



CISQ Specifications for Automated Quality Characteristic Measures

Produced by CISQ Technical Work Groups for:

Reliability

Performance Efficiency

Security

Maintainability

CISQ-TR-2012-01

CONSORTIUM FOR IT SOFTWARE QUALITY

Executive Summary

This document describes a specification for automating the measurement of four Software Quality Characteristics—Reliability, Performance Efficiency, Security, and Maintainability. These measures are consistent with the definitions given for each Quality Characteristic in ISO/IEC 25010 which defines the Quality Characteristics for software systems. The measures for each Quality Characteristic aggregate counts of violations of rules of good architecture and coding practice related to that characteristic.

Each Quality Characteristic is decomposed into a set of issues that collectively cover its various sub-characteristics as defined in ISO/IEC 25010. Each issue is decomposed into a set of rules that collectively help avoid the consequences the issue it creates in operational systems. Each rule is translated into a violation that can be detected through automated analysis of the code. The set of rules and violations for each Quality Characteristic were selected based on being of sufficiently high severity that they need to be remediated in the code. The base measure for each rule is a count of these violations. The Quality Characteristic measure is the sum of all the violations of rules related to its issues.

This specification will be supplemented by other CISQ Technical Reports that describe how these measures can be used in evaluating and managing IT business applications. CISQ may also publish from time to time reports that further elaborate clarifying details or scoring options to aid the calculation and use of these measures. The CISQ membership may also decide to add specifications for automatable measures of other Quality Characteristics to this collection in the future.

© Object Management Group, 2012. The content of this document is copyrighted by the Object Management Group. It may be used with citation to OMG.

Executive Summary.....	3
1. Software Quality Characteristic Measurement	5
1.1 Purpose	5
1.2 Measuring Software Quality Characteristics.....	5
1.3 Objectives for Software Quality Characteristic Measures	6
1.4 Development of Software Quality Characteristic Measures	7
1.5 Rule Violation-Based Measures	8
2. Terms and Definitions	9
3. Compliance	11
3.1 Compliance	11
3.2 Required Inputs.....	11
4. CISQ Quality Characteristic Measures	12
4.1 Structure of Quality Characteristic Measures.....	12
4.2 Source of Quality Measure Elements	12
4.3 CISQ Reliability Measure.....	13
4.3.1 Definition of Reliability.....	13
4.3.2 CISQ Reliability Measure Elements	13
4.4 Performance Efficiency Measure.....	17
4.4.1 Definition of Performance Efficiency	17
4.4.2 CISQ Performance Efficiency Measure Elements.....	17
4.5 CISQ Security Measure.....	19
4.5.1 Definition of Security	19
4.5.2 CISQ Security Measure Elements.....	19
4.6 CISQ Maintainability Measure	25
4.6.1 Definition of Maintainability	25
4.6.2 CISQ Maintainability Measure Elements	25
5. Software Quality Characteristic Calculation Process	28
5.1 Weighting Schemes.....	28
5.2 Calculation Formula	28
6. References	29

1. Software Quality Characteristic Measurement

1.1 Purpose

The purpose of these measurement specifications is to create standards for measuring Software Quality Characteristics that are automated, objective, economical to use, and technically feasible. The objective of the work leading to these specifications is to provide international standard definitions against which IT organizations, IT service providers, and software vendors can implement automated measurement of the structural quality of software. In order to maintain consistency with the ISO/IEC 25000 series of standards (System and software Quality Requirements and Evaluation, SQuaRE), Software Quality Characteristics are defined for the purpose of this specification as attributes that can be measured from the static properties of software, and can be related to the dynamic properties of a computer system as affected by its software. The ISO/IEC 25000 series is replacing ISO/IEC 9126 and is the international standard for defining the elements of internal software quality.

Software Quality Characteristics are increasingly being incorporated into development and outsourcing contracts as the equivalent of Service Level Agreements (SLAs). That is, target thresholds measured by these Quality Characteristics are being set for delivered software. When thresholds are not met the supplier is subject to rework or financial penalties. Currently there are no standards defining most of the Software Quality Characteristic measures being used in contracts. Consequently, providers are subject to different interpretations and calculations of common Quality Characteristics in each contract. This specification addresses that problem by providing common measurement definitions and calculations for 4 Software Quality Characteristics that were prioritized by the member companies of the Consortium for IT Software Quality—reliability, security, performance efficiency, and maintainability.

