

UNIT-V Software Quality

SOFTWARE QUALITY

Conformance to explicitly stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software.

SOFTWARE QUALITY ASSURANCE

SQA Activities

Software quality assurance is composed of a variety of tasks associated with two different constituencies—the software engineers who do technical work and an SQA group that has responsibility for quality assurance planning, oversight, record keeping, analysis, and reporting. The charter of the SQA group is to assist the software team in achieving a high quality end product.

Prepares an SQA plan for a project. The plan is developed during project planning and is reviewed by all interested parties. Quality assurance activities performed by the software engineering team and the SQA group are governed by the plan. The plan identifies

- evaluations to be performed
- audits and reviews to be performed
- standards that are applicable to the project
- procedures for error reporting and tracking
- documents to be produced by the SQA group
- amount of feedback provided to the software project team

Participates in the development of the project's software process description.

The software team selects a process for the work to be performed. The SQA group reviews the process description for compliance with organizational policy, internal software standards, externally imposed standards (e.g., ISO-9001), and other parts of the software project plan.

Reviews software engineering activities to verify compliance with the defined

software process. The SQA group identifies, documents, and tracks deviations from the process and verifies that corrections have been made.

Audits designated software work products to verify compliance with those

defined as part of the software process. The SQA group reviews selected work products; identifies, documents, and tracks deviations; verifies that corrections have been made; and periodically reports the results of its work to the project manager.

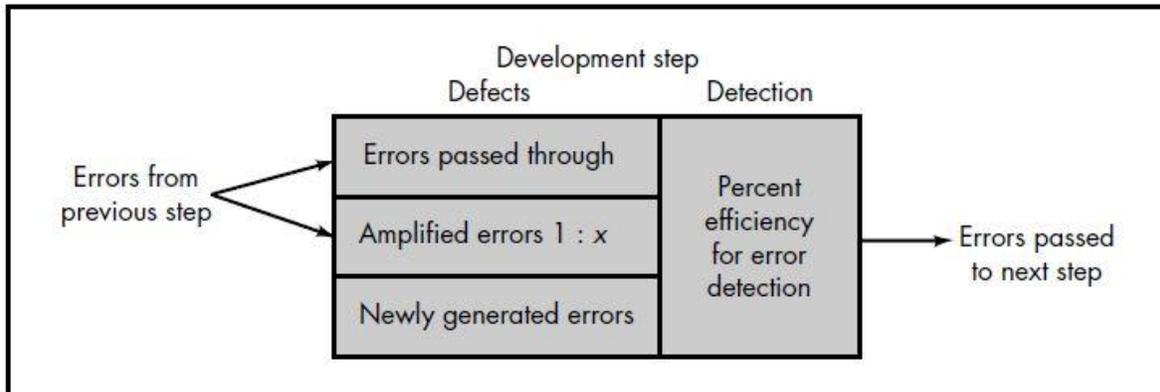
Ensures that deviations in software work and work products are documented

and handled according to a documented procedure. Deviations may be encountered in the project plan, process description, applicable standards, or technical work products.

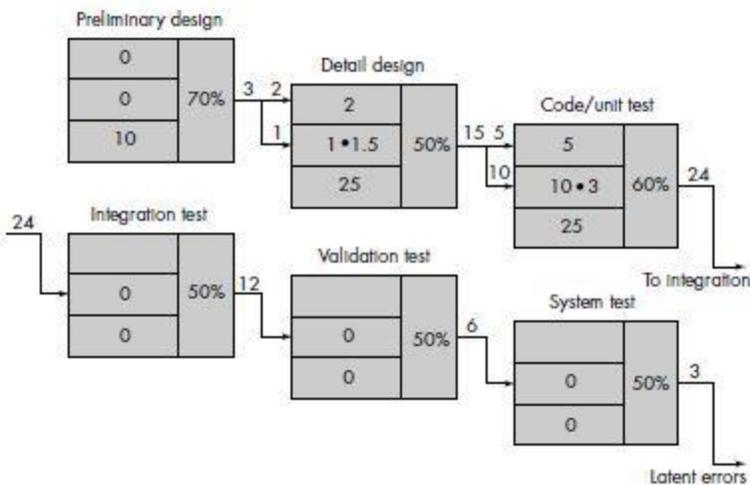
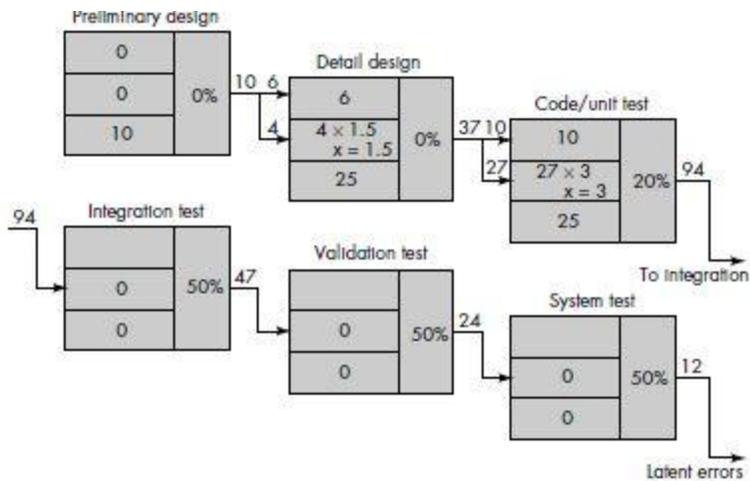
Records any noncompliance and reports to senior management. Noncompliance items are tracked until they are resolved.

