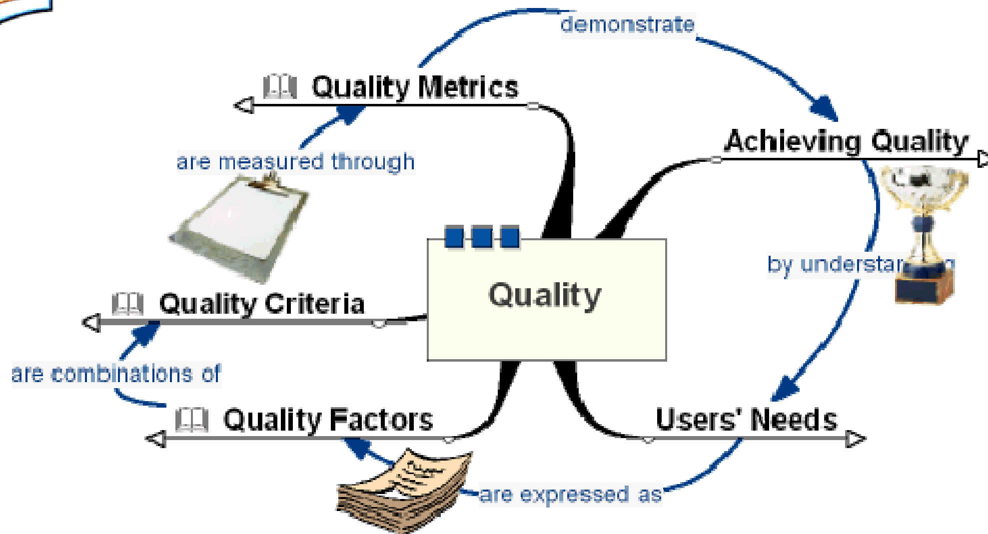
Defining Quality



1 Achieving Quality

by understanding: Users' Needs.

1.1 Engineer

To engineer in quality means to add quality to software during the engineering process

To achieve this, software engineers must be aware of quality requirements at the same time they are building the functional requirements

Quality requirements thus take on the same relationship to the product as functional requirements do

If we engineer quality in, we add quality to the product as we build it

1.2 Review

Reviewing software means to place requirements, designs, and code as well as other life-cycle work products under manual inspection for compliance with standards and specifications

A noncompliant feature found during an inspection is a defect

Reviews are conducted on requirements after the requirements have been specified, on design after the design is complete, and on code after the programming is complete

Reviews are a critical step in achieving software quality but it is not enough --> certain levels of reliability cannot be achieved merely as a result of more review

1.3 Test

Testing is the oldest of the three basic ways of achieving quality

To test software means to execute the software under simulated conditions to see whether or not it fails

Because testing depends on a program's execution, it occurs after the software has been specified, designed, and coded

Testing is a critical step in achieving software quality, but it is not enough --> software that is not maintainable will not improve merely as a result of more testing

2 Users' Needs

are expressed as: Quality Factors.

2.1 Operational

2.1.1 Functional

Functionality deals with what the software does while executing

1.1.1.1 How secure is it?

See also: Integrity.

1.1.1.2 How often will it fail?

See also: Reliability.

1.1.1.3 Can it survive during failure

See also: Survivability.

1.1.1.4 How easy is it to use?

See also: Usability.

2.1.2 Performance

Performance deals with how well it does it

1.1.1.5 How much is needed in the way of resources?

See also: Efficiency.

1.1.1.6 Does it comply with requirements?

See also: Correctness.

1.1.1.7 Does it prevent hazards?

See also: Safety.

1.1.1.8 Does it interface easily?

See also: Interoperability.

2.2 Maintenance

2.2.1 Change

Change deals with modifying the software to correct errors, adapt code to new environments, or add new functionality

1.1.1.9 How easy is it to repair?

See also: Maintainability.

1.1.1.10 How easy is it to expand?

See also: Expandability.

1.1.1.11 How easy is it to change?

See also: Flexibility.

1.1.1.12 How easy is it to transport?

See also: Portability.

1.1.1.13 Is it reusable in other systems?

See also: Reusability.

