CHAPTER 22

METRICS FOR PROCESS AND PROJECTS

KEY CONCEPTS
DRE
metrics
- function-oriented
- object-oriented
- private
- project
- process
- public
- quality
- size-oriented
- use-case
- WebApp

metrics baseline
metrics programs
SSPI

Measurement enables us to gain insight into the process and the project by providing a mechanism for objective evaluation. Lord Kelvin once said:

> When you can measure what you are speaking about and express it in numbers, you know something about it; but when you cannot measure, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind: it may be the beginning of knowledge, but you have scarcely, in your thoughts, advanced to the stage of a science.

The software engineering community has taken Lord Kelvin's words to heart. But not without frustration and more than a little controversy!

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 work products and to assist in tactical decision-making as a project proceeds (Chapter 15).

In their guidebook on software measurement, Park, Goethert, and Florac [PAR96] note the reasons that we measure: (1) to *characterize* in an effort to gain an understanding "of processes, products, resources, and environments, and to establish baselines for comparisons with future assessments"; (2) to *evaluate* "to determine status with respect to plans"; (3) to *predict* by "gaining understandings

QUICK LOOK

What is it? Software process and project metrics are quantitative measures that enable software engineers to gain insight into the efficacy of the software process and the projects that are conducted using the process as a framework. Basic quality and productivity data are collected. These data are then analyzed, compared against past averages, and assessed to determine whether quality and productivity improvements have occurred. Metrics are also used to pinpoint problem areas so that remedies can be developed and the software process can be improved.

Who does it? Software metrics are analyzed and assessed by software managers. Measures are often collected by software engineers.

Why is it important? If you don't measure, judgment can be based only on subjective evaluation. With measurement, trends (either good or bad) can be spotted, better estimates can be made, and true improvement can be accomplished over time.

What are the steps? Begin by defining a limited set of process and project measures that are easy to collect. These measures are often normalized using either size- or function-oriented metrics. The result is analyzed and compared to

past averages for similar projects performed within the organization. Trends are assessed and conclusions are generated.

What is the work product? A set of software metrics that provides insight into the process and an understanding of the project.

How do I ensure that I've done it right? By applying a consistent, yet simple measurement scheme that is never used to assess, reward, or punish individual performance.

of relationships among processes and products and building models of these relationships"; and (4) to *improve* by "identify[ing] roadblocks, root causes, inefficiencies, and other opportunities for improving product quality and process performance."

Measurement is a management tool. If conducted properly, it provides a project manager with insight. And as a result, it assists the project manager and the software team in making decisions that will lead to a successful project.

22.1 Metrics in the Process and Project Domains

KEY POINT

Process metrics have long-term impact. Their intent is to improve the process itself. Project metrics often contribute to the development of process metrics.

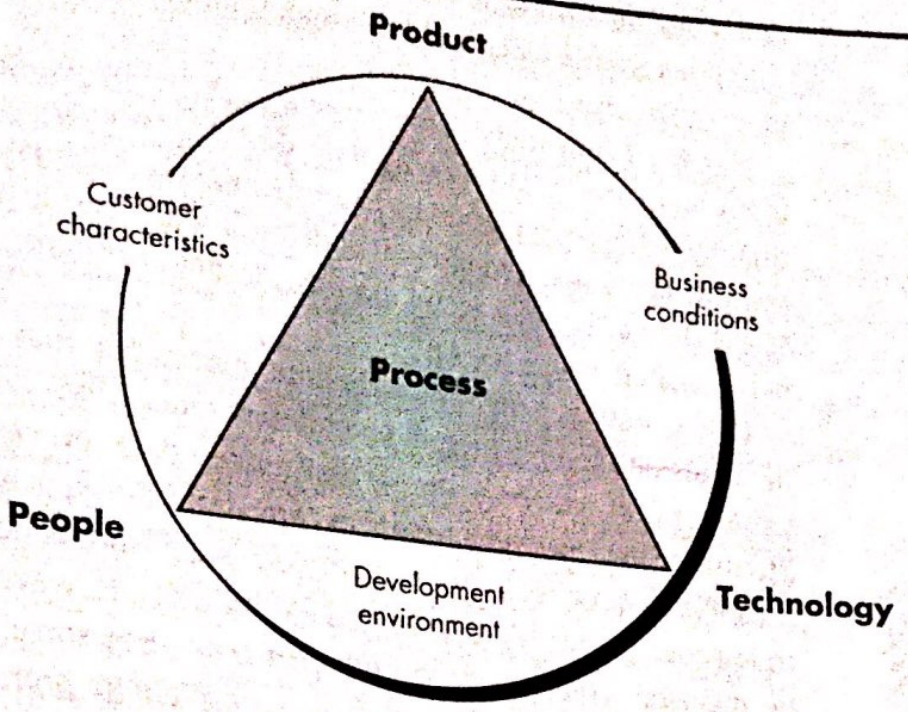
Process metrics are collected across all projects and over long periods of time. Their intent is to provide a set of process indicators that lead to long-term software process improvement. *Project metrics* enable a software project manager to (1) assess the status of an ongoing project, (2) track potential 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.