1.2 Measuring Software Quality Characteristics

Measuring the internal or structural quality aspects of software has a long history in software engineering. Internal software quality measurement can be traced back to pioneering work in the 1970s by Maurice Halstead (1976), Thomas McCabe (1977), Tom Gilb (1976), Barry Boehm and his colleagues at TRW (1978), and Jim McCall and his colleagues at GE Space Division (1977). In particular, the work at TRW and GE in defining the attributes of a wide variety of Software Quality Characteristics was the precursors to the quality model in ISO/IEC 25010. The Quality Characteristics defined in ISO/IEC 25010 have never been defined to the level that they can be measured through automated analysis. CISQ was formed to take this next step in defining a subset of these Quality Characteristics.

Currently there are two primary approaches to measuring internal Software Quality Characteristics—measuring structural elements and measuring violations of good architectural and coding practice.

Structural Elements—the first and historical approach is based on counts of the structural elements of software. Halstead’s Software Science (1976), McCabe’s Cyclomatic Complexity (1976), Henry and Kafura’s information flow metrics (1981), and Chidamber and Kemerer’s Object-Oriented Metric Suite (1994) are examples of measurement based on formulas derived from counts of various structural elements.

Counts of structural characteristics have a 20-30 year history and are backed by numerous validation studies (Curtis, 1980). Counts of structural elements do not of themselves constitute a defect in the

software. Rather they are indicators of potential defects or problems. That is, the probability that the code possesses defects or will be the site of future defect injections increases with higher values of these Software Quality Characteristic measures. Consequently, these measures are often used to set threshold values that, when exceeded, require the offending component to be remediated.

Rule Violations—The second, and more recent addition to assessing the structural quality of software is based on the analysis and measurement of violations of rules of good coding and architectural practice that can be detected by analyzing the source code. These rule violations are occasionally referred to as anti-patterns, although these violations may not always qualify as a code pattern. The CWE/SANS 25 and OWASP Top Ten lists of security vulnerabilities are examples of this second approach. The Software Assurance community has been a leader in this area of measurement by championing the detection of rule violations as a way of improving one aspect of software quality—software security. Although the Software Assurance community has developed methods for scoring the severity of individual vulnerabilities, standards have not been developed for calculating component or application-level security measures that aggregate security-related rule violations detected through static code analysis into application-level security measures.

1.3 Objectives for Software Quality Characteristic Measures

To the extent possible CISQ measures quantify software some of the Quality Characteristics defined in ISO 25010 which is replacing ISO 9126. ISO 25010 defines a Quality Characteristic as being composed from several quality sub-characteristics. Each quality sub-characteristic consists of a collection of quality attributes that can be quantified as Quality Measure Elements. These Quality Measure Elements can either be counts of structural components or violations of rules of good architectural or coding practice.

Currently the ISO/IEC 25010 series has yet to define rules or violations of good coding practice that can be detected at the source code level and counted in order to compute measures of the Quality Characteristics. This specification extends these definitions to the detail required to create measures for each Quality Characteristic that can be computed from statically analyzing the source code. Figure 1 presents an example of Software Quality Characteristic measurement framework suggested in ISO/IEC 25010 and ISO/IEC 15939 using a partial decomposition for Maintainability.

