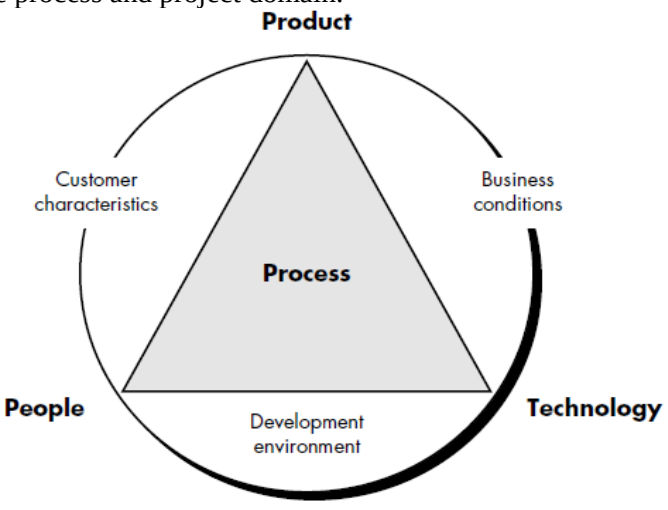
SOFTWARE METRICS

Measurement is fundamental to any engineering discipline, and software engineering is no exception. Measurement enables us to gain insight by providing a mechanism for objective evaluation. Software metrics refers to a broad range of measurements for computer software. Measurement can be applied to the software process with the intent of improving it on a continuous basis. Measurement can be used throughout a software project to assist in estimation, quality control, productivity assessment, and project control. Finally, measurement can be used by software engineers to help assess the quality of technical work products and to assist in tactical decision making as a project proceeds. There are four reasons for measuring software processes, products, and resources: to characterize, to evaluate, to predict, or to improve. Although the terms measure, measurement, and metrics are often used interchangeably, it is important to note the subtle differences between them. As measure can be used either as a noun or a verb, definitions of the term can become confusing. 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. The IEEE Standard Glossary of Software Engineering Terms defines metric as "a quantitative measure of the degree to which a system, component, or process possesses a given attribute."

Metrics should be collected so that process and product indicators can be ascertained. Process indicators enable a software engineering organization to gain insight into the efficacy of an existing process (i.e., the paradigm, software engineering tasks, work products, and milestones). They enable managers and practitioners to assess what works and what doesn't. 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.

FIGURE Determinants for software quality and organizational effectiveness



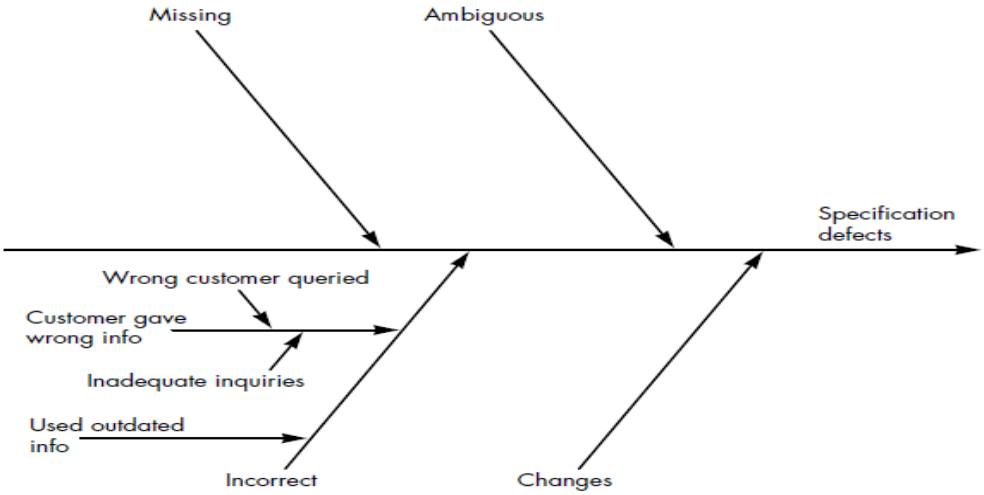
Process Metrics

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." We measure the efficacy of a software process indirectly. That is, we derive a set of metrics based on the outcomes that can be derived from the process. Outcomes include measures of errors uncovered before release of the software, defects delivered to and reported by end-users, work products delivered (productivity), human effort expended, calendar time expended, schedule conformance, and other measures. We also derive process metrics by measuring the characteristics of specific software engineering tasks. For example, we might measure the effort and time spent performing the umbrella activities and the generic software engineering activities. As an organization becomes more comfortable with the collection and use of process metrics, the derivation of simple indicators gives way to a more rigorous approach called statistical software process improvement (SSPI). In essence, SSPI uses software failure analysis to collect information about all errors and defects encountered as an application, system, or product is developed and used. Failure analysis works in the following manner:

1) All errors and defects are categorized by origin (e.g., flaw in specification, flaw in logic, nonconformance to standards).
2) The cost to correct each error and defect is recorded.
3) The number of errors and defects in each category is counted and ranked in descending order.
4) The overall cost of errors and defects in each category is computed.
5) Resultant data are analyzed to uncover the categories that result in highest cost to the organization.
6) Plans are developed to modify the process with the intent of eliminating (or reducing the frequency of) the class of errors and defects that is most costly.

A fishbone diagram can help in diagnosing defect data. The spine of the diagram (the central line) represents the quality factor under consideration (in this case specification defects that account for 25 percent of the total). Each of the ribs (diagonal lines) connecting to the spine indicate potential causes for the quality problem (e.g., missing requirements, ambiguous specification, incorrect requirements, changed requirements). The spine and ribs notation is then added to each of the major ribs of the diagram to expand upon the cause noted. Expansion is shown only for the incorrect cause. The collection of process metrics is the driver for the creation of the fishbone diagram. A completed fishbone diagram can be analyzed to derive indicators that will enable a software organization to modify its process to reduce the frequency of errors and defects.



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 technical work commences, other project metrics begin to have significance. Production rates represented in terms of pages of documentation, review hours, function points, and delivered source lines are measured. In addition, errors uncovered during each software engineering task are tracked. As the software evolves from specification into design, technical metrics are collected to assess design quality and to provide indicators that will influence the approach taken to code generation and testing.

The intent of project metrics is twofold. First, these metrics are used to minimize the development schedule by making the adjustments necessary to avoid delays and mitigate potential problems and risks. Second, project metrics are used to assess product quality on an ongoing basis and, when necessary, modify the technical approach to improve quality.

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.

Another 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.

In actuality, this model can be applied to both process and project. In the project context, the model can be applied recursively as each framework activity occurs. Therefore the output from one activity becomes input to the next. Results metrics can be used to provide an indication of the usefulness of work products as they flow from one framework activity (or task) to the next.

Software Measurement

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 to build 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.

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 software organization maintains simple records, a table of size-oriented measures, such as the one shown below, 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. Referring to the table entry 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.

Project	LOC	Effort	$(000)	Pp. doc.	Errors	Defects	People
alpha	12,100	24	168	365	134	29	3
beta	27,200	62	440	1224	321	86	5
gamma	20,200	43	314	1050	256	64	6
...	...	...	...	...	...	...	...

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.

Size-oriented metrics are not universally accepted as the best way to measure the process of software development –the reason being, use of lines of code as a key measure. Proponents of the LOC measure claim that LOC is an "artifact" of all software development projects that can be easily counted, that many existing software estimation models use LOC or KLOC as a key input, and that a large body of literature and data predicated on LOC already exists. On the other hand, opponents argue that LOC measures are programming language dependent, that they penalize well-designed but shorter programs, that they cannot easily accommodate nonprocedural languages, and that their use in estimation requires a level of detail that may be difficult to achieve (i.e., the planner must estimate the LOC to be produced long before analysis and design have been completed).

Function-Oriented Metrics (Function Points)

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 were first proposed by Albrecht, who suggested a measure called the function point. Function points are derived using an empirical relationship based on countable (direct) measures of software's information domain and assessments of software complexity. The basis of function points is that the "functionality" of a system, that is, what the system performs, is the measure of the system size. In function points, the system functionality is calculated in terms of the number of functions it implements, the number of inputs, the number of outputs, etc.—parameters that can be obtained after requirements analysis and that are independent of the specification (and implementation) language.

The original formulation for computing the function points uses the count of five different parameters, namely, external input types, external output types, logical internal file types, external interface file types, and external inquiry types. According to the function point approach, these five parameters capture the entire functionality of a system. However, two elements of the same type may differ in their complexity and hence should not contribute the same amount to the "functionality" of the system. To account for complexity, each parameter in a type is classified as *simple, average,* or *complex.* The definition of each of these types and the interpretation of their complexity levels is given later.

Function type	Simple	Average	Complex
External input	3	4	6
External output	4	5	7
Logical internal file	7	10	15
External interface file	5	7	10
External inquiry	3	4	6

Table : Function point contribution of an element.