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.

22.1.1 Process Metrics and Software Process Improvement

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" [PAU94].

Referring to Figure 22.1, process sits at the center of a triangle connecting three factors that have a profound influence on software quality and organizational performance. The skill and motivation of people has been shown [BOE81] to be the single most influential factor in quality and performance. The complexity of the product can have a substantial impact on quality and team performance. The technology (i.e., the software engineering methods and tools) that populates the process also has an impact.



Determinants for software quality and organizational effectiveness (adapted from [PAU94])

In addition, the process triangle exists within a circle of environmental conditions that include the development environment (e.g., CASE tools), business conditions (e.g., deadlines, business rules), and customer characteristics (e.g., ease of communication and collaboration).

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 generic software engineering activities described in Chapter 2.

KEY POINT

skill and vation of the vare people doing vork are the most rtant factors that ence software y.

> "Software metrics let you know when to laugh and when to cry."
>
> **Tom Gilb**

What is the difference een private ublic uses oftware cs?

Grady [GRA92] argues that there are "private and public" uses for different types of process data. Because it is natural that individual software engineers might be sensitive to the use of metrics collected on an individual basis, these data should be private to the individual and serve as an indicator for the individual only. Examples of *private metrics* include defect rates by individual, defect rates by software component, and errors found during development.

The "private process data" philosophy conforms well with the personal software process approach (Chapter 2) proposed by Humphrey [HUM95]. Humphrey recognizes

that software process improvement can and should begin at the individual level. Private process data can serve as an important driver as the individual software engineer works to improve.

Some process metrics are private to the software project team but public to all team members. Examples include defects reported for major software functions (that have been developed by a number of practitioners), errors found during formal technical reviews, and lines of code or function points per component or function.[1] These data are reviewed by the team to uncover indicators that can improve team performance.

Public metrics generally assimilate information that originally was private to individuals and teams. Project level defect rates (absolutely not attributed to an individual), effort, calendar times, and related data are collected and evaluated in an attempt to uncover indicators that can improve organizational process performance.

Software process metrics can provide significant benefit as an organization works to improve its overall level of process maturity. However, like all metrics, these can be misused, creating more problems than they solve. Grady [GRA92] suggests a "software metrics etiquette" that is appropriate for both managers and practitioners as they institute a process metrics program:

What guidelines should be applied when we collect software metrics?

- Use common sense and organizational sensitivity when interpreting metrics data.
- Provide regular feedback to the individuals and teams who collect measures and metrics.
- Don't use metrics to appraise individuals.
- Work with practitioners and teams to set clear goals and metrics that will be used to achieve them.
- Never use metrics to threaten individuals or teams.
- Metrics data that indicate a problem area should not be considered "negative." These data are merely an indicator for process improvement.
- Don't obsess on a single metric to the exclusion of other important metrics.