2.2.2 Management

Management needs deal with planning for change, controlling versions of the software, testing, and installation

1.1.1.14 Is performance verification easy?

See also: Verifiability.

Possibility to provide the evidence that the system works, without extensive testing by the customer.

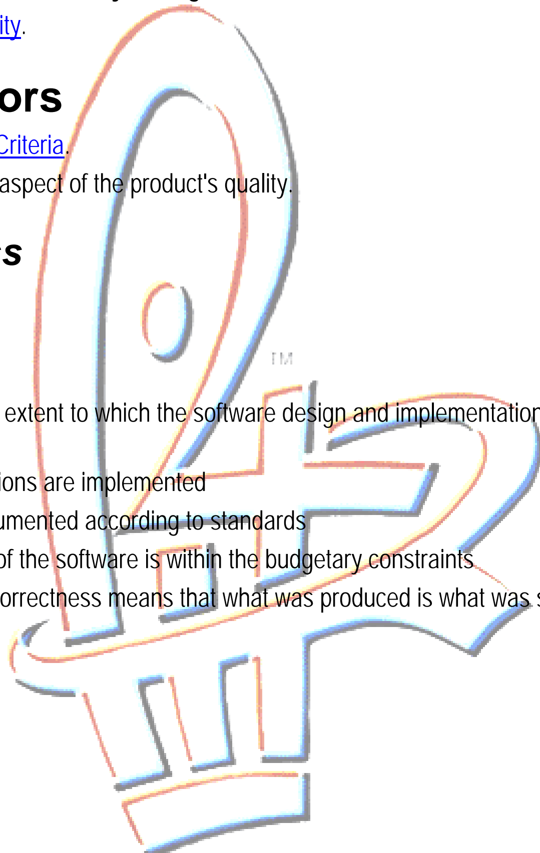
1.1.1.15 Is the software easily managed?

See also: Manageability.

3 Quality Factors

are combinations of: Quality Criteria.

The user oriented view of an aspect of the product's quality.



3.1 Correctness

See also: Completeness.

See also: Consistency.

See also: Traceability.

Correctness deals with the extent to which the software design and implementation conform to the stated requirements

Whether all specified functions are implemented

Whether the design is documented according to standards

Whether the performance of the software is within the budgetary constraints

Fitness for use regarding correctness means that what was produced is what was specified and vice versa

3.2 Efficiency

See also: Communication.

See also: Processing.

See also: Storage.

Efficiency deals with the resources needed to provide the required functionality.

Processor or execution efficiency

Random-access memory or storage efficiency

Communication lines

Fitness for use regarding efficiency means that the resources required for the execution of the software are affordable

3.3 Expandability

See also: Generality.

See also: Modularity.

See also: Self-Descriptiveness.

See also: Simplicity.

See also: Support.

See also: Augmentability.

Expandability deals with the aspects of software maintenance that is increasing the software's functionality or performance to meet new needs

Adding a new type of deduction to an existing payroll program

Adding an interface to a new sensor

Fitness for use regarding expandability means that the software was built to be open-ended making it easy to modify it to add new capabilities

3.4 Flexibility

See also: Generality.

See also: Modularity.

See also: Self-Descriptiveness.

See also: Simplicity.

Flexibility deals with the adaptive needs of software maintenance that is modifying the software to work in different environments

A word processor's ability to adapt to different kinds of printers

An air traffic control system's ability to adapt to different airport configurations

Fitness for use regarding flexibility means that the software is readily adaptable, via a user interface or change of data to a wide range of different environments without immediately resorting to expandability type changes

3.5 Integrity

See also: System Accessibility.

Integrity deals with security against either overt or covert access to the programs or databases

Fitness for use regarding integrity means that the user is reasonably certain that the software and data are not being tampered with or stolen

3.6 Interoperability

See also: Commonality.

See also: Functional Scope.

See also: Independence.

See also: Modularity.

See also: System Compatibility.

Interoperability deals with how easy it is to couple the software with software in other systems or applications

Some spreadsheet packages are "integratable" into a chart-drawing package

Some databases may be combined into a single management information system

Fitness for use regarding interoperability means that the software produces or uses results that comply with industry or organization standards

3.7 Maintainability

See also: Completeness.

See also: Consistency.

See also: Modularity.

