**4-1 Measures, Metrics, and Indicators**

Although the terms measure, measurement, and metrics are often used interchange-ably, it is important to note the subtle differences between them. Within the software engineering context, a measure provides a quantitative indication of the extent, amount, dimension, capacity, or size of some attribute of a product or process. Measurement is the act of determining a measure.

When a single data point has been collected (e.g., the number of errors uncovered in the review of a single module), a measure has been established.

Measurement occurs as the result of the collection of one or more data points (e.g., a number of module reviews are investigated to collect measures of the number of errors for each).

Software metric relates the individual measures in some way (e.g., the average number of errors found per review or the average number of errors found per per-son-hour expended on reviews.

A software engineer collects measures and develops metrics so that indicators will be obtained. An indicator is a metric or combination of metrics that provide insight into the software process, a software project, or the product itself.

**4-2 Metrics in the Process and Project Domains**

Measurement is commonplace in the engineering world. We measure power consumption, weight, physical dimensions, temperature, voltage; signal-to-noise ratio . . . the list is almost endless. Unfortunately, measurement is far less common in the soft-ware engineering world. We have trouble agreeing on what to measure and trouble evaluating measures that are collected.

Process metrics are collected across all projects and over long periods of time. Their intent is to provide indicators that lead to long-term software process improvement. Project indicators enable a software project manager to (1) assess the status of an ongoing project, (2) track potential risks, (3) uncover problem areas before they go “critical,” (4) adjust work ﬂow or tasks, and (5) evaluate the project team’s ability to control quality of software work products.

In some cases, the same software metrics can be used to determine project and then process indicators. In fact, measures that are collected by a project team and converted into metrics for use during a project can also be transmitted to those with responsibility for software process improvement. For this reason, many of the same metrics are used in both the process and project domain.

**4-2-1 Process Metrics**

The only rational way to improve any process is to measure speciﬁc attributes of the process, develop a set of meaningful metrics based on these attributes, and then use the metrics to provide indicators that will lead to a strategy for improvement. But before we discuss software metrics and their impact on software process improvement, it is important to note that process is only one of a number of “controllable factors in improving software quality and organizational performance

Referring to Figure 4.1, process sits at the center of a triangle connecting three factors that have a profound inﬂuence on software quality and organizational performance.

1-The skill and motivation of people has been shown to be the single most inﬂuential factor in quality and performance.

2- The complexity of the product can have a substantial impact on quality and team performance.

3-The technology (i.e., the software engineering methods) that populates the process also has an impact.

In addition, the process triangle exists within a circle of environmental conditions that include:

4- The development environment (e.g., CASE tools).

5- Business conditions (e.g., deadlines, business rules).

6-and customer characteristics (e.g., ease of communication).



Figure 4-1 Determinants for software quality and organizational effectiveness

**4-2-2 Project Metrics**

 Software process metrics are used for strategic purposes. Software project measures are tactical. That is, project metrics and the indicators derived from them are used by a project manager and a software team to adapt project work ﬂow and technical activities.

 The ﬁrst application of project metrics on most software projects occurs during estimation. Metrics collected from past projects are used as a basis from which effort and time estimates are made for current software work. As a project proceeds, measures of effort and calendar time expended are compared to original estimates (and the project schedule). The project manager uses these data to monitor and control progress.

 As quality improves, defects are minimized, and as the defect count goes down, the amount of rework required during the project is also reduced. This leads to a reduction in overall project cost.

 A model of software project metrics suggests that every project should measure:

• **Inputs**: measures of the resources (e.g., people, environment) required to do the work.

• **Outputs**: measures of the deliverables or work products created during the software engineering process.

• **Results**: measures that indicate the effectiveness of the deliverables.

**4-3 Software Measurement**

 Measurements in the physical world can be categorized in two ways: direct measures (e.g., the length of a bolt) and indirect measures (e.g., the "quality" of bolts produced, measured by counting rejects). Software metrics can be categorized similarly.

 Direct measures of the software engineering process include cost and effort applied. Direct measures of the product include lines of code (LOC) produced, execution speed, memory size, and defects reported over some set period of time. Indirect measures of the product include functionality, quality, complexity, efﬁciency, reliability, maintainability, and many other "–abilities".