Defect Amplification and Removal

A defect amplification model can be used to illustrate the generation and detection of errors during the preliminary design, detail design, and coding steps of the software engineering process. The model is illustrated schematically in Figure. A box represents a software development step. During the step, errors may be inadvertently generated. Review may fail to uncover newly generated errors and errors from previous steps, resulting in some number of errors that are passed through. In some cases, errors passed through from previous steps are amplified (amplification factor, x) by current work. The box subdivisions represent each of these characteristics and the percent of efficiency for detecting errors, a function of the thoroughness of the review.



Following Figure illustrates a hypothetical example of defect amplification for a software development process in which no reviews are conducted. Referring to the figure, each test step is assumed to uncover and correct 50 percent of all incoming errors without introducing any new errors (an optimistic assumption). Ten preliminary design defects are amplified to 94 errors before testing commences. Twelve latent errors are released to the field. Figure 8.4 considers the same conditions except that design and code reviews are conducted as part of each development step. In this case, ten initial preliminary design errors are amplified to 24 errors before testing commences. Only three latent errors exist. Recalling the relative costs associated with the discovery and correction of errors, overall cost (with and without review for our hypothetical example) can be established.



MEASURES, METRICS, AND INDICATORS

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). A 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 person-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

Project Metrics

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.

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.

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.

Function-Oriented Metrics

Function-oriented software metrics use a measure of the functionality delivered by the application as a normalization value,

Function points are computed by completing the table shown in Figure .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. Each user input that provides distinct application oriented data to the software is counted. Inputs should be distinguished from inquiries, which are counted separately.

Number of user outputs. Each user output that provides application oriented information to the user is counted. In this context output refers to reports, screens, error messages, etc. Individual data items within a report are not counted separately.

Number of user inquiries. An inquiry is defined as an on-line input that results in the generation of some immediate software response in the form of an on-line output. Each distinct inquiry is counted.

Number of files. Each logical master file (i.e., a logical grouping of data that may be one part of a large database or a separate file) is counted.

Number of external interfaces. All machine readable interfaces (e.g., data files on storage media) that are used to transmit information to another system are counted.

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. Nonetheless, the determination of complexity is somewhat subjective.