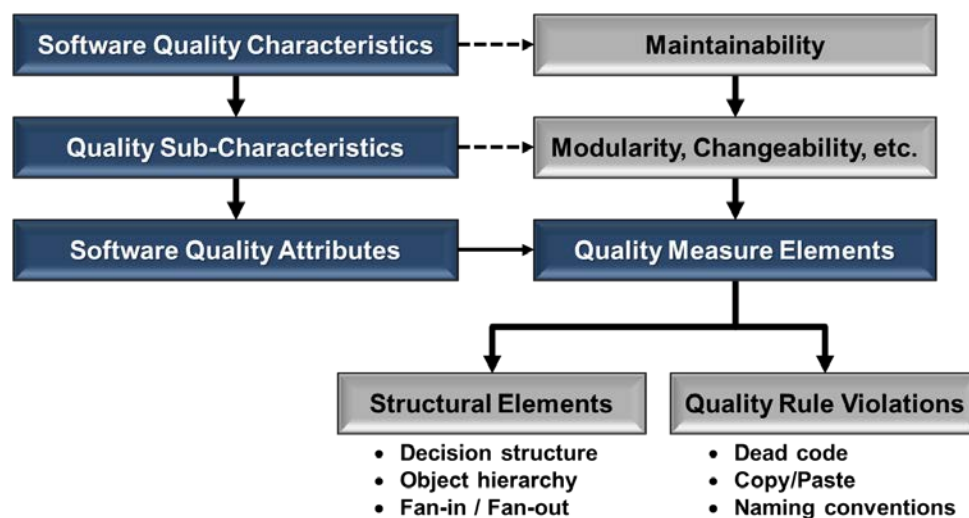


Figure 1. ISO/IEC 25010 & 15939 Framework for Software Quality Characteristics Measurement

Items in the blue boxes in Figure 1 represent the elements of the measurement framework in ISO/IEC 25020 and 15939. Items in the gray boxes are the example instantiations of these framework elements for Quality Characteristic of Maintainability. In particular, the Software Quality Attributes of ISO/IEC 25010 correspond to the Quality Measure Elements in ISO/IEC 15939. Throughout this specification we will refer to the countable structural measure elements and Quality Rule violations as Quality Measure Elements. Scores for individual Quality Measure Elements are summed to create the measure for a Quality Characteristic.

1.4 Development of Software Quality Characteristic Measures

Companies interested in joining CISQ held executive forums in Frankfurt, Germany; Arlington, VA; and Bangalore, India to set strategy and direction for the consortium. In these forums four Quality Characteristics were selected as the most important targets for automation—reliability, security, maintainability, and performance efficiency. These targets cover four of the eight Quality Characteristics described in ISO 25010. Figure 2 displays the ISO/IEC 25010 software product quality model with the four Software Quality Characteristics selected as the initial focus for defining automated measures by CISQ highlighted in orange.

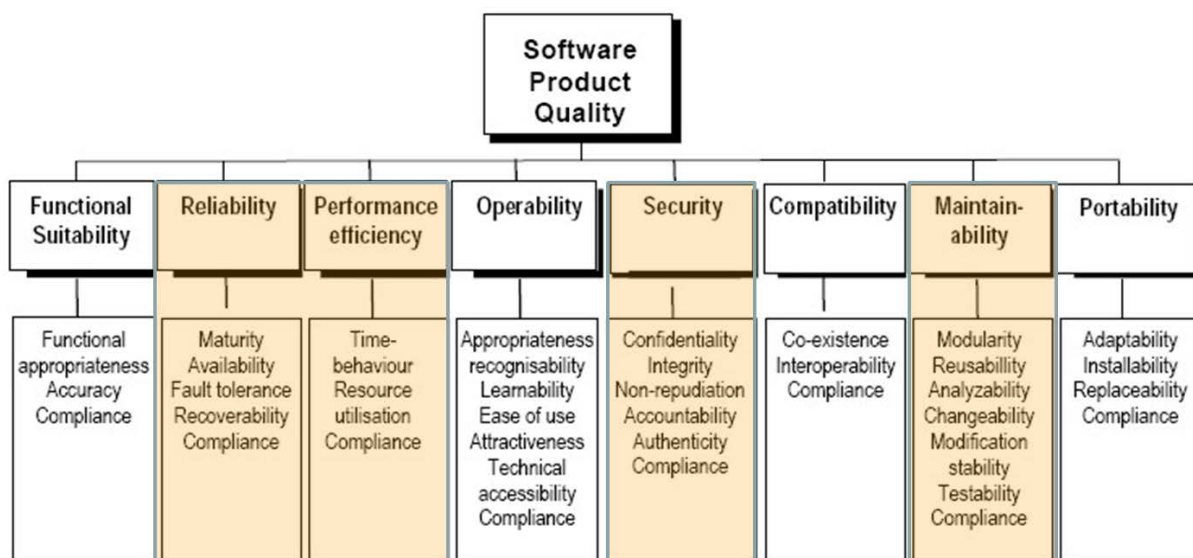


Figure 2. Software Product Quality Characteristics from ISO/IEC 25010.