See also: Self-Descriptiveness.

See also: Simplicity.

See also: Traceability.

See also: Visibility.

Maintainability deals with the ease of finding and fixing errors

Fitness for use regarding maintainability means that the software is productive through the maintenance lifecycle, covering error detection through the issue of a new release

3.8 Manageability

See also: Document Quality.

See also: Support.

Manageability deals with the administrative aspects of modification to the software. It includes:

Tools to support changes such as configuration control systems

Media control such as tape or disk handling

Fitness for use regarding manageability means that the support environment is complete and easy to use

3.9 Portability

See also: Independence.

See also: Modularity.

See also: Self-Descriptiveness.

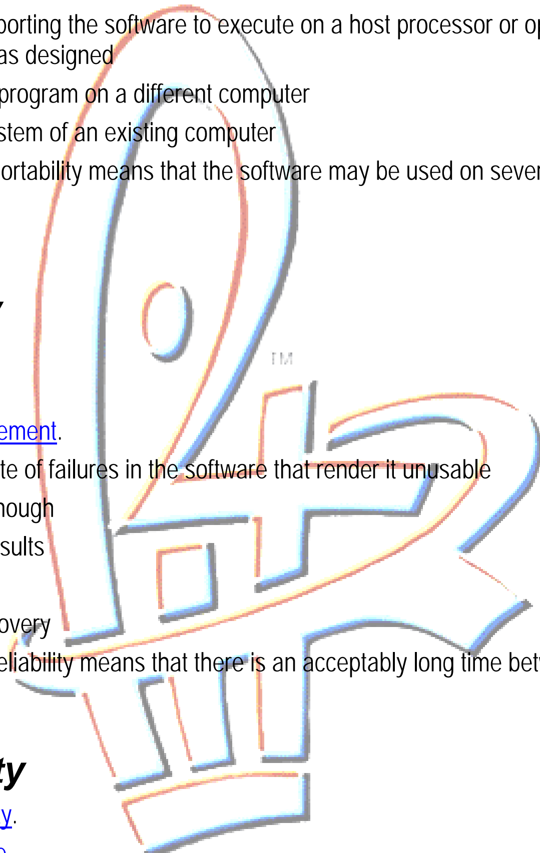
See also: Support.

Portability deals with transporting the software to execute on a host processor or operating system different from the one for which it was designed

Recompiling a FORTRAN program on a different computer

Changing the operating system of an existing computer

Fitness for use regarding portability means that the software may be used on several different operating systems or computers



3.10 Reliability

See also: Simplicity.

See also: Accuracy.

See also: Anomaly Management.

Reliability deals with the rate of failures in the software that render it unusable

Software is not accurate enough

Software gives incorrect results

Response time is too slow

Software stops without recovery

Fitness for use regarding reliability means that there is an acceptably long time between failures

3.11 Reusability

See also: Document Quality.

See also: Functional Scope.

See also: Generality.

See also: Independence.

See also: Modularity.

See also: Self-Descriptiveness.

See also: Simplicity.

See also: Support.

Reusability deals with the use of portions of the software for other applications. It includes:

Use of mathematical libraries in both statistical and scientific applications

Use of target-tracking algorithms in both civil air traffic control and military air defence applications

Fitness for use regarding reusability means that there is a large library of standard building blocks available to the software

3.12 Safety

See also: Distributivity.

See also: Safety Management.

See also: Accuracy.

See also: Anomaly Management.

Safety deals with the absence of unsafe software conditions

An unsafe software condition could lead to a hazard

loss of life or liability

damage to property

Fitness for use regarding safety means that the software can be trusted

3.13 Survivability

See also: Autonomy.

See also: Distributivity.

See also: Anomaly Management.

Survivability deals with the continuity of reliable software execution (perhaps with degraded functionality) in the presence of system failure

Fitness for use regarding survivability means that the user can still use essential functions even though some part of the system is down.

3.14 Usability

See also: Operability.

See also: Training.

Usability deals with the initial effort required to learn, and the recurring effort to use the functionality of the system

Usability can be enhanced or degraded by:

The naturalness of the user interface

The readability of documentation

The number of keystrokes required for a given command

Fitness for use regarding usability means that the software is easier to use than not to use

3.15 Verifiability

See also: Modularity.