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[2] encountered as an application, system, or product is developed and used.

1 Lines of code and fun...

22.1.2 Project Metrics

Unlike software process metrics that are used for strategic purposes, software project metrics 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 workflow 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 models created, 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 requirements into design, technical metrics (Chapter 15) are collected to assess design quality and to provide indicators that will influence the approach taken to code generation and testing.

How should we use trics during the oject itself?

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.

SAFEHOME

Establishing a Metrics Approach

The scene: Doug Miller's office as the *SafeHome* software project is about to begin.

The players: Doug Miller (manager of the *SafeHome* software engineering team) and Vinod Raman and Jamie Lazar, members of the product software engineering team.

The conversation:

Doug: Before we start work on this project, I'd like you guys to define and collect a set of simple metrics. To start, you'll have to define your goals.

Jamie (interrupting): And based on the timeline management has been talking about, we'll never have the time. What good are metrics anyway?

Doug (raising his hand to stop the onslaught): Slow down and take a breath, guys. The fact that we've never done it before is all the more reason to start now, and the metrics work I'm talking about shouldn't take much time at all . . . in fact, it just might save us time.

Vinod: How?

Doug: Look, we're going to be doing a lot more in-

intelligent, become Web enabled, all that . . . and we need to understand the process we use to build software . . . and improve it so we can build software better. The only way to do that is to measure.

Jamie: But we're under time pressure, Doug. I'm not in favor of more paper pushing . . . we need the time to do our work, not collect data.

Doug (calmly): Jamie, an engineer's work involves collecting data, evaluating it, and using the results to improve the product and the process. Am I wrong?

Jamie: No, but . . .

Doug: What if we hold the number of measures we collect to no more than five or six and focus on quality?

Vinod: No one can argue against high quality . . .

Jamie: True . . . but, I don't know, I still think this isn't necessary.

Doug: I'm going to ask you to humor me on this one. How much do you guys know about software metrics?

Jamie (looking at Vinod): Not much.

Doug: Here are some Web refs . . . spend a few hours getting up to speed.

Jamie (smiling): I thought you said this wouldn't take any time.

Doug: Time you spend learning is never wasted . . . go do it and then we'll establish some goals, ask a few questions, and define the metrics we need to collect.

22.2 Software Measurement

In Chapter 15, we noted that software measurement can be categorized in two ways: (1) *direct measures* of the software process (e.g., cost and effort applied) and product (e.g., lines of code (LOC) produced, execution speed, and defects reported over some set period of time), and (2) *indirect measures* of the product that include functionality, quality, complexity, efficiency, reliability, maintainability, and many other "-abilities" discussed in Chapter 15.

> "Not everything that can be counted counts, and not everything that counts can be counted."
>
> **Albert Einstein**

Project metrics can be consolidated to create process metrics that are public to the software organization as a whole. But how does an organization combine metrics that come from different individuals or projects?

To illustrate, we 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. Both size- and function-oriented metrics are normalized in this manner.

ADVICE

use many factors software work, use metrics to are individuals or

22.2.1 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 software organization maintains simple records, a table of size-oriented measures, such as the one shown in Figure 22.2, 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. Referring to the table entry (Figure 22.2) 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.

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), defects per KLOC, $ per KLOC, pages of documentation per KLOC. In addition, other interesting metrics can be computed: errors per person-month, KLOC per person-month, $ per page of documentation.

Size-oriented metrics are not universally accepted as the best way to measure the software process [JON86]. Most of the controversy swirls around the 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,

OINT
ented metrics
ely used, but
about their
and
ility continues.

RE 22.2
riented
cs

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
...	...	...	...	...	...	...	...

opponents argue that LOC measures are programming language dependent, that when productivity is considered, 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).