This document presents the specifications for these four Quality Characteristics resulting from work by consortium members during 2010-2011. This work was initiated by CISQ members developing lists of software quality issues related to each Software Quality Characteristic selected for specification. These issues were then translated into software Quality Rules worded as architectural or coding practices or conventions to avoid the problem described in the software quality issue. The rules were then transformed into software Quality Measure Elements by counting violations of these practices and conventions.

Each of the Software Quality Characteristic measures is computed as counts of the most critical rule violations, that is, the most critical causes of problems within the context of that Quality Characteristic.

The measures are based on rule violations that can be detected through static analysis of the source code. Quality Rules can have language or platform-specific variations as necessary. An example of quality issues and their associated rules is contained in the CWE/SANS Institute Top 25 Most Dangerous Software Errors that provides a list of the top 25 security rule violations and associated rules. This list can be found at <http://cwe.mitre.org/top25/#Listing>.

1.5 Rule Violation-Based Measures

The use of rule violations in Quality Characteristic metrics presents several challenges for establishing baselines. Growth in the number of different rule violation types aggregated into a Quality Characteristic measure could continually raise the bar for measuring quality, reducing the validity of baseline comparisons. Further, different vendors will initially have different sets of rule violations they detect, making comparisons difficult across commercial software quality measurement offerings.

One solution to this problem is to create a stable list of rule violations that are used for computing a baseline for each Quality Characteristic. For instance, in security the CWE (Common Weakness Enumeration) Top 25 rule violations could form the basis for a stable baseline measure. For each Quality Characteristic such a list would provide a minimum set of rule violations that must be included in calculating the attribute measure.

Since the impact and frequency of specific rule violations will change over time, this approach allows specific violations to be included, excluded, amplified, or diminished over time in order to promote the most effective benchmarking, diagnostic, and predictive use. The specific rule violations included in each Quality Characteristic measure could change over time, but the set would be controlled in ways that retained the ability to compare baselines. Vendors could compute the standard baseline measure, as well as their own extended measure that included other rule violations for each Quality Characteristic.

2. Terms and Definitions

Effectiveness—the accuracy and completeness with which users achieve specified goals. (ISO 9241-11)

Efficiency—resources expended in relation to the accuracy and completeness with which users achieve goals. (ISO 9241-11)

Internal Software Quality—the degree to which a set of static attributes of a software product satisfy stated and implied needs for the software product to be used under specified conditions. This will be referred to as software structural quality, or simply structural quality in this specification. (ISO 25010)

Maintainability—degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers. (ISO 25010)

Performance Efficiency—performance relative to the amount of resources used under stated conditions. (ISO 25010)

Software Quality Property—measurable component of software quality. (derived from ISO 25010)

Reliability—degree to which a system, product, or component performs specified functions under specified conditions for a specified period of time. (ISO/IEC/IEEE 24765)

Rule Violation—a construction in source code that violates a rule of good architectural and coding practice that based on historical evidence, can cause problems in software development, maintenance, or operations.

Security—degree to which a product or system protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization. (ISO 25010)

Software Product—a set of computer programs, procedures, and possibly, associated documentation and data. (ISO 25010)

Software Product Quality Model—a model that categorizes product quality properties into eight characteristics (functional suitability, reliability, performance efficiency, usability, security, compatibility, maintainability and portability). Each characteristic is composed of a set of related sub-characteristics. (ISO 25010)

Software Quality—degree to which a software product satisfies stated and implied needs when used under specified conditions. (ISO 25010)

Software Quality Attribute—an inherent property or characteristic of software that can be distinguished quantitatively or qualitatively by human or automated means. (derived from ISO 25010)

Software Quality Characteristic—a category of software quality attributes that bears on software quality. (ISO 25010)

Software Quality Characteristic Measure—a software quality measure derived from measuring the attributes related to a specific Software Quality Characteristic.