See also: Self-Descriptiveness.

See also: Simplicity.

See also: Support.

See also: Traceability.

See also: Visibility.

Verifiability deals with how easy it is to verify that the software is working correctly

It includes the presence or absence of automatic self-testing and a library of standard test programs

Fitness for use regarding verifiability means that it is simple to certify that the software is correct

4 Quality Criteria

are measured through: Quality Metrics.

Software oriented attributes or quality attributes

4.1 Augmentability

The software is augmentable if it is easy to add new functional capability or new data - it is easy to expand its capabilities without major redesign or modification.

A software designer trying to achieve augmentability should always assume that the current capabilities will be expanded in the future and try to avoid constructs that limit that expansion or try to add constructs that make the possibility of expansion easier.

4.1.1 Is the software partitioned so that each partition is logically complete and self-contained?

4.1.2 Is the software designed to meet or exceed sparse resource requirements?

4.1.3 Have the performance requirements been implemented as adaptable parameters?

4.1.4 Is the database designed to be expanded and easily changed?

4.1.5 Are the software and hardware interfaces clearly defined and isolated?

4.1.6 Is commercial or reusable software used wherever possible?

4.1.7 Are all data base items referenced using common variable naming conventions?

4.2 Autonomy

The software is autonomous when, in the event of failure of interfacing components, services or devices, it can be automatically reconfigured and continue to execute - decoupled from its execution environment

4.2.1 Is the software designed to use the minimal amount of operating system services?

4.2.2 Are the calling sequences between interfacing software units simple and easy to understand?

4.2.3 Is the software free of encoded knowledge about devices or operating systems?

4.2.4 Are the hardware and device protocol processing independent?

4.2.5 Does the operating system and the hardware components have self-test services?

4.3 Commonality

Software exhibits the characteristics of commonality when it uses established standards for data representation, for interfaces to communication networks and for interfaces with the user.

4.3.1 Use a standard for communicating with other systems

4.3.2 Use one standard interface protocol

4.3.3 Use one single module or component for interface protocol handling

4.3.4 Use a standard data representation for communication with other systems

4.3.5 Use a common format for external messages

4.3.6 Use on single module or component for data representation translation

4.4 Completeness

Software is complete when every data item, function, interface, decision branch, line of code, etc., is necessary to accomplish the required capability:

There is no extra code that is not really necessary

There are sufficient data items, functions, etc., to implement all of the required capabilities

All, and only the actual program code is required to implement the requirements

4.4.1 All inputs, processing, and outputs should be clearly and precisely defined

4.4.2 All software requirements should be allocated to design components

4.4.3 All references should be unambiguous

4.4.4 All defined functions should be used

4.4.5 All referenced functions should be defined

4.4.6 All options should be defined for all decision points

4.4.7 All defined and referenced calling sequence parameters should be in agreement

4.5 Consistency

Software is consistent when the same standards are used throughout the design and implementation - When one can be assured that the data name "X" in one module refers to the same global item "X" referred to elsewhere.

To achieve consistency it is necessary to establish standards before the design and implementation effort begins and to ensure that the standards are uniformly enforced

4.5.1 Is there a standard design representation?

4.5.2 Is the use of data consistent through the project?

4.5.3 Are there standard data usage representations and naming conventions?

4.5.4 Are calling sequences and input/output and error conventions consistently used throughout development?

4.5.5 Are there consistent global, module, and data type definitions and have they been consistently applied?

4.5.6 Is there a requirement to standardise the external input/output protocol and format for all modules?

4.5.7 Is there a requirement to standardise error handling?

4.5.8 Do all references to the same function use a single, unique name?

4.5.9 Is there a requirement to standardise all data representations in the design?

4.5.10 Is there a requirement to standardise the naming of all data?

4.6 Distributivity

Software is distributed if its subparts are located on different processing or storage devices. The goal is to minimize the operational effect of failures, that is, to avoid single points of failure that can bring the whole system down. Thus, allocation or replication of functions and data on different devices will result in execution alternatives in case of a failure on a single device.