22.2.2 Function-Oriented Metrics

Function-oriented software metrics use a measure of the functionality delivered by the application as a normalization value. The most widely used function-oriented metric is the *function point* (FP). Computation of the function point is based on characteristics of the software's information domain and complexity. The mechanics of FP computation has been discussed in Chapter 15.[3]

The function point, like the LOC measure, is controversial. Proponents claim that FP is programming language independent, making it ideal for applications using conventional and nonprocedural languages, and that it is based on data that are more likely to be known early in the evolution of a project, making FP more attractive as an estimation approach. Opponents claim that the method requires some "sleight of hand" in that computation is based on subjective rather than objective data, that counts of the information domain (and other dimensions) can be difficult to collect after the fact, and that FP has no direct physical meaning—it's just a number.

22.2.3 Reconciling LOC and FP Metrics

The relationship between lines of code and function points depends 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. To quote Albrecht and Gaffney [ALB83]:

> The thesis of this work is that the amount of function to be provided by the application (program) can be estimated from the itemization of the major components[4] of data to be used or provided by it. Furthermore, this estimate of function should be correlated to both the amount of LOC to be developed and the development effort needed.

The following table[5] [QSM02] provides rough estimates of the average number of lines of code required to build one function point in various programming languages:

---

3 See Section 15.3.1 for a detailed discussion of FP computation.

4 It is important to note that "the itemization of major components" can be interpreted in a variety of ways. Software engineers who work in an object- ber of classes

Programming Language	LOC per Function point			
	Avg.	Median	Low	High
Access	35	38	15	47
Ada	154	—	104	205
APS	86	83	20	184
ASP 69	62	—	32	127
Assembler	337	315	91	694
C	162	109	33	704
C++	66	53	29	178
Clipper	38	39	27	70
COBOL	77	77	14	400
Cool:Gen/IEF	38	31	10	180
Culprit	51	—	—	—
DBase IV	52	—	—	—
Easytrieve+	33	34	25	41
Excel47	46	—	31	63
Focus	43	42	32	56
FORTRAN	—	—	—	—
FoxPro	32	35	25	35
Ideal	66	52	34	203
IEF/Cool:Gen	38	31	10	180
Informix	42	31	24	57
Java	63	53	77	—
JavaScript	58	63	42	75
JCL	91	123	26	150
JSP	59	—	—	—
Lotus Notes	21	22	15	25
Mantis	71	27	22	250
Mapper	118	81	16	245
Natural	60	52	22	141
Oracle	30	35	4	217
PeopleSoft	33	32	30	40
Perl	60	—	—	—
PL/1	78	67	22	263
Powerbuilder	32	31	11	105
REXX	67	—	—	—
RPG II/III	61	49	24	155
SAS	40	41	33	49
Smalltalk	26	19	10	55
SQL	40	37	7	110
VBScript36	34	27	50	—
Visual Basic	47	42	16	158

A review of these data indicates that one LOC of C++ provides approximately 2.4 times the "functionality" (on average) as one LOC of C. Furthermore, one LOC of a Smalltalk provides at least four times the functionality of a LOC for a conventional programming language such as Ada, COBOL, or C. Using the information contained in the table, it is possible to "backfire" [JON98] existing software to estimate the number of function points, once the total number of programming language statements are known.

Function points and LOC-based metrics have been found to be relatively accurate predictors of software development effort and cost. However, to use LOC and FP for estimation (Chapter 23), a historical baseline of information must be established.

Within the context of process and project metrics, we are concerned primarily with productivity and quality—measures of software development "output" as a function of effort and time applied and measures of the "fitness for use" of the work products that are produced. For process improvement and project planning purposes, our interest is historical. What was software development productivity on past projects? What was the quality of the software that was produced? How can past productivity and quality data be extrapolated to the present? How can it help us improve the process and plan new projects more accurately?

22.2.4 Object-Oriented Metrics

Conventional software project metrics (LOC or FP) can be used to estimate object-oriented software projects. However, these metrics do not provide enough granularity for the schedule and effort adjustments that are required as we iterate through an evolutionary or incremental process. Lorenz and Kidd [LOR94] suggest the following set of metrics for OO projects:

Number of scenario scripts. A scenario script (analogous to use-cases discussed throughout Parts 2 and 3 of this book) is a detailed sequence of steps that describes the interaction between the user and the application. The number of scenario scripts is directly correlated to the size of the application and to the number of test cases that must be developed to exercise the system once it is constructed.