 The cost and effort required building software, the number of lines of code produced, and other direct measures are relatively easy to collect, as long as specific conventions for measurement are established in advance. However, the quality and functionality of software or its efﬁciency or maintainability are more difﬁcult to assess and can be measured only indirectly.

 Consider a simple example. Individuals on two different project teams record and categorize all errors that they ﬁnd during the software process. Individual measures are then combined to develop team measures. Team A found 342 errors during the software process prior to release. Team B found 184 errors. All other things being equal, which team is more effective in uncovering errors throughout the process? Because we do not know the size or complexity of the projects, we cannot answer this question. However, if the measures are normalized, it is possible to create software metrics that enable comparison to broader organizational averages.

**4-3-1 Size-Oriented Metrics**

 Size-oriented software metrics are derived by normalizing quality and/or productivity measures by considering the size of the software that has been produced. If a soft-ware organization maintains simple records, a table of size-oriented measures, such as the one shown in Figure 4.2, can be created. The table lists each software development project that has been completed over the past few years and corresponding measures for that project.



Figure 4-2 Size-oriented metrics

Referring to the table entry (Figure 4.4) for project alpha: 12,100 lines of code were developed with 24 person-months of effort at a cost of $168,000. It should be noted that the effort and cost recorded in the table represent all software engineering activities (analysis, design, code, and test), not just coding. Further information for project alpha indicates that 365 pages of documentation were developed, 134 errors were recorded before the software was released and 29 defects were encountered after release to the customer within the first year of operation. Three people worked on the development of software for project alpha.

In order to develop metrics that can be assimilated with similar metrics from other projects, we choose lines of code as our normalization value. From the rudimentary data contained in the table, a set of simple size-oriented metrics can be developed for each project:

• Errors per KLOC (thousand lines of code).

• Defects4 per KLOC.

• $ per LOC.

• Page of documentation per KLOC.

In addition, other interesting metrics can be computed:

• Errors per person-month.

• LOC per person-month.

• $ per page of documentation.

**4-3-2 Function-Oriented Metrics**

Function-oriented software metrics use a measure of the functionality delivered by the application as a normalization value. Since ‘functionality’ cannot be measured directly, it must be derived indirectly using other direct measures. Function-oriented metrics suggested a measure called the function point.

Function points are computed by completing the table shown in Figure 4.3.

Figure 4-3 computing function points

 Five information domain characteristics are determined and counts are provided in the appropriate table location. Information domain values are deﬁned in the following manner:

* Number of user inputs.
* Number of user outputs.
* Number of user inquiries.
* Number of ﬁles.
* Number of external interfaces.

 Once these data have been collected, a complexity value is associated with each count. Organizations that use function point methods develop criteria for determining whether a particular entry is simple, average, or complex.

To compute function points (FP), the following relationship is used:



Where count total is the sum of all FP entries obtained from Figure 4.3.

The Fi (i = 1 to 14) are "complexity adjustment values" based on responses to the following questions [ART85]:

1. Does the system require reliable backup and recovery?

2. Are data communications required?

3. Are there distributed processing functions?

4. Is performance critical?

5. Will the system run in an existing, heavily utilized operational environment?

6. Does the system require on-line data entry?

7. Does the on-line data entry require the input transaction to be built over multiple screens or operations?

8. Are the master ﬁles updated on-line?

9. Are the inputs, outputs, ﬁles, or inquiries complex?

10. Is the internal processing complex?

11. Is the code designed to be reusable?

12. Are conversion and installation included in the design?

13. Is the system designed for multiple installations in different organizations?

14. Is the application designed to facilitate change and ease of use by the user?

 Each of these questions is answered using a scale that ranges from 0 (not important or applicable) to 5 (absolutely essential).

 Once function points have been calculated, they are used in a manner analogous to LOC as a way to normalize measures for software productivity, quality, and other attributes:

• Errors per FP.

• Defects per FP.

• $ per FP.

• Pages of documentation per FP.

• FP per person-month.

**4-4 Reconciling Different Metrics Approaches**

 The relationship between lines of code and function points depend upon the programming language that is used to implement the software and the quality of the design. A number of studies have attempted to relate FP and LOC measures.