4.6.1 Are critical software components on two or more devices?

4.6.2 Is the control of the critical components distributed as well - not in only one location?

4.6.3 One function is allocated to a single device

4.6.4 One logical file is allocated to a single device

4.6.5 Files are accessible from all process elements on the distributed system

4.6.6 Are alternate routes available in case of processor failure somewhere on the distributed system?

4.7 Document Quality

Software is well documented if a person can easily access information about the software design and capabilities and the information is understandable and complete. For example, it should be easy to trace from an error condition to the failed component, and then it should be easy to understand the design of the component and be able to fix it. It should also be easy to look up information on how to use the software capabilities.

4.7.1 Documents are free from access control

4.7.2 Is all documentation structured and written clearly and simply such that procedures, functions, algorithms, and so forth can be easily understood?

4.7.3 Does the design documentation clearly depict control and data flow?

4.7.4 Is the documentation adequately indexed such that information can be easily accessed?

4.7.5 Does the documentation completely characterize the operational capabilities of the software (e.g., identify all the performance parameters and limitations)?

4.7.6 Is the software functionality documented clearly in separate documents?

4.7.7 Do comprehensive descriptions of algorithms exist?

4.8 Done

4.8.1 Accuracy

See also: Error Tolerance.

See also: Range.

See also: Available.

See also: Failures.

See also: Communication.

The software is said to be accurate if it produces results that are within required accuracy tolerances (e.g. +/- 0.001). To be accurate, mathematical libraries, numerical techniques, numerical conversations and the like must preserve the number of bits of precision during calculations.

4.8.2 Anomaly Management

See also: Error Tolerance.

See also: Range.

See also: Available.

See also: Failures.

See also: Loops and Indexes.

See also: Communication.

The software is said to have anomaly management if it can detect and recover from error conditions rather than disrupting processing or halting/. The software should be designed for survivability when faced with software and hardware failure. Anomaly management includes detection and containment of:

improper input data

computational failures

hardware faults

device failures

communication errors

4.9 Efficiency

Economic use of resources

4.9.1 Communication

Software communicates efficiently if it uses no more communication line bandwidth than is necessary to perform the required function, that is, if it minimizes the number of bits per second across a communication line (a network or telecommunication line)

1.1.1.16 Does the software meet or exceed the spare performance requirements for communication equipment?

1.1.1.17 Is the shortest path always chosen between two communicating components?

1.1.1.18 Data compression is used for communicating messages

1.1.1.19 Error correction is used instead of always resending messages

4.9.2 Processing

Software processes efficiently if it uses no more processor bandwidth than is necessary to perform the required functions, i.e. if it minimizes the number of instructions per second executed by a processor.

1.1.1.20 Does the software meet of exceed the spare performance requirements for processor equipment?

1.1.1.21 Is the number of memory overlays minimised?

1.1.1.22 Is an optimising compiler used if necessary?

1.1.1.23 Is the data storage designed for efficient processing?

1.1.1.24 Is data initialised at the point of declaration of the data?

1.1.1.25 Avoid non-loop dependent statements in loops

1.1.1.26 Avoid recalculating the same expression in more than one statement

1.1.1.27 Avoid data packing and unpacking in a loop

1.1.1.28 Avoid different-size data items in the same expression

1.1.1.29 Avoid mixed data types in the same expression

1.1.1.30 Minimise the number of data items modified in each module

4.9.3 Storage

Software stores data or instructions efficiently if is uses no more RAM or disk memory than is necessary to perform the required functions, that is, if it minimizes the number of bytes of storage.

1.1.1.31 Does the software meet or exceed the spare performance requirements for storage equipment?

1.1.1.32 Is virtual storage management used?

1.1.1.33 Is dynamic memory allocation used?

1.1.1.34 Is an optimising compiler used if necessary?

1.1.1.35 Avoid redundant storage of files and libraries

1.1.1.36 Is the software separated to efficiently use memory segments?

1.1.1.37 Are all data items packed?

4.10 Functional Scope

Software has good functional scope if its functions have a wide range of applicability, that is, if the functions are reusable in other similar applications or systems, or at other sites.