Number of key classes. *Key classes* are the "highly independent components" [LOR94] that are defined early in object-oriented analysis (Chapter 8).[6] Because key classes are central to the problem domain, the number of such classes is an indication of the amount of effort required to develop the software and also an indication of the potential amount of reuse to be applied during

Customization index, $C = N_{dp}/(N_{dp} + N_{sp})$

The value of C ranges from 0 to 1. As C grows larger the level of WebApp customization becomes a significant technical issue.

Similar Web application metrics can be computed and correlated with project measures such as effort expended, errors and defects uncovered, and models or documentation pages produced. As the database grows (after a number of projects have been completed), relationships between the WebApp measures and project measures will provide indicators that can aid in project estimation.

SOFTWARE TOOLS

Project and Process Metrics

Objective: To assist in the definition, collection, evaluation, and reporting of software measures and metrics.

Mechanics: Each tool varies in its application, but all provide mechanisms for collecting and evaluating data that lead to the computation of software metrics.

Representative Tools[8]

Function Point WORKBENCH, developed by Charismatek (www.charismatek.com.au), offers a wide array of FP-oriented metrics.

MetricCenter, developed by Distributive Software (www.distributive.com), supports automated data collection, analysis, chart formatting, report generation, and other measurement tasks.

PSM Insight, developed by Practical Software and Systems Measurement (www.psmsc.com), assists in the creation and subsequent analysis of a project measurement database.

SLIM tool set, developed by QSM (www.qsm.com), provides a comprehensive set of metrics and estimation tools.

SPR tool set, developed by Software Productivity Research (www.spr.com), offers a comprehensive collection of FP-oriented tools.

TychoMetrics, developed by Predicate Logic, Inc. (www.predicate.com), is a tool suite for management metrics collection and reporting.

22.3 METRICS FOR SOFTWARE QUALITY

The overriding goal of software engineering is to produce a high-quality system, application, or product within a timeframe that satisfies a market need. 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.

Private metrics collected by individual software engineers are assimilated to provide project-level results. Although many quality measures can be collected, the

8 Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

primary thrust at the project level is to measure errors and defects. Metrics derived from these measures provide an indication of the effectiveness of individual and group software quality assurance and control activities.

Metrics such as work product (e.g., requirements or design) errors per function point, errors uncovered per review hour, and errors uncovered per testing hour provide insight into the efficacy of each of the activities implied by the metric. Error data can also be used to compute the *defect removal efficiency* (DRE) for each process framework activity. DRE is discussed in Section 22.3.2.

22.3.1 Measuring Quality

Although there are many measures of software quality,[9] correctness, maintainability, integrity, and usability provide useful indicators for the project team. Gilb [GIL88] suggests definitions and measures for each.

WebRef
Excellent source of information on software quality and related topics (including metrics) can be found at www.qualityworld.com

Correctness. A program must operate correctly or it provides little value to its users. Correctness is the degree to which the software performs its required function. The most common measure for correctness is defects per KLOC, where a defect is defined as a verified lack of conformance to requirements. 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 cyber-terrorists and hackers. This attribute measures a system's ability to withstand 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:

$$\text{integrity} = \Sigma [1 - (\text{threat} \times (1 - \text{security}))]$$

For example, if threat (the probability that an attack will occur) is 0.25 and security (the likelihood of repelling an attack) is 0.95, the integrity of the system is 0.99 (very high). If, on the other hand, the threat probability is 0.50 and the likelihood of repelling an attack is only 0.25, the integrity of the system is 0.63 (unacceptably low).

Usability. If a program is not easy to use, it is often doomed to failure, even if the functions that it performs are valuable. Usability is an attempt to quantify ease-of-use and can be measured in terms of characteristics presented in Chapter 12.