Each unique input (data or control) type that is given as input to the application from outside is considered of *external input type* and is counted. An external input type is considered unique if the format is different from others or if the specifications require a different processing for this type from other inputs of the same format. The source of the external input can be the user, or some other application, files. An external input type is considered *simple* if it has a few data elements and affects only a few internal files of the application. It is considered *complex* if it has many data items and many internal logical files are needed for processing them. The complexity is *average* if it is in between. Note that files needed by the operating system or the hardware (e.g., configuration files) are not counted as external input files because they do not belong to the application but are needed due to the underlying technology.

Similarly, each unique output that leaves the system boundary is counted as an *external output type.* Again, an external output type is considered unique if its format or processing is different. Reports or messages to the users or other applications are counted as external input types. The complexity criteria are similar to those of the external input type. For a report, if it contains a few columns it is considered *simple,* if it has multiple columns it is considered *average,* and if it contains complex structure of data and references many files for production, it is considered *complex.* Each application maintains information internally for performing its functions.

Each logical group of data or control information that is generated, used, and maintained by the application is counted as a logical internal file type. A logical internal file is simple if it contains a few record types, complex if it has many record types and average if it is in between.

Files that are passed or shared between applications are counted as external interface file type. Note that each such file is counted for all the applications sharing it. The complexity levels are defined as for logical internal file type.

A system may have queries also, where a query is defined as an input-output combination where the input causes the output to be generated almost immediately. Each unique input-output pair is counted as an external inquiry type. A query is unique if it differs from others in format of input or output or if it requires different processing. For classifying the query type, the input and output are classified as for external input type and external output type, respectively. The query complexity is the larger of the two.

Each element of the same type and complexity contributes a fixed and same amount to the overall function point count of the system (which is a measure of the functionality of the system), but the contribution is different for the different types, and for a type, it is different for different complexity levels. The amount of contribution of an element is shown in table below.

Once the counts for all five different types are known for all three different complexity classes, the raw or unadjusted function point (UFP) can be computed as a weighted sum as follows:

$$UFP = \sum_{i=1}^{i=5} \sum_{j=1}^{j=3} w_{ij} C_{ij},$$

where ***i*** reflects the row and ***j*** reflects the column in the table above; w_{ij} is the entry in the ith row and jth column of the table (i.e., it represents the contribution of an element of the type **i** and complexity **j**); and $\mathbf{C_{ij}}$ is the count of the number of elements of type **i** that have been classified as having the complexity corresponding to column **j** .

Once the UFP is obtained, it is adjusted for the environment complexity. For this, 14 different characteristics of the system are given. These are:

1) data communications

2) distributed processing
3) performance objectives
4) operation configuration load
5) transaction rate
6) on-line data entry
7) end user efficiency
8) on-line update
9) complex processing logic
10) re-usability
11) installation ease
12) operational ease
13) multiple sites
14) desire to facilitate change

The degree of influence of each of these factors is taken to be from 0 to 5, representing the six different levels:

a) not present (0)
b) insignificant influence (1)
c) moderate influence (2)
d) average influence (3)
e) significant influence (4)
f) strong influence (5)

The 14 degrees of influence for the system are then summed, giving a total N (N ranges from 0 to 14*5=70). This N is used to obtain a complexity adjustment factor (CAP) as follows:

CAP = 0.65 + 0.01N

With this equation, the value of CAF ranges between 0.65 and 1.35. The delivered function points (DFP) are simply computed by multiplying the UFP by CAF. That is:

Delivered Function Points = CAF * Unadjusted Function Point

As we can see, by adjustment for environment complexity, the DFP can differ from the UFP by at most 35%. The final function point count for an application is the computed DFP.

Function points have been used as a size measure extensively and have been used for cost estimation. Studies have also been done to establish correlation between DFP and the final size of the software (measured in lines of code). As can be seen from the manner in which the functionality of the system is defined, the function point approach has been designed for the data processing type of applications. For data processing applications, function points generally perform very well and have now gained a widespread acceptance. For such applications, function points are used as an effective means of estimating cost and evaluating productivity. However, its utility as a size measure for nondata processing types of applications (e.g., real-time software, operating systems, and scientific applications) has not been well established, and it is generally believed that for such applications function points are not very well suited.