4.10.1 Avoid redundancy management for replicated functions

4.10.2 Is software reuse maximised in similar functions?

4.10.3 Has multi-site commonality with site-specific adaptability been designed in from the beginning?

4.10.4 Are all input/output functions documented as to their specific meaning and limitations?

4.10.5 Is the exact format of each I/O function specified?

4.11 Generality

Units of software are general if they provide a service for more than one function, that is, if they are called by two or more software components. In attempting to achieve generality, the designer should think about building a primitive function that can be of service for many uses. In many cases, this can be accomplished by moving constraints such as the data size item and type from within the unit to the parameters in the unit interface.

4.11.1 Input, processing, and output functions should not be mixed in a single module

4.11.2 Application and machine-dependent functions should not be mixed in a single module

4.11.3 Limits on input data structure size should be parameterized

4.11.4 Limits on input data item values should be parameterized

4.12 Independence

Application software that is independent does not contain any references to its environment, that is, the software can be lifted and reused in other environments with little or no modification. For example, the software is independent if there are no references to the uniqueness of the computing system, operating system, utilities, I/O routines, libraries, database system, microcode, computer architecture, or system algorithms.

4.12.1 Use a programming language (version) that is commonly available

4.12.2 The dependence on the software system library routines should be low

4.12.3 The software should be free from operating system references

4.12.4 Avoid non-standard constructs of the programming language

4.12.5 Minimise the number of units performing external I/O

4.12.6 Avoid expressions dependent on word or character size

4.12.7 Avoid machine-dependent data item representations

4.12.8 Avoid the use of assembly or machine languages

4.13 Modularity

Software is modular if it is designed and implemented with all of the modern techniques that lead to an orderly, cohesive component structure with optimum coupling. Modularity is a conglomeration of various techniques, goals, and attributes that have been proven to lead to highly maintainable software. The key element in achieving modularity is the use of a structured design technique that will result in an organized, well-defined structure of components.

4.13.1 Use a structured design technique

4.13.2 Use only structured programming language constructs

4.13.3 Use a structured testing technique

4.13.4 Represent designs in formally defined, unambiguous syntax

4.13.5 Design units that have a single processing objective

4.13.6 Maximize the cohesion of units

4.13.7 Minimise the complexity and volume in unit coupling

4.13.8 Avoid control variables used as formal parameters

4.13.9 Return control and all output data to the calling unit

4.13.10 Make local data and code inaccessible outside the local unit

4.13.11 Make each unit separately compilable

4.14 Operability

Software is operable if it is easy to use, that is, if it includes, among other things,
An understandable user interface
A minimum number of keystrokes for the most often performed job
Selectable options to adapt the software to a specific user
and if it keeps the user informed of conditions over which he/she has control.

Operability also includes the masking of system implementation details so that performing some job is natural from the user's perspective.

4.14.1 All steps of the operation should be described

4.14.2 All error conditions and responses should be reported to the operator

4.14.3 Have provisions been made for the operator to interrupt, obtain status, save, modify, and continue processing?

4.14.4 Are hard copy logs of interactions maintained?

4.14.5 Are there requirements to provide simple and consistent operator messages and require simple and consistent operator responses?

4.14.6 Is resource status information available?

4.14.7 Is it possible to reallocate functions and data to resources?

4.14.8 Are there requirements to make system implementation details transparent to the user (e.g., the user can access a file without knowing its location on the network)?

4.14.9 Are there requirements to provide the user with user-selectable input media (e.g., terminal, tape drive)?

4.14.10 Are there requirements to provide the user with user-selectable output media, formats, and amounts?

4.14.11 Have all output data items been provided with unique, descriptive labels?

4.14.12 Are meaningful units of measure used for output data items?

4.14.13 Are there requirements for all error messages to clearly identify the nature of the error to the user?

4.14.14 Do error messages specify the required user response?

4.14.15 Is the number of different message and response formats minimised?

4.15 Safety Management

Software is safe if it avoids unsafe conditions, i.e., those conditions that could lead to loss of life, property damage, or liability. Units of software (either data or functions) that participate in avoiding unsafe conditions are termed critical.

Software has good safety management when critical units are designed and implemented to avoid unsafe conditions. Avoiding unsafe conditions is achieved through separation of these critical units from non-critical units, by testing for unsafe conditions, and by fail-safe recovery from unsafe conditions when they occur. The term "critical" when used in other quality criteria (e.g., accuracy and anomaly management) often denotes safety requirements as well.