The four factors just described are only a sampling of those that have been proposed as measures for software quality. Chapter 15 considers this topic in additional detail.

22.3.2 Defect Removal Efficiency

A quality metric that provides benefits at both the project and process level is *defect removal efficiency* (DRE). In essence, DRE is a measure of the filtering ability of quality assurance and control activities as they are applied throughout all process framework activities.

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). If used as a metric that provides an indicator of the filtering ability of quality control and assurance activities, DRE encourages a software project team to institute techniques for finding as many errors as possible before delivery.

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 design (where they may or may not be found). When used in this context, we redefine DRE as

$$DRE_i = E_i / (E_i + E_{i+1})$$

where E_i is the number of errors found during software engineering activity i and E_{i+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.

ADVICE

is low as you through analysis esign, spend time improving ay you conduct technical s.

A quality objective for a software team (or an individual software engineer) is to achieve DRE, that approaches 1. That is, errors should be filtered out before they are passed on to the next activity.

SAFEHOME

Establishing a Metrics Approach

The scene: Doug Miller's office two days after initial meeting on software metrics.

The players: Doug Miller (manager of the *SafeHome* software engineering team) and Vinod Raman and Jamie Lazar, members of the product software engineering team.

The conversation:

Doug: You both had a chance to learn a little about process and project metrics?

Vinod and Jamie: [Both nod]

Doug: It's always a good idea to establish goals when you adopt any metrics. What are yours?

Vinod: Our metrics should focus on quality. In fact, our overall goal is to keep the number of errors we pass on from one software engineering activity to the next to an absolute minimum.

Doug: And be very sure you keep the number of defects released with the product to as close to zero as possible.

Vinod (nodding): Of course.

Jamie: I like DRE as a metric, and I think we can use it for the entire project. Also, we can use it as we move from one framework activity to the next. It'll encourage us to find errors at each step.

Vinod: I'd also like to collect the number of hours we spend on reviews.

Jamie: And the overall effort we spend on each software engineering task.

Doug: You can compute a review-to-development ratio . . . might be interesting.

Jamie: I'd like to track some use-case data as well. Like the amount of effort required to develop a use-case, the amount of effort required to build software to implement a use-case, and . . .

Doug (smiling): I thought we were going to keep this simple.

Vinod: We should, but once you get into this metrics stuff, there's a lot of interesting things to look at.

Doug: I agree, but let's walk before we run, and stick to our goal. Limit data to be collected to five or six items, and we're ready to go.

22.4 INTEGRATING METRICS WITHIN THE SOFTWARE PROCESS

The majority of software developers still do not measure, and sadly, most have little desire to begin. As we noted earlier in this chapter, the problem is cultural. Attempting to collect measures where none had been collected in the past often precipitates resistance. "Why do we need to do this?" asks a harried project manager. "I don't see the point," complains an overworked practitioner.

In this section, we consider some arguments for software metrics and present an approach for instituting a metrics collection program within a software engineering organization. But before we begin, some words of wisdom are suggested by Grady and Caswell [GRA87]:

> Some of the things we describe here will sound quite easy. Realistically, though, establishing a successful company-wide software metrics program is hard work. When we say

that you must wait at least three years before broad organizational trends are available, you get some idea of the scope of such an effort.

The caveat suggested by the authors is well worth heeding, but the benefits of measurement are so compelling that the hard work is worth it.

22.4.1 Arguments for Software Metrics

Why is it so important to measure the process of software engineering and the product (software) that it produces? The answer is relatively obvious. If we do not measure, there is no real way of determining whether we are improving. And if we are not improving, we are lost.

By requesting and evaluating productivity and quality measures, a software team (and their management) can establish meaningful goals for improvement of the software process. In Chapter 1 we noted that software is a strategic business issue for many companies. If the process through which it is developed can be improved, a direct impact on the bottom line can result. But to establish goals for improvement, the current status of software development must be understood. Hence, measurement is used to establish a process baseline from which improvements can be assessed.