 The following table provides rough estimates of the average number of lines of code required to build one function point in various programming languages:



**4-5 Metrics for Software Quality**

 The overriding goal of software engineering is to produce a high-quality system, application, or product. To achieve this goal, software engineers must apply effective methods coupled with modern tools within the context of a mature software process. In addition, a good software engineer (and good software engineering managers) must measure if high quality is to be realized.

**4-5-1 an Overview of Factors That Affect Quality**

 McCall and Cavano deﬁned a set of quality factors that were a ﬁrst step toward the development of metrics for software quality. These factors assess software from three distinct points of view:

 (1) Product operation (using it)

 (2) Product revision (changing it)

 (3) Product transition (modifying it to work in a different environment; i.e., “porting" it).

**4-5-2 Measuring Quality**

 There are many measures of software quality, correctness; maintainability, integrity, and usability provide useful indicators for the project team.

* **Correctness**: A program must operate correctly or it provides little value to its users. The most common measure for correctness is defects per KLOC. When considering the overall quality of a software product, defects are those problems reported by a user of the program after the program has been released for general use. For quality assessment purposes, defects are counted over a standard period of time, typically one year.
* **Maintainability**: Software maintenance accounts for more effort than any other software engineering activity. Maintainability is the ease with which a program can be corrected if an error is encountered, adapted if its environment changes, or enhanced if the customer desires a change in requirements. There is no way to measure maintainability directly; therefore, we must use indirect measures. A simple time-oriented metric is mean-time-to-change (MTTC), the time it takes to analyze the change request, design an appropriate modiﬁcation, implement the change, test it, and distribute the change to all users. On average, programs that are maintainable will have a lower MTTC (for equivalent types of changes) than programs that are not maintainable.
* **Integrity:** Software integrity has become increasingly important in the age of hackers and ﬁrewalls. This attribute measures a system's ability to with-stand attacks (both accidental and intentional) to its security. Attacks can be made on all three components of software: programs, data, and documents. To measure integrity, two additional attributes must be deﬁned: threat and security. Threat is the probability (which can be estimated or derived from empirical evidence) that an attack of a speciﬁc type will occur within a given time. Security is the probability (which can be estimated or derived from empirical evidence) that the attack of a speciﬁc type will be repelled. The integrity of a system can then be deﬁned as

 Integrity = summation [(1 – threat) (1 – security)]

 Where threat and security are summed over each type of attack.

* **Usability**: The catch phrase "user-friendliness" has become ubiquitous in discussions of software products. If a program is not user-friendly, it is often doomed to failure, even if the functions that it performs are valuable. Usability is an attempt to quantify user-friendliness and can be measured in terms of four characteristics: (1) the physical and or intellectual skill required to learn the system, (2) the time required to become moderately efﬁcient in the use of the system, (3) the net increase in productivity (over the approach that the system replaces) measured when the system is used by someone who is moderately efﬁcient, and (4) a subjective assessment (sometimes obtained through a questionnaire) of users attitudes toward the system.

**4-5-3 Defect Removal Efficiency**

 A quality metric that provides beneﬁt at both the project and process level is defect removal efﬁciency (DRE). When considered for a project as a whole, DRE is defined in the following manner:

DRE = E/(E + D)

 Where E is the number of errors found before delivery of the software to the end-user and D is the number of defects found after delivery. The ideal value for DRE is 1. That is, no defects are found in the software. Realistically, D will be greater than 0, but the value of DRE can still approach 1. As E increases (for a given value of D), the overall value of DRE begins to approach 1. In fact, as E increases, it is likely that the ﬁnal value of D will decrease (errors are ﬁltered out before they become defects).

DRE can also be used within the project to assess a team’s ability to ﬁnd errors before they are passed to the next framework activity or software engineering task. For example, the requirements analysis task produces an analysis model that can be reviewed to ﬁnd and correct errors. Those errors that are not found during the review of the analysis model are passed on to the design task (where they may or may not be found). When used in this context, we redeﬁne DRE as



where Ei is the number of errors found during software engineering activity i and Ei+1 is the number of errors found during software engineering activity i+1 that are traceable to errors that were not discovered in software engineering activity i.