper insight in to the development process which in return helps in judging if any improvements are occurring in the product and the process.

Metrics gathered from the initial phases of the project help in recognizing any potential problems for the future phases of the project.

2.3 Software Metrics and Mistakes to Avoid

It is important to keep the above objectives in mind while calculating and collecting metrics. While working with software metrics [28] has suggested a list of mistakes that should be avoided. The list is as follows:

1. Management is not backing up the measurement process.
2. Do not start the measurement process with an exhaustive list of software metrics to be calculated. This can cause more resistance in the organization.
3. Do not start with too little captured data, which will be of no use to the stake holders.
4. The attributes being measured are not answering the questions raised by the stake holders. This shows that wrong attributes are being measured.
5. Metrics might be defined vaguely. This leads to inconsistent interpretations of the same metric definition.
6. Using metrics to evaluate the performance of the individuals.
7. Do not use metrics data to reward "desired behaviors". These rewards can be given by considering only a small subset of the metrics.
8. The data being gathered for metrics calculation purposes is not being used at all.
9. Stakeholders do not realize the importance and objective of metrics. Due to this a lot of resistance is experienced in the organization.

10. The stakeholders do not understand the metrics completely. This lack of knowledge leads to misinterpretation of metrics.

2.4 Traditional Software Metrics

According to [29], traditional software metrics are used in parallel with the procedural approach. After the introduction of object oriented metrics, traditional software metrics were enhanced to capture the object oriented characteristics such as encapsulation, inheritance and polymorphism. Some still disagree upon the idea that traditional metrics can be used for an object oriented approach.

In [31], a list of traditional software metrics has been given for further reading. The list is provided here for the readers:

- **Average Module Length**, measures the average module size.
- **Binding among Modules**, measures data sharing among modules.
- **Cyclomatic Complexity Number**, measures the number of decisions in the control graph.
- **Control flow Complexity and Data Flow Complexity** is a combine metric based on variable definitions and cross-references.
- **Conditions and Operations Count** counts pairs of all conditions and loops within the operations.
- **Complexity Pair** combines cyclomatic complexity with logic structure.
- **Coupling Relation** assigns a relation to every couple of modules according to the kind of coupling.

- **Cohesion Ratio Metrics** measure the number of modules having functional cohesion divided by the total number of modules.
- **Decision Count** offers a method to measure program complexity.
- **Delivered Source Instructions** counts separate statements on the same physical line as distinct and ignores comment lines.
- **Extent of Reuse** categorizes a unit according the lever of reuse.
- **Equivalent Size Measure** measures the percentage of modifications on a reused module.
- **Executable Statements** counts separate statements on the same physical line as distinct and ignores comment lines, data declarations and headings.
- **Function Count** measures the number of functions and the source lines in every function.
- **Function Points** measures the amount of functionality in a system.
- **Global Modularity** describes global modularity in terms of several specific views of modularity.
- **Information Flow** measures the total level of information flow between individual modules and the rest of a system.
- **Knot Measure** is the total number of crossing points on control flow lines.
- **Lines of Code** measure the size of a module.
- **Live Variables** deals with the period each variable is used.
- **Minimum Number of Paths** measures the minimum number of paths in a program and the reach ability of any node.

- **Morphology metrics** measure morphological characteristics of a module, such as size, depth, width and edge-to-node ratio.
- **Nesting Levels** measures the complexity as depth of nesting.
- **Composite Metric** of Software Science and Cyclomatic Complexity combines software science metrics with McCabe's complexity measure.
- **Software Science Metrics** are a set of composite size metrics.
- **Specifications Weight Metrics** measure the function primitives on a given data flow diagram.
- **Tree Impurity** determines how far a graph deviates from being a tree.
- **Transfer Usage** measures the logical structure of the program.

2.5 Object Oriented Metrics

Object oriented metrics were designed for object oriented measurements. These metrics can be used in analysis, design and production phases to determine the quality of the attributes being measured. A brief survey of some of the object oriented metrics listed in literature is stated as follows:

Moreau and Dominick

[29] has stated that some of the earliest work on object oriented metrics was performed by Moreau and Dominick. They derived the following three metrics:

1. **Message Vocabulary Size:** It is the "number of different types of message sent by a particular object".

2. **Inheritance Complexity:** It is the size of the inheritance tree.
3. **Message Domain Size:** It is the "number of types of messages to which an object will respond".

Morris

Morris derived some interesting productivity metrics for object oriented approach. [30] gives a listing of these metrics as:

1. **Methods per Class:**

   Average number of methods per object class =

   Total number of methods/Total number of object classes.

2. **Inheritance Dependencies:**

   Inheritance tree depth = max (inheritance tree path length).

3. **Degree of Coupling Between Objects:**

   Average number of uses dependencies per object =

   Total number of arcs / Total number of objects

   Arcs = max (number of uses arcs) – in an object uses network

   Arcs – attached to any single object in a uses network

4. **Degree of Cohesion of Objects:**

   Degree of Cohesion of Objects =

   Total Fan-in for All Objects / Total Number of Objects.

5. **Object Library Effectiveness:**

   Average Number =

   Total Number of Object Reuses / Total Number of Library Objects.

6. **Factoring Effectiveness:**

Factoring Effectiveness = Number of Unique Methods / Total Number of Methods.

7. **Degree of Reuse of Inheritance Methods:**

   Percent of Potential Method Uses Actually Reused =

   (Total number of Actual Method Uses / Total Number of Potential Method Uses) * 100

   Percent of Potential Method Uses Overridden =

   (Total Number of Methods Overridden / Total Number of Potential Method Uses) * 100

8. **Average Method Complexity:**

   Average Method Complexity =

   Sum of the cyclomatic complexity of all Methods / Total Number of Application Methods.

9. **Application Granularity:**

   Application Granularity = Total Number of Objects / Total Function Points.

Chidamber and Kemerer

Chidamber and Kemerer continued with the set of metrics derived by Morris. [30] gives details of the Chidamber and Kemerer suite of metrics.

1. **Weighted Methods per Class:** It is defined as the "sum of the complexities of all methods of a class".
2. **Depth of Inheritance Tree:** It is defined as the "maximum length form the node to the root of the tree".
3. **Number of Children:** It is defined as the "sum of immediate subclasses".

4. **Coupling between Object Classes:** It is defined as the "count of classes to which the class is coupled".
5. **Response for a Class:** It is defined as the "number of methods in the set of all methods that can be invoked in response to a message sent to an object of a class".
6. **Lack of Cohesion in Methods:** It is defined as the "number of different methods within a class that reference a given instance variable".

MOOD (Metrics for Object Oriented Design) or Brito e Abreu

MOOD metrics have been described in [30] as:

1. **Method Hiding Factor:** It is defined as the "ratio of the sum of the invisibilities of all methods defined in all classes to the total number of methods defined in the system under consideration".
2. **Attribute Hiding Factor:** It is defined as the "ratio of the sum of the invisibilities of all attributes defined in all classes to the total number of attributes defined in the system under consideration".
3. **Method Inheritance Factor:** It is defined as the "ratio of the sum of the inherited methods in all classes of the system under consideration to the total number of available methods (locally defined plus inherited) for all classes.
4. **Attribute Inheritance Factor:** It is defined as the "ratio of the sum of inherited attributes in all classes of the system under consideration to the total number of available attributes".