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

$$FP = \text{count total} \times [0.65 + 0.01 \times \sum(F_i)] \quad (4-1)$$

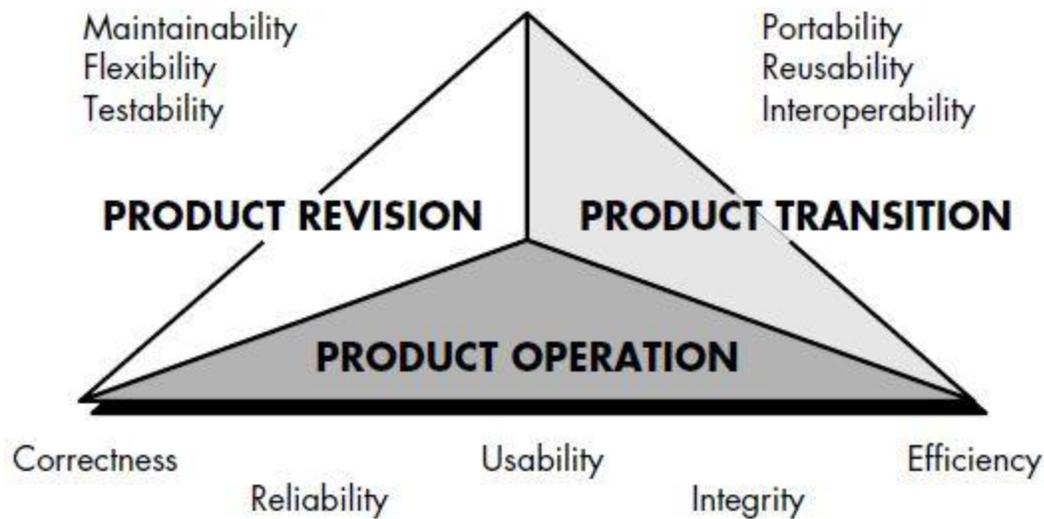
where count total is the sum of all FP entries obtained from Figure

The F_i ($i = 1$ to 14) are "complexity adjustment values" based on responses to the questions

McCall's Quality Factors

The factors that affect software quality can be categorized in two broad groups:

- (1) factors that can be directly measured (e.g., defects per function-point) and
- (2) factors that can be measured only indirectly (e.g., usability or maintainability).



McCall and his colleagues provide the following descriptions:

Correctness. The extent to which a program satisfies its specification and fulfills the customer's mission objectives.

Reliability. The extent to which a program can be expected to perform its intended function with required precision

Efficiency. The amount of computing resources and code required by a program to perform its function.

Integrity. Extent to which access to software or data by unauthorized persons can be controlled.

Usability. Effort required to learn, operate, prepare input, and interpret output of a program.

Maintainability. Effort required to locate and fix an error in a program.

Flexibility. Effort required to modify an operational program.

Testability. Effort required to test a program to ensure that it performs its intended function.

Portability. Effort required to transfer the program from one hardware and/or software system environment to another.

Reusability. Extent to which a program [or parts of a program] can be reused in other applications—related to the packaging and scope of the functions that the program

performs.

Interoperability. Effort required to couple one system to another.

ISO 9126 Quality Factors

Functionality. The degree to which the software satisfies stated needs as indicated by the following subattributes: suitability, accuracy, interoperability, compliance, and security.

Reliability. The amount of time that the software is available for use as indicated by the following subattributes: maturity, fault tolerance, recoverability.

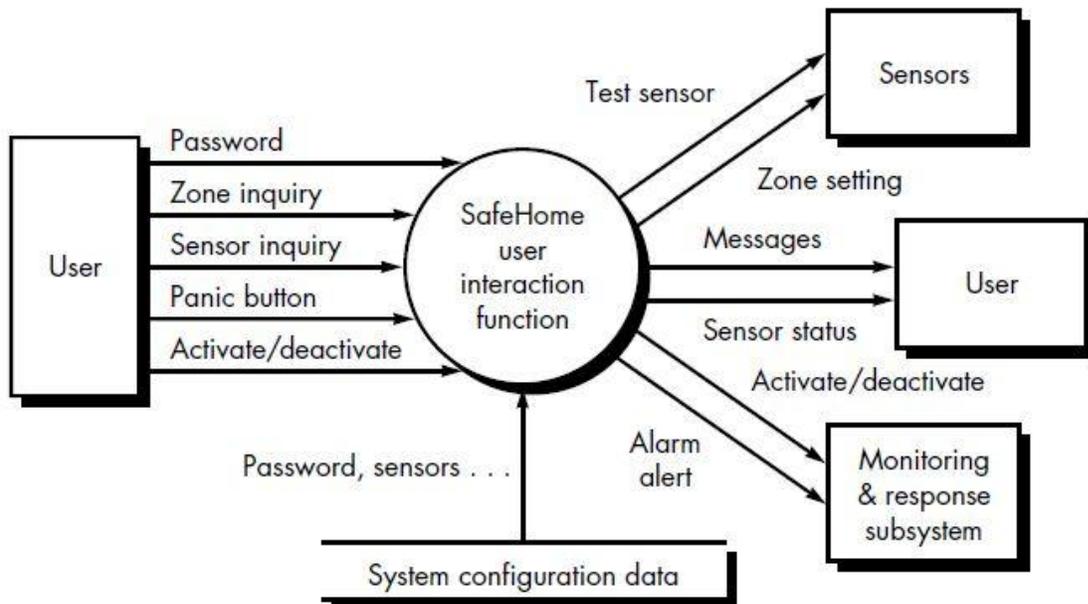
Usability. The degree to which the software is easy to use as indicated by the following subattributes: understandability, learnability, operability.