4.15.1 Separate critical functions from non-critical ones

4.15.2 Use the highest quality of unit interface (data coupling) between critical units

4.15.3 Separate error recovery code from normal code in critical units

4.15.4 Separate critical functions and data likely to change

4.15.5 Schedule the execution of critical units in a deterministic manner

4.15.6 Specify unsafe conditions and build in tests for them

4.15.7 Implement fail-safe recovery from detected unsafe conditions

4.16 Self Descriptiveness

Software is self-descriptive if a reader of the design or source code can easily understand how a function has been implemented, that is, if the software is a self-documented description of itself.

Self-descriptiveness deals with the quantity of comments included in the code, with their effectiveness, and with programming or design language descriptiveness.

4.16.1 Document the results of decisions in design representations

4.16.2 Identify external interfaces in design representations

4.16.3 Establish standards for writing unit prologues, commenting, and structuring code

4.16.4 Follow established unit standards

4.16.5 Establish and follow standards for global data commenting

4.16.6 Provide comments for all decision points and transfers of control

4.16.7 Provide comments for all machine-dependent code

4.16.8 Use a higher order language wherever possible

4.16.9 Provide comments for all non-standard statements

4.16.10 Provide comments for all data item attributes (e.g., range and accuracy)

4.16.11 Logical block and indent code

4.16.12 Do not overload the meanings of keywords in the programming language

4.16.13 Describe the purpose or intent of the code in code comments

4.16.14 Describe the properties of data items in the names of the data items

4.17 Simplicity

Software implementation is simple when it is straightforward, clear-cut, and unconfused. Software in this category is usually well thought out in advance and is sometimes programmed several times to find the simplest implementation. The opposite of simple code is sometimes referred to as "spaghetti code".

4.17.1 Minimize the use of common data blocks

4.17.2 Avoid singly used data items in common blocks

4.17.3 Use data items only in accordance with their descriptions

4.17.4 Make units independent of their input sources and output destinations

4.17.5 Provide single entrances to and single exits from units

4.17.6 Avoid Go To statements

4.17.7 Avoid the use of conditional branch statements

4.17.8 Avoid negative and compound Boolean statements

4.17.9 Avoid unnatural exits from loops

4.17.10 Avoid nesting beyond three to five levels

4.17.11 Avoid multiple statements on one line of code

4.17.12 Avoid continuation lines of code

4.17.13 Avoid repeated or redundant code

4.17.14 Do not alter loop variables within a loop

4.17.15 Avoid self-modifying code

4.17.16 Separate I/O functions from computational functions

4.17.17 Place similar functions into single packages

4.17.18 Avoid initializing global data in one unit and using the data in another unit

4.17.19 Minimize fan-out (the number of units invoked by a given unit)

4.18 Support

Software is supportive if it includes tools, databases, and procedures that ease the management of change after the software is delivered. Although these attributes refer to support software as opposed to operational software, they are important to the latter's fitness for use.

4.18.1 Close all program trouble reports before delivery

4.18.2 Provide services to support correction and upgrade of software

4.18.3 Provide services for on-call consultation

4.18.4 Document change management procedures

4.18.5 Provide tools to support control over site adaptation of different versions of software

4.18.6 Provide tools for change management

4.18.7 Provide tools for updates of documentation

4.18.8 Provide a data base or testing software and procedures

4.18.9 Automate software testing

4.18.10 Provide a complete software development environment

4.18.11 Provide a library of reusable software components

4.19 System Accessibility

The software (including any data bases that it interacts with) has good system accessibility if it has complete control over who accesses it and when and how it is accessed. In a word, system accessibility means security, including responding to access violations such as improper access logging and alerts by automatic disconnection.

4.19.1 Limit user access by user identification and passwords

4.19.2 Control data by authorization tables

4.19.3 Control the scope of task operations during execution

4.19.4 Control network access by authorization tables

4.19.5 Keep logs of all system accesses

4.19.6 Output alerts in case of access violations

4.19.7 Allow only valid data base operations

4.19.8 Monitor data bases to ensure that they are valid

4.19.9 Design in the ability to disconnect any interfacing system

4.19.10 Password-protect files from unauthorized access

4.19.11 Disallow unauthorized modifications to software

4.19.12 Protect against loss of services by inadvertent actions

4.19.13 Disallow bypass of security features

4.20 System Compatibility

Software has good system compatibility when it results in harmonious intersystem operations, that is, when the hardware, software, and communications functions work together as an interoperating whole. System compatibility is achieved primarily by designing software around the constraints imposed by the interoperating system.