5. **Polymorphism Factor:** It is defined as the "ratio of the actual number of possible different polymorphic situation for class C_i to the maximum number of possible distinct polymorphic situations for class C_i.
6. **Coupling Factor:** It is defined as the "ratio of the maximum possible number of couplings in the system to the actual number of couplings not imputable to inheritance".

Li and Henry

Li and Henry based their metrics on Chidamber and Kemerer. The new metrics introduced by them as stated in [29] are:

1. **Message-Passing Coupling:** It is defined to "measure the complexity of message passing among classes".
2. **Data Abstraction Coupling:** It is defined as the "number of instances of Abstract Data Types".
3. **Number of Methods:** It is defined as the "number of local methods".
4. **Number of semicolons:** This is a traditional size metric.
5. **Number of properties:** It is a size metric. It is defined as the "number of attributes plus the number of local methods".

There are a number of other metrics discussed in [29] from Chen and Lu to measure the complexity of the classes; Abbott, Korson and McGregor about choosing design alternatives; and Hitz and Montazeri about object level and class level coupling.

2.6 Software Metrics and Project Monitoring

Various processes have been discussed in [39] for deriving the metrics, analyzing the values produced by metrics and interpreting the results for product and process improvement.

[39] suggests the use of the four-stage model which comprises of identifying any alarming values shown by metrics, understanding the rational behind those values, and the process of correcting the identified problems.

It is suggested that the project monitoring should be done at all phases of the project by computing metrics at all project phases. Baselines are recommended as a set of measurements. If the measured value varied a lot from the average then the reasons for abnormality need to be investigated.

It is stated in [39], that the initial study of quality metrics suggested that there was no strong correlation between the quality metrics and final product quality because of two main reasons:

- A metric's value can be a result of various factors and,
- Secondly no interpretation scale existed for software metrics.

Two types of project monitoring activities have been supported in [39]:

- *Checkpoint monitoring*
- *Continuous monitoring*

Checkpoint monitoring occurs at the end of the phase when the promised milestones have been delivered. Continuous monitoring has been further classified as snap shot monitoring and time-based monitoring.

Snapshot monitoring [39] can be done during a phase at any instant. It is quite similar to Checkpoint monitoring, but the main difference is that the results obtained from Snapshot monitoring can most probably be invalid. This is highly possible, if the Snapshot monitoring is done when the components under development are in various stages.

Time-based monitoring can be more reliable than the above mentioned methods. It involves monitoring the status of a specific metric over a period of time. And any disturbing variances can be used as an alarm.

The monitoring model suggested above has its own assumptions. The monitoring model assumes that during requirements specification, acceptance testing plans are created. During high level design, integration test plans and system test plans are created. During detailed design, black box testing plans are created. During coding, white box testing plans are created. The test plans comprise of the test cases. Besides this regular code inspection, reviews and walkthroughs are held. These exercises help in pointing out any potential problems.

A general approach towards using software metrics has been defined as follows in [39]:

- Setting quantifiable targets for each phase. These targets should be such that they can be translated into an estimate of effort, cost and time. Estimating the targets can be complicated and erroneous depending on the estimation model chosen.
- Capture the actual values from each phase.
- Compare the actual values with the targeted values.
- Devise a plan to address and correct any variations from the targeted values.

In [39], various categories of metrics have been suggested for project monitoring purposes along with the proper methods for estimating their target values. Reasons for deviation have been discussed in detail and the methods by which these deviations can be properly corrected.

2.7 Software Metrics and Project Management

After collecting all the software metrics, the resultant data will be utilized for making better project management decisions. Even if the data is gathered from one project, it will provide some insight into the development process. This insight will become deeper once the data increases with time from various other projects. If it is determined that the data can be used as a "norm" then future project plans can be compared with this data. If for future projects the values are away from the norm then the analysis can be done on what lead to such variations. Such analysis can help identify problems in the process being followed.

Collecting the data for the metrics can be a problem if the team and other stakeholders do not realize the benefit of the metrics. Team might be afraid that the gathered data will be used against them to judge their performance. Team might think that it takes too long to gather the data for the metrics. [4] suggests that to overcome such hurdles a software measurement culture needs to be established. All the stakeholders need to be committed to the use of metrics. All the team members need to understand the objectives and benefits of metrics. It is suggested in [4] that the privacy of data should be respected at all times to maintain the faith of the team members in metrics. The metrics process should be started with a small set of metrics being calculated. This set should be able to address the

questions raised by the various stakeholders. A fair rationale behind calculating this small set of metrics should be present. The team should be aware of this rationale to avoid any resistance from their side. Once the metrics are calculated, the results should be shared with the team, with explanations on how the data is going to be used. Share with the team members on what trends are visible in the collected data and how that can benefit the project. [4] suggests some metrics that are appropriate for managing at the individual, project team or organization level:

Individual Level

- Work effort distribution
- Estimated vs. actual task duration and effort
- Code covered by unit testing
- Number of defects found by unit testing
- Code and design complexity

Project Team

- Product size
- Work effort distribution
- Requirements status
- Percentage of test cases passed
- Estimated vs. actual duration between major milestones
- Estimated vs. actual staffing levels
- Number of defects found by integration and system testing
- Number of defects found by inspections

- Defect status
- Requirements stability
- Number of tasks planned and completed

Development Organization

- Released defect levels
- Product development cycle time
- Schedule and effort estimating accuracy
- Reuse effectiveness
- Planned and actual cost

CHAPTER 3

EVALUATION FRAMEWORK

3.1 Process – Metric Evaluation Template

The template shown in Table 3.1 will be used for analyzing a software process model in parallel to the software metrics. The analysis will be done by examining the particular project phase along with any characteristics of the process model that apply to the project phase. All the product and project management artifacts that are received as a byproduct of this phase will be listed here.

The software metrics will be derived from the product and project management artifacts. The direct metrics will be picked up from the artifacts without any computations required. The indirect metrics can be a combinational result of direct or indirect metrics. Product metrics will mostly be derived from the product artifacts and will reflect upon the progress of the development at various stages. Process metrics will be derived from the process artifacts and will reflect upon the progress of the project plan.

After deriving a relationship between the various metrics, artifacts, characteristics of the process model and the phases of the project we will focus on the relationship between software metrics and risks. We will study how the changes in metrics can effectively reduce the related risks. If the risks are marked high then which software metrics can play a key role in bringing the risk level down. To improve the software metrics which artifacts will be affected? How the future iterations or phases can be improved in order to reduce the risks.

These are the questions that we will address for each process model.

Table 3.1 Process – Metric Evaluation Template (To be applied to a particular phase of a process model)

Phase		Particular phase of the project
Characteristics of the Software Process Model		Any distinct characteristics of the process model that apply to this particular phase of the project.
Product Artifacts		Created in this phase for managing the development of the product.
Project Management Artifacts		Created in this phase for managing the project and the software process model.
Product Metrics	Direct Metrics	Measured directly from the observed attributes in the product artifacts
	Indirect Metrics	Calculated from the direct or other indirect product metrics.
Process Metrics	Direct Metrics	Measured directly from the observed attributes in the process artifacts
	Indirect Metrics	Calculated from the direct or other indirect process metrics.
Risks		Any risks associated with this phase of the project.
Recommended Root Cause Analysis Approach		Any points that need to be considered due to an alarm raised by software metric.

3.2 Process – Metric Evaluation Framework

The study of the process models on the above terms will help in determining a set of metrics which will improve the process and the product. It will be beneficial to do a comparative study of the process models along with their respective set of metrics. The goal of this comparative study will be to analyze how different process models can accommodate the feedback from the software metrics in order to improve the process and the product. We might see that some process models will not be able to benefit much from some metrics due to their inherit characteristics at certain project phases.