Software Quality Issue—architectural or coding practices that are known to cause problems in software development, maintenance, or operations and for which software Quality Rules can be defined that help avoid problems created by the issue.

Software Quality Measure—a measure that is defined as a measurement function of two or more values of software Quality Measure Elements. (ISO 25010)

Software Quality Measurement—(verb) a set of operations having the object of determining a value of a software quality measure. (ISO 25010)

Software Quality Measure Element—a measure defined in terms of a software quality attribute and the measurement method for quantifying it, including optionally the transformation by a mathematical function. (ISO 25010)

Software Quality Model—a defined set of software characteristics, and of relationships between them, which provides a framework for specifying software quality requirements and evaluating the quality of a software product. (derived from ISO 25010)

Software Quality Rule—an architectural or coding practice or convention that represents good software engineering practice and avoids problems in software development, maintenance, or operations.

Software Quality Sub-characteristic—a sub-category of a Software Quality Characteristic to which software quality attributes and their software Quality Measure Elements are conceptually related. (derived from ISO 25010)

Structural Element—a component of software code that can be uniquely identified and counted such as a token, decision, variable, etc.

Structural Quality—the degree to which a set of static attributes of a software product satisfy stated and implied needs for the software product to be used under specified conditions—a component of software quality. This concept is referred to as internal software quality in ISO 25010.

3. Compliance

3.1 Compliance

Implementations of this specification should be able to demonstrate the following attributes in order to claim compliance.

- **Automated**—although some inputs may need to be provided manually to initiate and support identification of a few rule violations, the analysis of the source code and the actual counting must be fully automated. These initial inputs may include the source code of the application and how boundaries between application layers or tiers are defined.
- **Objective**—Two independent analyses of the same application must produce the same counts for each of the Quality Measure Elements measured as part of a Software Quality Characteristic.
- **Transparent**—Implementations that comply with this specification must clearly list each and every input the implementation requires and list each and every output that the implementation generates. Implementations of this specification are encouraged to provide a list of inputs and outputs at interim stages of the analysis process.
- **Verifiable**—Compliance with this specification requires that an implementation state the assumptions/heuristics it uses with sufficient detail so that the calculations may be independently verified by third parties. In addition, all inputs used are required to be clearly described and itemized so that they can be audited by a third party.

3.2 Required Inputs

The following inputs are needed by static code analyzers in order to interpret violations of the software Quality Rules that would be included in individual software Quality Measure Elements.

- The entire source code for the application being analyzed
- Documentation or information about how the layers of tiers in the design of the application and how functions are allocated to them
- Information about whether the design is based on DOM or SAX in order to evaluate multiple inheritance mechanisms
- A list of vetted libraries are being used to "neutralize" input data
- What routines / API calls are being used for remote authentication, to any custom initialization and cleanup routines, to synchronize resources, or to neutralize accepted file types or the names of resources
- The encryption algorithms that are being used

Static code analyzers will also need a list of the rule violations grouped by Software Quality Characteristic categories that constitute each software Quality Measure Element. These rule violations are constitute violations of software Quality Rules and are counted in the measures listed in Tables 1-4 below.

4. CISQ Quality Characteristic Measures

4.1 Structure of Quality Characteristic Measures

Each of the four CISQ Quality Characteristic measures will be constructed from a set of Quality Measure Elements whose occurrence violates a Quality Rule. Each Quality Rule was selected because it addressed a significant Quality Issue within the domain of the CISQ Quality Characteristic. The content of each CISQ Quality Characteristic will consist of this hierarchy of Quality Issues that are decomposed into Quality Rules that are quantified by Quality Measure Elements. Quality Measure Elements are aggregated in the manner presented in Section 5 to create a measure of each CISQ Quality Characteristic. The components used in the presentation of each CISQ Quality Characteristic adhere to the following definitions.