Efficiency. The degree to which the software makes optimal use of system resources as indicated by the following subattributes: time behavior, resource behavior.

Maintainability. The ease with which repair may be made to the software as indicated by the following subattributes: analyzability, changeability, stability, testability.

Portability. The ease with which the software can be transposed from one environment to another as indicated by the following subattributes: adaptability, installability, conformance, replaceability.

METRICS FOR THE ANALYSIS MODEL



Function-Based Metrics/Function Oriented Metrics

The function point metric can be used effectively as a means for predicting the size of a system that will be derived from the analysis model. To illustrate the use of the FP metric in this context, we consider a simple analysis model representation, illustrated in Figure. Referring to the figure, a data flow diagram for a function within the *SafeHome* software is represented. The function manages user interaction, accepting a user password to activate or deactivate the system, and allows inquiries on the status of security zones and various security sensors. The function displays a series of prompting messages and sends appropriate control signals to various components of the security system.

The data flow diagram is evaluated to determine the key measures required for computation of the function point metric:

- number of user inputs
- number of user outputs
- number of user inquiries
- number of files
- number of external interfaces

Measurement parameter	Count	Weighting Factor			=	
		Simple	Average	Complex		
Number of user inputs	3	3	4	6	=	9
Number of user outputs	2	4	5	7	=	8
Number of user inquiries	2	3	4	6	=	6
Number of files	1	7	10	15	=	7
Number of external interfaces	4	5	7	10	=	20
Count total	→					50

Three user inputs—**password**, **panic button**, and **activate/deactivate**—are shown in the figure along with two inquiries—**zone inquiry** and **sensor inquiry**. One file (**system configuration file**) is shown. Two user outputs (**messages** and **sensor status**) and four external interfaces (**test sensor**, **zone setting**, **activate/deactivate**, and **alarm alert**) are also present. These data, along with the appropriate complexity, are shown in Figure.

The *count total* shown in Figure must be adjusted using Equation:

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

where *count total* is the sum of all FP entries obtained from Figure and Fi ($i = 1$ to 14) are "complexity adjustment values." For the purposes of this example,

we assume that $\sum (Fi)$ is 46 (a moderately complex product). Therefore,
 $FP = 50 \sum [0.65 + (0.01 \sum 46)] = 56$

Based on the projected FP value derived from the analysis model, the project team can estimate the overall implemented size of the *SafeHome* user interaction function. Assume that past data indicates that one FP translates into 60 lines of code (an object oriented language is to be used) and that 12 FPs are produced for each person-month of effort. These historical data provide the project manager with important planning information that is based on the analysis model rather than preliminary estimates. Assume further that past projects have found an average of three errors per function point during analysis and design reviews and four errors per function point during unit and integration testing. These data can help software engineers assess the completeness of their review and testing activities.

Information domain values are defined in the following manner:

Number of user inputs. Each user input that provides distinct application oriented data to the software is counted. Inputs should be distinguished from inquiries, which are counted separately.

Number of user outputs. Each user output that provides application oriented information to the user is counted. In this context output refers to reports, screens, error messages, etc. Individual data items within a report are not counted separately.

Number of user inquiries. An inquiry is defined as an on-line input that results in the generation of some immediate software response in the form of an on-line output. Each distinct inquiry is counted.

Number of files. Each logical master file (i.e., a logical grouping of data that may be one part of a large database or a separate file) is counted.

Number of external interfaces. All machine readable interfaces (e.g., data files on storage media) that are used to transmit information to another system are counted.

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. Nonetheless, the determination of complexity is somewhat subjective.

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

$$FP = \text{count total} \sum [0.65 + 0.01 \sum (Fi)] \quad (4-1)$$

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

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

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). The constant values in Equation (4-1) and the weighting factors that are applied to information domain counts are determined empirically.

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.

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 4.4, 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 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

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
•	•	•	•	•	•		
•	•	•	•	•	•		
•	•	•	•	•	•		

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.

Process Metrics

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 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) that populate the process also has an impact. 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).