4.20.1 Are there requirements for this system to use the same communication protocol as the interoperating system?

4.20.2 Are there requirements for common interpretation of the content of all messages sent from and received by this system and by the interoperating system (e.g., all variables in the message have the same meaning)?

4.20.3 Are there requirements for this system to use the same structures and sequence for message contents as the interoperating system?

4.20.4 Are there requirements for this system to use the same I/O rates as the interoperating system?

4.20.5 Are there requirements for this system to use the same communication protocol as the interoperating system?

4.20.6 Are there requirements for this system to establish the same data types, representations, and units as the interoperating system?

4.20.7 Are there requirements for this system to establish the same data base structure as the interoperating system?

4.20.8 Are there requirements for this system to establish the same data base access techniques as the interoperating system?

4.20.9 Are there requirements for this system to use the same word lengths as the interoperating system?

4.20.10 Are there requirements for this system to use the same interrupt structures as the interoperating system?

4.20.11 Are there requirements for this system to use the same source code languages as the interoperating system?

4.20.12 Are there requirements for this system to use the same operating system as the interoperating system?

4.20.13 Are there requirements for this system to use the same support software as the interoperating system?

4.21 Traceability

Software is traceable when it is easy to find all the code that implements a particular requirement and, conversely, when it is easy to find all the requirements that are implemented by a particular portion of code.

Traceability is important in verifying correctness and during perfective maintenance. It is achieved through careful and complete allocation of requirements downward to software components during the design process, and through documentation of the traceability and allocation results.

4.21.1 Document the origin of each requirement of each software component

4.21.2 Document each component to which each requirement is allocated

4.21.3 Graphically portray the decomposition of components

4.21.4 Implement software in conformity with design specifications

4.21.5 Document allocated requirements in unit prologues

4.22 Training

Software is capable of training users when it includes provisions to help them learn how to use the software to perform their jobs. These provisions include training classes, on-line help capability, and adjustable levels of sophistication in the user interface.

4.22.1 Have lesson plans and training material been included with the software?

4.22.2 Have realistic simulation exercises been created?

4.22.3 Is there sufficient "help" and diagnostic information available as an integral part of the software?

4.22.4 Are there requirements to provide selectable levels of aid and guidance for systems users of different levels of experience?

4.23 Virtuality

Software has the attribute of virtuality if it represents different physical components by the same logical or virtual interface. The representation can occur across any interface: user interacting with software, units calling other units, or software interfacing with devices. The importance of the representation is to be able to replace devices or functions without disrupting or changing the interface.

4.23.1 Present a logical (not physical) view of the system to the user

4.23.2 Isolate I/O interfaces to single units

4.23.3 Use logical record interfaces

4.23.4 Avoid encoding knowledge of the physical properties of the system

4.24 Visibility

Software is visible if there is insight into its process as regards coding and testing. Insight is gained through objective evidence of the correct functioning of the software during execution, primarily through the thoroughness of testing and the success of the test results. Visibility is equally applicable to both the initial development and maintenance changes. It is added to the software by embedding self-testing in the code and through completeness of software testing procedures

4.24.1 Provide tests for all software performance requirements

4.24.2 Design test scenarios to maximize the number of units tested

4.24.3 Document inputs and outputs for performance test cases

4.24.4 Provide tests for all unit-to-unit interfaces

4.24.5 Provide tests for all parameters in each unit interface

4.24.6 Provide tests for all execution paths in each unit

4.24.7 Add monitors to detect intermittent errors and marginal performance

4.24.8 Add built-in software to isolate defective components

5 Quality Metrics

demonstrate: Achieving Quality.

Quantitative measures of the characteristics.

These need to be established locally based on the information available, the goals of the metrics and the importance of the various aspects.

The less measurements can be used, to provide the maximum information, the less effort required to collect the measurements, the better the solution.