- **Quality Issue**—a high level description of a cause of problems within the domain covered by the Quality Characteristic.
- **Quality Rule**—a specific guideline for avoiding violations of good architectural or coding practice. Rules are defined at a level that can be detected through static analysis of the code. Violations of the rule are a contributors to the problem expressed in the quality issue.
- **Quality Measure Element**—specifies how structural elements or rule violations can be quantified to assess the extent to which a rule has been violated.

4.2 Source of Quality Measure Elements

Quality Measure Elements were proposed by representatives of the CISQ member companies and evaluated on the severity of their impact. *Severity* indicates the level of harm caused by or the potential risk created by violations of each Quality Rule. A severe violation of a Quality Rule will manifest its impact through the quality issue it creates in maintenance or operations.

The final set of Quality Measure Elements for each Quality Characteristic were selected because their impact was judged to have a high enough severity rating that the violation of its Quality Rule must be remediated. Representatives for the participating member companies evaluated the candidate Quality Measure Elements and voted on the elements they felt were severe enough to include in the final measure. The final measure for each Quality Characteristic aggregates between 16 and 30 Quality Measure Elements.

4.3 CISQ Reliability Measure

4.3.1 Definition of Reliability

According to ISO/IEC 25010, Reliability concerns “the degree to which a system or component performs its required functions under stated conditions for a specified period of time.” This definition is consistent with ISO/IEC/IEEE 24765-2010 which provides a common vocabulary for software and systems engineering. In ISO/IEC 25010 Reliability consists of the following sub-characteristics.

- **Maturity**—the probability of executing faults in the software.
- **Availability**—the degree to which a system or component is operational and accessible when required for use.
- **Fault tolerance**—the degree to which a system or component continues normal operation despite the presence of hardware or software faults.
- **Recoverability**—the degree to which the product can recover the data directly affected and reestablish the desired state of the system in the case of an interruption or a failure.
- **Reliability Compliance**—the degree to which the product adheres to standards, conventions, or regulations in laws and similar prescriptions relating to reliability.

4.3.2 CISQ Reliability Measure Elements

The CISQ Reliability measure is composed from 30 Quality Measure Elements that quantify violations of 28 Quality Rules affecting 10 Quality Issues related to Reliability. Table 1 enumerates these Quality Measure Elements, Rules, and Issues.

Table 1. Reliability Issues, Rules, and Measure Elements

Quality Issue	Quality Rule	Quality Measure Element
Issue 1: Inconsistent or incomplete handling of errors and exceptions leads to inaccurate identification and inadequate response to errors	Rule 1: Exception handling blocks such as Catch and Finally blocks must not be empty.	Measure 1: # of exception handling blocks such as Catch and Finally blocks that are empty Measure 2: # of generic exceptions thrown and caught
	Rule 2: Methods, procedures and functions doing Insert, Update, Delete, Create Table or Select must include error management (check of database error variables or exception handling).	Measure 3: # of functions doing Insert, Update, Delete, Create Table, and Select that do not include error management capabilities
Issue 2: Some coding weaknesses result in unexpected and faulty behaviors	Rule 3: Classes that implement a serializable interface must also implement a serializable method and subfields within the object that are serializable.	Measure 4: # of classes that implement a serializable interface must also implement a serializable method Measure 5: # of classes that have

		fields that are not serializable if the class is declared as serializable
	Rule 4: Persistent classes should implement hashCode() and equals()	Measure 6: # of classes declared as persistent that do not implement both hashCode() and equals ()
	Rule 5: Application software running on servers should not duplicate functionality that is being provided by the application server (e.g., creating threads inside a J2E framework)	Measure 7: # of invocations of thread creating functions, sockets, class loaders, and Java I/O in application code using J2E framework
	Rule 6: Classes that contain pointers must implement their own copy method	Measure 8: # of classes with pointers that do not implement their copy method
	Rule 7: Supply an initial value for all non-static variables	Measure 9: # of non-static variables that do not supply an initial value
	Rule 8: Avoid deleting instances from inside the instance	Measure 10: # of 'Delete This' commands
	Rule 9: Type casting should only be performed between compatible types	Measure 11: # of type casting between incompatible types
	Rule 10: When moving in-memory data never truncate it, but ensure that the source and the destination have the compatible sizes	Measure 12: # of functions that move in memory data between buffers of incompatible sizes
	Rule 11: No function should have a variable number of parameters.	Measure 13: # of functions that have a variable number of parameters
	Rule 12: All resource allocation statements should be followed by a test of the return value. Resource allocation statements include get memory, get thread, get db connection, and open file.	Measure 14: # of resources allocation statements not followed by a test of the return value
	Rule 13: Do not test floating point numbers for equality.	Measure 15: # of floating values tested for equality