> "We manage things by the numbers in many aspects of our lives . . . These numbers give us insight and help steer our actions."
>
> **Michael Mah and Larry Putnam**

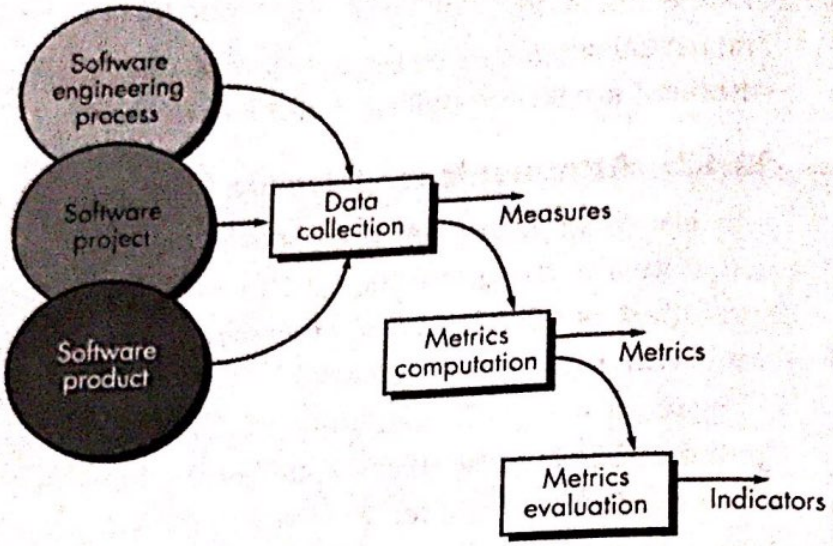
The day-to-day rigors of software project work leave little time for strategic thinking. Software project managers are concerned with more mundane (but equally important) issues: developing meaningful project estimates, producing higher-quality systems, getting product out the door on time. By using measurement to establish a project baseline, each of these issues becomes more manageable. We have already noted that the baseline serves as a basis for estimation. Additionally, the collection of quality metrics enables an organization to "tune" its software process to remove the "vital few" causes of defects that have the greatest impact on software development.[10]

22.4.2 Establishing a Baseline

What is a metrics baseline, and what benefit does it provide a software engineer?

By establishing a metrics baseline, benefits can be obtained at the process, project, and product (technical) levels. Yet the information that is collected need not be fundamentally different. The same metrics can serve many masters. The metrics baseline consists of data collected from past software development projects and can be as simple as the table presented in Figure 22.2 or as complex as a comprehensive database containing dozens of project measures and the metrics derived from them.

FIGURE 22.3
Software metrics collection process



To be an effective aid in process improvement and/or cost and effort estimation, baseline data must have the following attributes: (1) data must be reasonably accurate—"guestimates" about past projects are to be avoided; (2) data should be collected for as many projects as possible; (3) measures must be consistent, for example, a line of code must be interpreted consistently across all projects for which data are collected; (4) applications should be similar to work that is to be estimated—it makes little sense to use a baseline for batch information systems work to estimate a real-time, embedded application.

22.4.3 Metrics Collection, Computation, and Evaluation

KEY POINT
Baseline metrics data should be collected from a large representative sampling of past software projects.

The process for establishing a metrics baseline is illustrated in Figure 22.3. Ideally, data needed to establish a baseline has been collected in an on-going manner. Sadly, this is rarely the case. Therefore, data collection requires a historical investigation of past projects to reconstruct required data. Once measures have been collected (unquestionably the most difficult step), metrics computation is possible. Depending on the breadth of measures collected, metrics can span a broad range of application-oriented metrics (e.g., LOC, FP, object-oriented, WebApp) as well as other quality- and project-oriented metrics. Finally, metrics should be evaluated and applied during estimation, technical work, project control, and process improvement. Metrics evaluation focuses on the underlying reasons for the results obtained and produces a set of indicators that guide the project or process.

22.5 METRICS FOR SMALL ORGANIZATIONS

The vast majority of software development organizations have fewer than 20 soft-