CHAPTER 4

STUDY OF METRICS AND PROCESS MODELS

4.1 Metrics, Estimations and Risks

Metrics are ways of measuring the software processes. These metrics are in return used for software estimation. The metrics are used to measure the effort and the quality of the software. We can find the metrics from the different product artifacts.

Estimation is a process by which we try to judge the effort required in developing software. The better the estimates, less risks are expected to evolve in future.

The data that we obtain from software metrics is used for estimation. The estimation will be way off the mark if the metric data is not accurate. Examining the estimations can identify risks. Backtracking from the estimation to the metrics that were used to calculate those estimations can identify the problem areas that lead to risks. Those metrics can then be traced back to the artifacts that were used to find those metrics. This way we can identify the problem areas and make appropriate risk management strategies.

Two Layer Artifacts

We can differentiate the artifacts in two parts:

1. **Project Management Artifacts:** Created during the project life cycle to manage the project and the process.
2. **Product Artifacts:** Created at different stages of project life cycle to develop the product.

Characteristics of a Metric

We measure by discovering facts about our environment. These facts help the decision makers in understanding the context and making objective decisions.

Facts (metrics) should be available at any time during the project life cycle. Collecting metrics is costly. So it is necessary that only those metrics should be calculated which provide some value to the stakeholders.

Certain characteristics of a good metric:

1. It should provide some meaning to the stakeholders.
2. A correlation should exist between process changes and business performance.
3. It should be objective and unambiguously defined.
4. It should be able to display a trend over time.
5. It should be easily derivable from the existing workflows without introducing any overhead.
6. Automated tools can collect it.

There are two major ways in which metrics values can be utilized to improve the process and the product.

Firstly, if the metrics are providing alarming values then root cause analysis can be done to determine the problem areas in our project.

Secondly, by changing the values of the metrics we can see how the project will be affected. The amount of changes required will be shown through out the process model. We can also see if that much amount of change can be fitted in our project schedule and budget. This will help in making better decisions for improving product quality.

Based on these forecasts we can plan our future iterations in a phase. We will know which areas we need to concentrate on to bring our metric values to our desired value. If the desired value causes a lot of changes then we need to lower our desired values and make them more realistic.

To achieve this goal, we need to make sure that our artifacts and workflows are properly "connected". That is we can trace between any two dependent entities. This traceability will allow us to forecast the changes required in the process.

This forecast can be made on the basis of quality metrics and in-progress indicators. Not on the basis of primitive metrics.

To accomplish the above goal of predicting future iterations in a phase, we need to determine

- Relationship between metric and different artifacts produced in a phase.
- Relationships of different artifacts and their interdependencies.
- Relationship of different metrics to each other (how will changing one metric value affects other metric values)

After determining these relationships for RUP, we can easily adopt it for an agile process model also. Since agile can be considered as a lightweight version of RUP.

4.2 Evaluation of Software Metrics and Process Model

Waterfall Model

The Tables 4.1, 4.2, 4.3, 4.4, 4.5 and 4.6 shows the various phases in Waterfall Model and the metrics that can be derived at each phase. These tables are based on the process-metric evaluation framework explained in the above section.

Table 4.1 Project Initiation Phase of Waterfall Model

Phase		Project Initiation
Characteristics of the Software Process Model		• Project Plan is created with initial estimates of cost, size and time.
Product Artifacts		
Project Management Artifacts		Project Plan, Quality Assurance Plan
Product Metrics	Direct Metrics	
	Indirect Metrics	
Process Metrics	Direct Metrics	Estimated Duration, Estimated Cost, Estimated Size
	Indirect Metrics	Flesch-Kincaid Readability
Risks		Estimated duration and estimated cost are not close to real values.
Recommended Root Cause Analysis Approach		If the index of Flesch-Kincaid Readability is high for a Project Management Artifact, then it indicates that the technical documents are difficult to understand. It is recommended at this stage to revise the documents.

Flesch-Kincaid Readability [22] is used to test the readability of the technical documents. It helps in determining the level of difficulty of the technical document. In the Project Initiation Phase the Flesch-Kincaid Index is calculated from the Project Plan and the Quality Assurance Plan. If possible, the Flesch-Kincaid Index should be calculated from all the technical documents created in all the phases of the life cycle.

If the value of Flesch-Kincaid Index is high then this indicates a need for revision of the concerned documents.

Table 4.2 Requirements Phase of Waterfall Model

Phase		Requirements Phase
Characteristics of the Software Process Model		• Requirements are specified in accordance with the scope of the system stated in the Project Plan. • Requirements should be clearly and completely defined in this phase.
Product Artifacts		Requirements Specifications Document, Use case Document, Verification and Validation Plan, Data Flow Diagram
Project Management Artifacts		Risk Plan, Project Review Report
Product Metrics	Direct Metrics	
	Indirect Metrics	Bang, Specificity of Requirements, Completeness of Functional Requirements, Degree of Validation of Requirements, Flesch-Kincaid Readability
Process Metrics	Direct Metrics	Estimated Cost (with help of Bang), Estimated Duration, Estimated Resource Utilization
	Indirect Metrics	
Risks		• Requirements are incomplete • Requirements do not cover the functionality of the system. • Requirements are not validated (to be in alignment with scope). • Estimates are not close to the estimates done at the Project Initiation Phase
Recommended Root Cause Analysis Approach		• If the Bang Metric shows the project size to be larger than original estimation, then the Requirements Specifications Document along with Data Flow Diagram should be revisited. • If the requirements are incomplete, incorrect or invalidated then this phase should be repeated to avoid any problems in the future phases. It is recommended to identify and solve the problems earlier in the life cycle as they are expensive to solve in the later phases.

Bang Metric [21] estimates the size of the project. It uses the defined functionality of the project at a particular phase. In the Requirements phase, the Data Flow Diagram is used to derive the functionality of the system. The estimates resulting from Bang Metric are used to calculate an estimated cost for the project.

The estimates calculated in the Requirements phase can differ from the estimates calculated in the previous phase. If the newly calculated estimates of size, cost and duration are too high and undesirable, then the Project Plan needs to be modified to redefine the scope. This redefinition of scope can cause changes in the Requirements Specifications Document.

The *Specificity of Requirements* [21] helps in determining the level of ambiguity in the defined requirements. This metric has been defined as:

$$Q1=nui/nr$$

where,

Q1 is Specificity of Requirements,

nui is the number of requirements which have been identically interpreted,

and, nr is the sum of functional and non functional requirements.

The *Completeness of Functional Requirements* [21] helps in determining the level by which the functionality of the system has been covered by the Functional Requirements. It is defined as:

$$Q2=nu/(ni*ns)$$

where,

Q2 is Completeness of Functional Requirements,

nu is the number of Functional Requirements,

ni is the number of data inputs defined,

and, ns is the sum of various defined scenarios and system states.

The Degree of Validation of Requirements [21] helps in determining the level by which the requirements have been validated as being correct. It is defined as:

$$Q3' = nc/(nc+nnv)$$

where,

Q3' is the Degree of Validation of Requirements,

nc is the number of validated requirements,

and, nnv is the number of invalidated requirements.