A major **drawback** of the function point approach is that the process of computing the function points involves subjective evaluation at various points and the final computed function point for a given SRS may not be unique and can depend on the analyst. Some of the places where subjectivity enters are: (1) different interpretations of the SRS (e.g., whether something should count as an external input type or an external interface type; whether or not something constitutes a logical internal file; if two reports differ in a very minor way should they be counted as two or one); (2) complexity estimation of a user function is totally subjective and depends entirely on the analyst (an analyst may classify something as complex while someone else may classify it as average) and complexity can have a substantial impact on the final count as the weighs for simple and complex frequently differ by a factor of 2; and (3) value judgments for the environment complexity. These factors make the process of function point counting somewhat subjective. Organizations that use function points try to specify a more precise set of counting rules in an effort to reduce this subjectivity. It has also been found that with experience this subjectivity is reduced.

Overall, despite this subjectivity, use of function points for data processing applications continues to grow.

The main **advantage** of function points over the size metric of KLOC, the other commonly used approach, is that the definition of DFP depends only on information available from the specifications, whereas the size in KLOC cannot be directly determined from specifications. Furthermore, the DFP count is independent of the language in which the project is implemented. Though these are major advantages, another drawback of the function point approach is that even when the project is finished, the DFP is not uniquely known and has subjectivity. This makes building of models for cost estimation hard, as these models are based on information about completed projects (cost models are discussed further in the next chapter). In addition, determining the DFP—from either the requirements or a completed project—cannot be automated. That is, considerable effort is required to obtain the size, even for a completed project. This is a drawback compared to KLOC measure, as KLOC can be determined uniquely by automated tools once the project is completed.

Quality Metrics

As we have seen, the quality of the SRS has direct impact on the cost of the project. Hence, it is important to ensure that the SRS is of good quality. For this, some quality metrics are needed that can be used to assess the quality of the SRS. Quality of an SRS can be assessed either directly by evaluating the quality of the document by estimating the value of one or more of the quality attributes of the SRS, or indirectly, by assessing the effectiveness of the quality control measures used in the development process during the requirements phase. Quality attributes of the SRS are generally hard to quantify, and little work has been done in quantifying these attributes and determining correlation with project parameters. Hence, the use of these metrics is still limited. However, process-based metrics are better understood and used more widely for monitoring and controlling the requirements phase of a project.

Number of errors found is a process metric is useful for assessing the quality of requirement specifications. Once the number of errors of different categories found during the requirement review of the project is known, some assessment can be made about the SRS from the size of the project and historical data. This assessment is possible if the development process is under statistical control. In this situation, the error distribution during requirement reviews of a project will show a pattern similar to other projects executed following the same development process. From the pattern of errors to be expected for this process and the size of the current project (say, in function points), the volume and distribution of errors expected to be found during requirement reviews of this project can be estimated. These estimates can be used for evaluation.

For example, if much fewer than expected error were detected, it means that either the SRS was of very high quality or the requirement reviews were not careful. Further analysis can reveal the true situation. If too many clerical errors were detected and too few omission type errors were detected, it might mean that the SRS was written poorly or that the requirements review meeting could not focus on "larger issues" and spent too much effort on "minor" issues. Again, further analysis will reveal the true situation. Similarly, a large number of errors that reflect ambiguities in the SRS can imply that the problem analysis has not been done properly and many more ambiguities may still exist in the SRS. Some project management decision to control this can then be taken (e.g., build a prototype or do further analysis).

Clearly, review data about the number of errors and their distribution can be used effectively by the project manager to control quality of the requirements. From the historical data, a rough estimate of the number of errors that remain in the SRS after the reviews can also be estimated. This can be useful in the rest of the development process as it gives some handle on how many requirement errors should be revealed by later quality assurance activities.

Requirements rarely stay unchanged. Change requests come from the clients (requesting added functionality, a new report, or a report in a different format, for example) or from the developers (infeasibility, difficulty in implementing, etc.). Change request frequency can be used as a metric to assess the stability of the requirements and how many changes in requirements to expect during the later stages.

Many organizations have formal methods for requesting and incorporating changes in requirements. We have earlier seen a requirements change management process. Change data can be easily extracted from these formal change approval procedures. The frequency of changes can also be plotted against time. For most projects, the frequency decreases with time. This is to be expected; most of the changes will occur early, when the requirements are being analyzed and understood. During the later phases, requests for changes should decrease.

For a project, if the change requests are not decreasing with time, it could mean that the requirements analysis has not been done properly. Frequency of change requests can also be used to "freeze" the requirements—when the frequency goes below an acceptable threshold, the requirements can be considered frozen and the design can proceed. The threshold has to be determined based on experience and historical data.