Table 4.3 Design Phase of Waterfall Model

Phase		Design Phase
Characteristics of the Software Process Model		The design is based on the requirements stated in the previous phase.
Product Artifacts		Architecture Document, Design Document, Structure Charts, Class Diagrams, Data Dictionary, Pseudo Code, State Diagrams
Project Management Artifacts		Project Review Reports
Product Metrics	Direct Metrics	Fan-Out, Fan-In, Number of Classes, Methods per Class, Requirements Percent Coverage
	Indirect Metrics	Bang, Flesch-Kincaid Readability, Cyclomatic Complexity
Process Metrics	Direct Metrics	Estimated Duration, Estimated Cost, Estimated Resource Utilization
	Indirect Metrics	
Risks		• Complex design. • All requirements are not covered in design.
Recommended Root Cause Analysis Approach		• Design Complexity can be determined from the Product Metrics. If Fan In or Fan Out is high this indicates that the software interfaces need to be revised. • If the Requirements Percent Coverage is low then the Design needs to be revised in light of the Requirements Specifications Document. • If the estimated size and cost (from Bang) is larger than previously estimated, then alternate designs can be considered in the Design Phase. • Complexity of Design can be determined from the Cyclomatic Complexity Metric. If the metric's value is greater than a certain threshold than the particular module should be redesigned.

Fan In and *Fan Out* are derived from the Structure Charts. For a given module "M", Fan In represents the number of parent modules that call "M".

Fan Out represents the number of modules called by "M". A high value for Fan In and Fan Out can indicate a complicated design.

Number of Classes is a count of the classes defined in the design phase. A low value indicates presence of fewer classes which might be over loaded with a lot of diverse functions. In such a case, the design should be modified and new classes should be defined.

Methods per Class [30] is defined as:

Average number of methods per object class =

Total number of methods/Total number of object classes

[30] points out that higher the Methods per Class, higher are the chances of complicating the testing. But on the other hand, this high value can be desirable since it can lead to increased code reusability.

Requirements Percent Coverage can be defined as the ratio of the following:

Requirements Percent Coverage = Number of Functional Requirements defined in Requirements Phase/Number of Functional Requirements covered in Design Phase

Ideally the value of the ratio should be equal to 1. This will indicate that all the requirements have been covered in the design phase.

Cyclomatic Complexity [21] counts the number of paths from one point in the flow graph to another. Flow graph can be derived from the pseudo code written in the design phase. Cyclomatic Complexity is defined as:

$$v(G)=E-N+2P$$

where,

v(G) is Cyclomatic Complexity,

E is the number of edges,

N is the number of nodes,

where,

FP is the total number of adjusted functional points,

count total is the sum of all parameters (multiplied by their complexity value) listed above,

and, Fi is the complexity adjustment value found by the responses to the reliability related questions.

Lines of Code (LOC) [21] is a count of the number of non-commented and non-blank lines of code.

Table 4.5 Integration Testing Phase of Waterfall Model

Phase		Integration Testing Phase
Characteristics of the Software Process Model		Integration testing is performed.
Product Artifacts		Test plans, Installation & Delivery Plan
Project Management Artifacts		Review Reports, Test Reports, Quality Assurance Test Reports
Product Metrics	Direct Metrics	Number of Defects, Defects Status v/s Number of Defects, Average Execution Time
	Indirect Metrics	Maintainability, Reliability
Process Metrics	Direct Metrics	
	Indirect Metrics	
Risks		• Large number of defects is unfixed. • The overall performance of the system is lower than expected.
Recommended Root Cause Analysis Approach		• The Defect Status v/s Number of Defects indicates the number of defects that are not fixed. This can point out the defects that need to be resolved. • The Execution Time of the system can determine the performance of the system. If the execution time is greater than expected then the particular modules should be revisited. If the module cannot be changed to improve the execution time, then the design cannot be changed in Waterfall to improve the execution time.

Percentage of Unit Test Cases passed is the ratio of the following:

Percentage of Unit Test Cases passed=

Passed Unit Test Cases/ Total number of Test Cases

Low value of the above ratio indicates the need to modify the problem areas of the code which did not pass the unit tests.

Number of Defects [21] has been defined as:

$$\text{Potential Number of Defects} = FP^{1.25}$$

This value can be compared to the actual number of defects identified in the code.

Therefore, *Defects Identification Ratio is defined as:*

Defects Identification Ratio = Number of Defects/Actual Number of Discovered Defects

If this value is low then this indicates that some parts of the code might not have been tested thoroughly. To avoid potential problems in future phases, more thorough testing of the code should be performed at this stage.

Design Percent Coverage can be defined as follows:

Design Percent Coverage = Number of Classes present in the Code/ Number of Classes identified in the Design Phase

If the ratio is low, this indicates that some classes have not been implemented as yet.

Functional Points [21] determines the functionality of the system by considering the user inputs, user outputs, files and external interfaces. A complexity value is attached to the above mentioned parameters. A complexity adjustment value is calculated based on the responses to fourteen questions regarding reliability of the system.

FP is defined as:

$$FP = \text{count total} * [0.65 + 0.01 * \sum Fi]$$

and, P is the number of connected components.

To measure the design's reliability the suite of metrics given by Chidamber and Kemerer can be used.

Table 4.4 Code and Unit Testing Phase of Waterfall Model

Phase		Code and Unit Testing Phase
Characteristics of the Software Process Model		• The system is developed. • Unit testing is performed on the modules.
Product Artifacts		Code Modules, Documentation, Unit Test Plan
Project Management Artifacts		Development Plan, Project Review Report, Unit Test Report, Quality Assurance Test Report
Product Metrics	Direct Metrics	Effort/Module, Lines of Code
	Indirect Metrics	Percentage of Unit Test Cases passed, Defects Identification Ratio, Design Percent Coverage, Functional Points, Cyclomatic Complexity, Flesch-Kincaid Readability
Process Metrics	Direct Metrics	
	Indirect Metrics	
Risks		• The design is not covered completely in implementation. • The code modules have low quality. • The code is too complex to be understood. • All the defects have not been identified.
Recommended Root Cause Analysis Approach		• Design Percent Coverage metric can indicate the areas of design that are not covered by the code modules. Those areas can be revisited in the Implementation Phase. • Quality of the modules can be determined from the number of unit test cases that they pass. If the passed test cases are low then the code needs to be revised and adjusted. • Complexity of the code can be determined from the Cyclomatic Complexity Metric. • If the Defect Identification Ratio is low in value then this indicates the need to do more thorough testing of the code.

Number of Defects is the count of the number of defects identified in the Integration Testing Phase and the open defects from the previous phase. The defects need to be closed before the end of the Integration Testing Phase. If the defects are not closed then it will lead to problems in the future phases.

Defect Status v/s Number of Defects is a graph that represents Defect Status on the x-axis and Number of Defects on the y-axis. This graph will give a clear picture of open, closed and being worked on defects.

Average Execution Time is the average of the time taken by the system to perform various functions. This measure can be used to determine the performance of the system. This measure can also be represented by a graph. The difficulty of the function can be represented on x-axis and y-axis can represent the average execution time. This graph can be used to compare the desired execution time to the actual execution time. Large differences can indicate either unrealistic desired execution time or a poorly designed system. If the execution time of the system needs to be improved, then changes can only be made to the code and not to the design, due to the inherent nature of Waterfall Model.

Maintainability can be calculated from the 3 metric or 4 metric formulae given in [40]. The 3-metric maintainability equation is given as:

Maintainability Index = 171 - 5.2 x ln(aveV) - 0.23 x aveV(g') - 16.2 x ln(aveLOC)

where,

aveV is the average Halstead Volume per module,

aveV(g') is the average extended cyclomatic complexity per module,

and aveLOC is the average lines of code per module.

Halstead Volume has been given as the following in [41]:

V= N * (LOG2 n)

where,

V is the Halstead Volume,

N is defined as the sum of total number of operators and total number of operands,

n is defined as the sum of distinct operators and distinct operands.

Extended Cyclomatic Complexity has been defined in [42] to be "an extension of Cyclomatic Complexity that accounts for the added complexity resulting from compound conditional expressions. A compound conditional expression is defined as an expression composed of multiple conditions separated by a logical OR or AND condition. If an OR or an AND condition is used within a control construct, the level of complexity is increased by one for computation of the Extended Cyclomatic Complexity measure."

The 4-metric maintainability equation is given as:

Maintainability Index =
171 - 5.2 x ln(aveV) - 0.23 x aveV(g') - 16.2 x ln(aveLOC) + 50 x sin(sqrt(2.4 x perCM))

where,

aveE is the average Halstead Effort per module,

aveV(g') is the average extended cyclomatic complexity per module,

aveLOC is the average lines of code per module,

and perCM is the average percent of lines of comments per module.

Halstead Effort has been defined in [41] as:

E= D * V

where,

E is Halstead Effort,

D is the difficulty which is defined as:

D=(total number of operators/2)*(total number of operands/2)

and V is the Halstead Volume.

Table 4.6 Maintenance Phase of Waterfall Model

Phase		Maintenance Phase
Characteristics of the Software Process Model		
Product Artifacts		Manuals, Online Help
Project Management Artifacts		
Product Metrics	Direct Metrics	Flesch-Kincaid Readability
	Indirect Metrics	
Process Metrics	Direct Metrics	
	Indirect Metrics	
Risks		Too technical for the end user to understand.
Recommended Root Cause Analysis Approach		If the Flesch-Kincaid Readability metric is high then the User Manual and Help documents should be revised.

Rational Unified Process (RUP)

For RUP tables 4.7, 4.8, 4.9 and 4.10 highlight the different artifacts that are created during different phases. Based on these artifacts we derive the different metrics that can be calculated from them.

Table 4.7 Inception Phase of RUP

Phase		Inception
Characteristics of the Software Process Model		• Establish business case. • Define the project scope.
Product Artifacts		Vision Document, Requirements Document, Informal Design Document, Informal Test Document, Informal Architectural Document
Project Management Artifacts		Business Case Document, Software Development Plan, Status Assessment, Release Specifications, Risk Assessment Document
Product Metrics	Direct Metrics	Flesch-Kincaid Readability
	Indirect Metrics	
Process Metrics	Direct Metrics	Estimated Cost, Estimated Schedule, Resource Estimation, Flesch-Kincaid Readability
	Indirect Metrics	Duration of Iterations, Percentage of Work Break Down, Percentage of Tasks assigned to people/team
Risks		The project scope is too big.
Recommended Root Cause Analysis Approach		Analyze the Vision Document, Requirements Document and Business Case Document to define the project scope.

Duration of Iterations is the allocated time given to each iteration of every phase. This can help in determining the size of the project. This measure can be plotted on a graph, with iterations represented on x-axis and time represented on y-axis.

Percentage of Work Break Down is the work breakdown per iteration. This helps in determining how work is distributed across the iterations. Percentage of Work Break Down should be studied along with the Duration of Iterations metric. If iteration has a large number of work assigned to it, but the duration of the iteration is small, then this

needs to be investigated. Either a large number of resources are going to be working in that iteration, or the work needs to be distributed in other iterations, or the duration of the iteration needs to be increased.

Table 4.8 Elaboration Phase of RUP

Phase		Elaboration
Characteristics of the Software Process Model		• Finalize the architecture, requirements and project plan. • Resolve the highest level risks.
Product Artifacts		Modified Vision Document, Modified Requirements Document, Modified Design Document, Modified Architecture Document, Initial Source Code, Initial Integrated Executables, Informal User Manual
Project Management Artifacts		Modified Business Case Document, Modified Release Specifications, Modified Software Development Plan, Informal Deployment Documents, Modified Risk Assessment Document
Product Metrics	Direct Metrics	Methods per Class, Inheritance Dependencies, Weighted Methods per Class (WMC), Number of Defects
	Indirect Metrics	Bang, Specificity of Requirements, Completeness of Functional Requirements, Degree of Validation of Requirements, Flesch-Kincaid Readability, Requirements Coverage by Design, Design Coverage by Test Cases, Number v/s Types of Class Relations, Feature Creep Percentage, Defects Status v/s Number of Defects
Process Metrics	Direct Metrics	
	Indirect Metrics	
Risks		• All requirements have not been completely captured. • Highest level risks are not eliminated. • All requirements are not captured in the design. • Test cases are not written for all the classes.

Table 4.8 (continued) Elaboration Phase of RUP

Recommended Root Cause Analysis Approach	• Completeness of Functional Requirements metric, Specificity of Requirements and Degree of Validation of Requirements can point to any inconsistencies with the requirements. • The Risk Assessment Document can show the status of risks. If any high level risk is unresolved then it should be considered. • If Requirements Covered By Design metric is closer to 1 then this indicates that majority of the requirements have been covered in design. The left over requirements can be taken care of in the next iteration of the Elaboration Phase. • If Design Coverage by Test Cases metric is closer to 1 then this indicates that test cases have been created for majority of classes. The left over classes can be considered in the next iteration of the Elaboration Phase.

Design complexity can be determined from the set of following metrics:

Methods per Class [30] is defined as:

Average number of methods per object class = Total number of methods/Total number of object classes

Inheritance Dependencies [30] is defined as:

Inheritance tree depth = max (inheritance tree path length)

The higher value for Inheritance Dependencies indicates greater chances of reusability of the object classes. On the other hand, this high value can be a problem in the testing phase.

Weighted Methods per Class (WMC) [30] is defined as the sum of complexities of all the methods in a class.

While, *Average Method Complexity* [30] is defined as:

Average Method Complexity = Sum of the cyclomatic complexity of all Methods/Total number of application methods

WMC can be a good indicator of the level of effort and time required for development of a class.

Number v/s Types of Class Relations is a graph with types of class relations plotted on x-axis and number of relations on y-axis. If the graph shows a large number of public classes then this should be checked in the next iteration.

For each iteration in Elaboration phase we can calculate *Requirements Covered by Design.* It is the ratio of the:

Requirements Covered By Design = Requirements captured in Object Classes/Total Number of Functional Requirements.

As the iterations progress, the ratio should get closer to 1.

For each iteration in Elaboration phase we can calculate Design *Covered by Test Cases.* It is the ratio of the:

Design Coverage By Test Cases (for a class) = Number of Test Cases for a Class/ Total number of functions in the class

As the iterations progress, new requirements might be added. To keep a check on new requirements we can use the metric, *Feature Creep Percentage* which is defined as:

Feature Creep Percentage = New Functional Requirements Introduced in Current Iteration/Total Functional Requirements of the System

Feature Creep Percentage is expected to decrease in each future iteration. If it does not decrease and keeps on increasing then checks need to be made about what is causing the increase in the requirements. Since this can cause the scope of the project to expand and the project end date to slip.

Number of Defects has been previously defined. With each iteration the value for this metric is expected to decrease. Besides this *Defects Status v/s Number of Defects* can be plotted and monitored over the iterations.

Table 4.9 Construction Phase of RUP

Phase		Construction
Characteristics of the Software Process Model		• All the modules are developed. • Unit Tests are performed.
Product Artifacts		Modules, User Manuals, Documentation
Project Management Artifacts		
Product Metrics	Direct Metrics	Number of use cases implemented, Number v/s Type of Defects, LOC,
	Indirect Metrics	Cyclomatic Complexity Metric, Scrap Ratio, Rework Ratio, Modularity, Adaptability, Maturity, Maintainability, Rework Stability, Rework Backlog, Modularity Trend, Adaptability Trend, Maturity Trend
Process Metrics	Direct Metrics	Defects Status v/s Number of Defects Defect Introduction and Identification Chart per Phase, Mean Time Between Failure (MTBF)
	Indirect Metrics	Average effort to fix a defect, Average Effort to write a Class
Risks		• The code is highly complex. • The system crashes quite frequently. • High level of effort is spent on rework. • The system architecture is degrading with time.
Recommended Root Cause Analysis Approach		• Complexity of the code is determined from the Cyclomatic Complexity Metric. • If MTBF is low then this shows that code is not reliable. Defects need to be identified and removed. Maturity Trend also shows the maturity level of the system. Low value indicates less reliable system. • Rework Ratio (and Scrap Ratio) can show the comparison between rework and total effort. If the ratio is too large then this indicates either incorrect requirements/faulty design/bad coding. The level of effort spent to make a change can be estimated from Adaptability Trend. • Modularity Trend can show the trend of changes being made to the system. If the changes are on a high even at the end of the project, then this indicates degradation of architecture.

Scrap Ratio [1] can be used at the end of the Construction phase to find out the quality of the developed system. The Scrap Ratio is defined as the percentage of the product that has to be reworked during the life cycle. For formula of Scrap Ratio refer to Table 1.3.

Rework Ratio [1] can be used as an end product quality indicator. Rework Ratio is defined as the percentage of the effort spent on rework as compared to the total effort on the system. For formula of Rework Ratio refer to Table 1.3.

Modularity [1] calculates the average breakage over time. For formula of Modularity refer to Table 1.3. *Modularity Trend* can be found by plotting Modularity over a period of time.

Adaptability [1] is defined as the trend of rework over time. Less number of changes over time indicates a good quality system. For formula of Adaptability refer to Table 1.3. *Adaptability Trend* can be found by plotting Adaptability over a period of time.

Maturity [1] is defined as the Mean Time Between Failure (MTBF) trend over time or defect rate over time. For formula of Maturity refer to Table 1.3. *Maturity Trend* can be found by plotting Adaptability over a period of time.

Maintainability [1] identifies relationship between maintenance cost and development cost. For formula of Maintainability refer to Table 1.3.

Rework Stability [1] identifies the difference between total rework and closed rework. For formula of Rework Stability refer to Table 1.4.

Rework Backlog [1] is the percentage of the current product baseline that needs to be repaired. For formula of Rework Backlog refer to Table 1.4.

Average Effort to fix a defect can be defined as:

Average Effort to fix a defect (for an iteration) = Time consumed to fix the defects in an iteration/Total number of defects fixed in an iteration

Average Effort to write a Class can be defined as:

Average Effort to write a Class (for an iteration) = Time consumed to code a Class in an iteration /Total number of Classes coded in an iteration

The above two metrics can also be calculated for the whole system rather than an iteration.

Table 4.10 Transition Phase of RUP

Phase		Transition
Characteristics of the Software Process Model		• The system is delivered to the end user. A beta version can be delivered to assure user satisfaction. • Training of users.
Product Artifacts		
Project Management Artifacts		User Surveys, Estimate v/s Planned Cost
Product Metrics	Direct Metrics	
	Indirect Metrics	
Process Metrics	Direct Metrics	
	Indirect Metrics	
Risks		• Users are not satisfied. • The actual project cost is crossing the planned cost.
Recommended Root Cause Analysis Approach		• User satisfaction can be determined by the user surveys. If fewer users are satisfied then the problem areas should be addressed in the next version of the system. • If actual cost is higher than planned cost then either the cost can be covered in the next version of the project (by cutting down the features).

Scrum

Tables 4.11, 4.12, 4.13 and 4.14 highlight the artifacts and the metrics derived at each phase in Scrum. Some of the artifacts are mentioned in the following tables which might be developed depending on the nature of the project. For example, an informal design and architecture document can be made rather than a formal design document. Similarly, the informal project plan can hold the risk management plan also.

Table 4.11 Project Planning Phase of Scrum

Phase		Project Planning
Characteristics of the Software Process Model		• Requirements are gathered. • Analysis is performed. • Decide on which release can be developed first. • Packet of requirements from Product Backlog is created for the upcoming release.
Product Artifacts		Product Backlog
Project Management Artifacts		Informal Project Plan, Informal Risk Management Plan
Product Metrics	Direct Metrics	
	Indirect Metrics	
Process Metrics	Direct Metrics	Estimated Duration, Estimated Cost and Resource Utilization for a release
	Indirect Metrics	
Risks		Requirements creep may occur.
Recommended Root Cause Analysis Approach		• If the duration and cost are unacceptable then changes should be made to the Product Backlog to reduce scope, hence reducing the duration and cost of the project.

Project Plan defines the delivery date, estimated cost, duration, resource utilization for the starting Sprint (release), functionality for one or more releases and selection of the tool and definition of the infrastructure.

Risks and process is managed by the different controls employed in SCRUM.

Requirements are expected to change in SCRUM but the applied controls should keep a check that these changes are monitored. Risks are evaluated at planning stage by reviewing Product Backlog. If some requirements seem infeasible then they are adjusted in the Product Backlog. Infeasibility of the requirements cannot be identified with the help of metrics. This has to be determined by the SCRUM Master or the project manager.

Table 4.12 Design Phase of Scrum

Phase		Design
Characteristics of the Software Process Model		High level design is created based on the Product Backlog.
Product Artifacts		Revised Product Backlog, Design Document, Architecture Document
Project Management Artifacts		
Product Metrics	Direct Metrics	
	Indirect Metrics	
Process Metrics	Direct Metrics	
	Indirect Metrics	
Risks		
Recommended Root Cause Analysis Approach		

Table 4.13 Sprint of Scrum

Phase	Sprint
Characteristics of the Software Process Model	• Development is done with respect to "the variables of time, requirements, quality, cost and competition". [24] • Sprint Planning Meeting is held to with the client and the team to decide the new set of requirements for this release and to prioritize work. Discussion about "existing product, business and technology conditions" [37] is held. Sprint Backlog and Sprint Goal is created. Product Backlog might be updated with addition/deletion of requirements. • Daily Scrums are held during the Sprint to monitor Sprint progress.

Table 4.13 (continued) Sprint of Scrum

Characteristics of the Software Process Model		• Sprint Review Meeting is held to show and discuss the Sprint Deliverable. • Unit testing and Release Testing are performed in the Sprint.
Product Artifacts		Sprint Backlog, Revised Product Backlog, Blocks List , Sprint Deliverable
Project Management Artifacts		Revised Project Plan, Sprint Goal
Product Metrics	Direct Metrics	
	Indirect Metrics	
Process Metrics	Direct Metrics	Release Burn-down Chart[38], End Date Predictor Chart[38]
	Indirect Metrics	
Risks		• Contradictory Requirements might occur. • All requirements in Sprint Backlog might not be implemented in the Sprint Cycle. • Requirements creep may occur.
Recommended Root Cause Analysis Approach		• Contradictory Requirements can be discovered in Daily Scrums or Sprint Planning Meeting. In such a case Sprint Backlog, Product Backlog should be revised by removing the contradictory requirements. If the changes in Product Backlog can not be made immediately then the issue should be noted in the Blocks List. • By predicting the end date of the Sprint, team can see if all the Sprint Backlog will be covered in the current Sprint or not. If the predicted end date does not match the original planned end date then this means that some requirements from Sprint Backlog have to be left for the future Sprints. • The release progress can be monitored from Release Burn-down Chart. If the Release Burn-down Chart shows the same number of requirements for each Sprint then this indicates no or low progress. In such a case, review the Sprint Backlog to identify the set of requirements which are causing problems during development. Check the Blocks List for any reported issues with these requirements.

Direct observations are highly important in controlling the Sprint. Direct observations can point out any deficiencies in the team skills, team collaboration and slipping end dates. Release Burn-down Chart [38] is used to measure the "net change in the amount of work remaining" [38]. The Release Burn-down Chart takes into consideration the development progress, new requirements and the removed requirements.

By using the Release Burn-down Chart the End Date Predictor Chart [38] can be easily obtained. End Date Predictor Chart shows "the number of sprints a project will take" [38] to finish. Both these charts can help in identifying any slippages that might have occurred. They can also serve as good progress indicators.

Table 4.14 Project Closure of Scrum

Phase		Project Closure
Characteristics of the Software Process Model		System Testing is conducted. Documentation is completed before deploying the system.
Product Artifacts		Documentation
Project Management Artifacts		
Product Metrics	Direct Metrics	
	Indirect Metrics	
Process Metrics	Direct Metrics	
	Indirect Metrics	
Risks		
Recommended Root Cause Analysis Approach		

Extreme Programming

Tables 4.15, 4.16, 4.17 and 4.18 highlight the artifacts and the metrics derived at each phase in Extreme Programming. The software metrics are taken from [9],[10],[11] and [12].

Table 4.15 Exploration Phase of Extreme Programming

Phase		Exploration
Characteristics of the Software Process Model		• Customer helps in writing stories which describe the systems requirements. • Developers can test the technology and proposed solutions by creating prototypes.
Product Artifacts		User Stories, Prototypes
Project Management Artifacts		
Product Metrics	Direct Metrics	Number of User Stories
	Indirect Metrics	
Process Metrics	Direct Metrics	
	Indirect Metrics	
Risks		Problem can be high in complexity. This is dealt by doing prototyping.
Recommended Root Cause Analysis Approach		

Table 4.16 Planning Phase of Extreme Programming

Phase		Planning
Characteristics of the Software Process Model		• Based on the User Stories, time estimates are calculated. • Decision on which user stories will be developed initially is made.
Product Artifacts		
Project Management Artifacts		Project Plan
Product Metrics	Direct Metrics	
	Indirect Metrics	
Process Metrics	Direct Metrics	Estimated Duration, Estimated Cost
	Indirect Metrics	
Risks		All the User Stories might not be implemented in the desired duration and cost.
Recommended Root Cause Analysis Approach		The Project Plan helps in estimating the number of iterations and releases that are required. This gives an estimate of the duration and cost. If this estimate does not match the desired values, then User Stories can be cut down by working with the Customer.

Table 4.17 Development Iteration of Extreme Programming

Phase		Development Iteration
Characteristics of the Software Process Model		• Iterative development continues until the system is completely developed. • User stories selected for that iteration are expanded into specific tasks. • Developers choose which tasks they prefer to do. • Developers provide estimates based on the tasks they have chosen. • Task can be reassigned to manage work load between developers. • Functional Test cases and Automated Test Cases are created. • Code is written by help of test cases. • Changes can be made to User Stories or new ones can be added.
Product Artifacts		Code, Functional Test Cases, Automated Test Classes, New or Changed User Stories
Project Management Artifacts		
Product Metrics	Direct Metrics	Number of Defects
	Indirect Metrics	Productivity (User Stories/Person-Month, Relative KLOC/Person-Month, Size/Effort, Velocity/Effort), Project Velocity, Number of Automated Test Class per User Story
Process Metrics	Direct Metrics	Release Length, Iteration Length
	Indirect Metrics	Pairing Frequency, Effort spent fixing defects, Relative Schedule Deviation, Relative Cost Deviation, Customer and Developer Satisfaction
Risks		All the user stories have not been covered by the test cases.
Recommended Root Cause Analysis Approach		The Test coverage metric informs about the number of user stories covered.

Pairing Frequency [33] can be defined as:

Pairing Frequency = (Number of Pairs/ (Total Number of Developers/2))*100

The number of pairs can be found by "examining the file headers" [33].

Release Length [33] is the time taken to develop a release.

Iteration Length [33] is the time taken in an iteration.

Productivity can be measured as *Size/Effort* [34]. This can be calculated for a team, a pair or for each programmer. This metric can indicate the areas of low productivity. Those problem pairs or individuals can be investigated further to identify the root cause of low productivity.

Effort Spent Fixing Defects [34] is defined as:

Effort Spent Fixing Defects = (Effort spent for bug fixing/Effort)*100

This can be calculated for the team or an individual across an iteration or the whole project [34].

Relative Schedule Deviation [34] is defined as:

Relative Schedule Deviation = ((Real time – Planned time)/Planned time)*100

This deviation can be calculated for the whole project, iteration, release, or for a specific user story [34].

Relative Cost Deviation [34] is defined as:

Relative Cost Deviation = ((Real costs – Planned costs)/Planned costs)*100

This deviation can be calculated for the whole project, iteration, release, or for a specific user story [34].

Customer/developer Satisfaction [34] is calculated from the questionnaires. This is rated on the scale of 0-100%. This metric can be measured for the whole project, iteration or release [34].

Number of Automated Test Class per User Story [35] is "a count of the automated test classes that test the functionality" [35].

Project Velocity [36] tracks the completeness of the project on basis of the number of functional test cases passed.

CHAPTER 5

CONTRIBUTIONS OF THE THESIS

The thesis starts with a synopsis of the software metrics and the process models. A literature overview is presented about the process models, software metrics and importance of software metrics in making better project management decisions.

In an effort to define a relationship between a process model and a set of software metrics, the author has defined a Process - Metric Evaluation Framework. The framework suggests that a process model should be studied along with various artifacts that are generated in each phase. These artifacts can then help in retrieving a set of metrics suitable for a specific process model.

Based on this framework the Process – Metric Evaluation Template has been created. This template served as a basic tool in evaluating process models with respect to t possible relationships between various artifacts and suitable software metrics. The template goes further and suggests the use of recommended root cause analysis approach at various points in a process model. The recommended root cause analysis approach can help in tracking and reducing risks that can be assessed by the artifacts and the software metrics.

The author has applied the Process – Metric Evaluation Template to various process models. The existing software metrics have been mapped to the artifacts generated in each phase of the process model. In some instances the author has suggested additional well suited software metrics that might be beneficial for the process model.

REFERENCES

1. Royce, W. (1998). *Software Project Management-A Unified Framework.* Addison-Wesley.

2. Mills, E. E. (1998, December). *Software Metrics.* Curriculum Module SEI-CM-12-1.1. Software Engineering Institute, Carnegie Mellon University.

3. *Software Metrics.* Retrieved May 15, 2004, from University of Antwerp Web site: http://www.lore.ua.ac.be/Teaching/SE1LIC/11Metrics.pdf.

4. Wiegers, K.E. (1999, July). A Software Metrics Primer. *Software Development.*

5. Peters, K. (1999). *Software Project Estimation.* Retrieved January 9, 2005, from Software Productivity Center Inc. Web site: http://www.software-engineer.org.

6. Lewis, J.P. (2001). Large Limits to Software Estimation. *ACM Software Engineering Notes*, 26(4), 54-59.

7. Ozcarn B. and Fathi, Y. (2000). *Software Metrics and Measurements.* Retrieved April 15, 2004, from Web site: http://www.questcon.com/home.nsf/.

8. Giles, A.E. and Barney, D. (1995, June). Metric Tools: Software Cost Estimation. *CrossTalk, The Journal of Defense Software Engineering.*

9. Jorgensen, M., Kirkeboen, G., Sjoberg, D., Anda, B. and Bratthall, L. (2000, June 5). Human Judgement in Effort Estimation of Software Projects. *International Conference on Software Engineering*, 45-51.

10. Fenton, N. and Neil, M. (2000, June 4-11). Software Metrics: Roadmap. *Proceedings of the conference on The future of Software engineering*, 357-370.

11. Fairley, R. E. (1992). Recent Advances in Software Estimation Techniques. *International Conference on Software Engineering*, 382-391.

12. Basili, V.R. (1990). Recent Advances in Software Measurement. *International Conference on Software Engineering*, 44-49.

13. Boehm. B. and Ross. R. (1988). Theory-W Software Project Management: A Case Study. *International Conference on Software Engineering*, 30-40.

14. Neal, R. D. *The Measurement of Reusable Software Artifacts*. Retrieved January 9, 2005, from The University of Maine Web site: http://www.umcs.maine.edu/.

15. Boehm, B., Egyed, A., Kwan, J., Port, D., Shah, A. and Madachy, R. (1998). Using the WinWin Spiral Model: A Case Study. *Computer*, 31(7), 33-44.

16. Baik, J., Boehm, B. and Steece, B. M. (2002, November). Disaggregating and Calibrating the CASE Tool Variable in COCOMO II. *IEEE Transactions on Software Engineering*, 28(11), 1009-1022.

17. Beck, K. and Boehm, B. (2003, June). Agility through Discipline: A Debate. *Computer*, 36(6), 44-46.

18. Boehm, B. and Turner, R. (2003, June). Using Risk to Balance Agile and Plan-Driven Methods. *Computer*, 36(6), 57-66.

19. Boehm, B. and Turner, R. (2003, June 25-28). Observations on Balancing Discipline and Agility. *Proceedings of the Conference on Agile Development*, 32.

20. Rational Software. *Rational Unified Process – Best Practices for Software Development Teams*. Retrieved May 15, 2004 from IBM Web site: http://www.rational.com/sitewide/support/whitepaper.

21. Hilda, B. K. (2003, May). *A Study of Software Metrics*. Masters Thesis.

22. Fernando, B. A. and Rogério, C. (1994). Candidate metrics for object-oriented software within a taxonomy framework. *Journal of Systems and Software*, 26(1), 87-96.

23. Boehm, B. (1988, May). A Spiral Model of Software Development and Enhancement. *Computer*, 21(5), 61-72.

24. Schwaber, K. (1995, October). SCRUM Development Process. Workshop Report: Business Object Design and Implementation. 10th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications. Addendum to the Proceedings. ACM/SIGPLAN OOPS Messenger, 6(4).

25. Beck, K. (1999, October). Embracing Change with Extreme Programming. *Computer*, 32(10), 70-77.

26. Reis, C. (2000). *A commentary on the Extreme Programming development process*. Retrieved December 20, 2004, from Web site: http://www.async.com.br/~kiko/papers/xp.ps.

27. Paulk, N.C. (2001, July). Extreme programming from a CMM perspective. *Software*, 18(6), 19-26.

28. Wiegers, K.E. *Software Metrics: Ten Traps to Avoid.* Retrieved December 20, 2004, from Process Impact Web site: http://www.processimpact.com/articles/mtraps.html.

29. Lamb. D. A. and Abounader, J. R. (1997). *Data Model for Object-Oriented Design Metrics*. Retrieved December 20, 2004, from Queen's University Web site: http://www-lips.ece.utexas.edu/~graser/pubs/proposal/proposal.pdf.

30. Sultanoglu. S. *Object Oriented Metrics*. Retrieved December 20, 2004, from Web site: http://yunus.hun.edu.tr/~sencer/oom.html.

31. Xenos, M., Stavrinoudis, D., Zikouli, K. and Christodoulakis, D. (2000). Object-Oriented Metrics - A Survey. *Proceedings of the FESMA 2000, Federation of European Software Measurement Associations.*

32. Boehm, B. and Hansen, W. (2001). The Spiral Model as a Tool for Evolutionary Acquisition, *CrossTalk, The Journal of Defense Software Engineering.*

33. Williams, L., Krebs, W., Layman, L., Anton, A. I. and Abrahamsson, P. *Toward a Framework for Evaluating Extreme Programming*. Retrieved January 9, 2005, from Web site: http://www. collaboration.csc.ncsu.edu/laurie/Papers/ease.pdf.

34. Ilieva, S., Ivanov, P. and Stefanova, E. (2004, August 31 – September 3). Analyses of an Agile Methodology Implementation. *Proceedings of the 30th EUROMICRO Conference (EUROMICRO'04)*, 326-333.

35. Williams, L., Layman, L. and Krebs. W. *Extreme programming evaluation framework for object oriented languages.* Retrieved January 9, 2005 from Web site: ftp://ftp.ncsu.edu/pub/unity/lockers/ftp/csc_anon/tech/2004/TR-2004-18.pdf.

36. Williams, L. and Upchurch, R. *Extreme programming for software engineering education?.* Retrieved January 9, 2005 from Web site: http://collaboration.csc.ncsu.edu/laurie/Papers/FIE_01.pdf.

37. Wake, W. C. (2004, January). *Scrum Development Process.* Retrieved January 9, 2005, from Web site: http://xp123.com/xplor/xp0401/Scrum-dev.pdf.

38. Mountain Goat Software. *An Alternative Release Burn down Chart.* Retrieved January 9, 2005 from Web site: http://www.mountaingoatsoftware.com/scrum/burndown.php.

39. Kitchenham, B. A. and Walker, J. G. (1989, January). A quantitative approach to monitoring software development. *Software Engineering Journal*, 4(1), 2-13.

40. Welker, K. D. and Oman, P. W. (1995, November –December). Software Maintainability Metrics Models in Practice. *CrossTalk, The Journal of Defense Software Engineering.*

41. Software Engineering Institute. (2004). *Halstead Complexity Measures – Software Technology Roadmap.* Retrieved January 10, 2005, from Carnegie Melon Web site: http://www.sei.cmu.edu/str/descriptions/halstead_body.html.

42. Certified Software Solutions. (2004). ProMet Code Metrics – Certified Software Solutions. Retrieved January 10, 2005, from Certified Software Solutions Web site: http://www.certifiedsoftware.com/products/promet.htm.

43. The Standish Group. (2005). *The Chaos Report.* Retrieved January 13, 2005, from The Standish Group Web site: http://www.standishgroup.com/sample_research/index.php.