Issue 3: Software that is unaware of resource bounds or fails to monitor resources risks exceeding resource and capacity limits	Rule 14: Ensure that whenever a function opens a resource, it is always de-allocated or closed.	Measure 16: # of functions that open resources that do not close those resources.
	Rule 15: Ensure references to buffers are within allocated buffer size	Measure 17: # references to buffers that exceed allocated buffer sizes
Issue 4: Failure to manage data integrity and consistency can lead to unexpected behavior	Rule 16: All data access goes through a central data manager (transaction manager)	Measure 18: # of data accesses that do not go through a central data manager.
Issue 5: Components used in multi-thread environments that do not protect their state can experience deadlock or livelock	Rule 17: Multi-thread functions should be made thread safe, for instance, servlets or struts action classes must not that have instance/non-final static fields	Measure 19: # of multi-thread functions that do not have non-final static fields
	Rule 18: When implementing the singleton pattern, ensure that the locks are granted before instantiating the singleton	Measure 20: # of singleton patterns that are instantiated without prior locking
	Rule 19: Avoid cyclic calls between packages that create hang ups through mutual dependencies that cannot be resolved (A calls B and B calls A)	Measure 21: # of cyclic calls between packages

Issue 6: When inheritance and polymorphism are not implemented in a safe way, unexpected behaviors can result	Rule 20: Superclass must not be using Subclass. A Superclass is not allowed to have knowledge of one of her Subclasses. The Superclass has knowledge of the Subclass if the Superclass directly calls a Subclass-method, uses a Subclass-attribute or refers to the name of the Subclass.	Measure 22: # of super-classes calling subclasses
	Rule 21: In languages where custom destructors can be written, Classes with at least one virtual Method must have a virtual Destructor	Measure 23: In languages where custom destructors can be written, # of classes with at least one virtual Method without a virtual Destructor
	Rule 22: In languages where custom destructors can be written, base Classes must have virtual Destructors	Measure 24: In languages where custom destructors can be written, # of base Classes without virtual Destructors
	Rule 23: In languages where custom destructors can be written, whenever base classes have a virtual destructor, subclasses must also implement virtual destructors	Measure 25: In languages where custom destructors can be written, # of subclasses whose base class has a virtual destructor, that lack a virtual destructor
Issue 7: Complex code complicates the testing process	Rule 24: Too many parameters to perform unit testing and achieve a good coverage ratio	Measure 26: # of functions containing parameters ≥ 7
	Rule 25: All modules have an acceptable cyclomatic complexity	Measure 27: # of modules have cyclomatic complexity \geq a language-specific value (Table to be inserted)
Issue 8: Built-in remote addresses cause problems when the target is moved	Rule 26: Avoid of hard-coded network resources (e.g., IP addresses, URLs, etc.)	Measure 28: # of hard-coded network resources such as IP addresses, URLs, etc.
Issue 9: Resources that are allocated must be freed to avoid resource exhaustion	Rule 27: Modules where resources are allocated must have associated free statements.	Measure 29: # of functions where resources are allocated without having associated free statements
Issue 10: Blocking calls can result in system failure if the called process fails	Rule 28: Blocking synchronous calls should have associated timeouts	Measure 30: # of synchronous calls that lack associated timeouts

4.4 Performance Efficiency Measure

4.4.1 Definition of Performance Efficiency

According to ISO/IEC 25010, Performance Efficiency concerns “the performance relative to the amount of resources used under stated conditions for a specified period of time.” In ISO/IEC 25010 Performance Efficiency consists of the following sub-characteristics.

- **Time Behavior**—the response and processing times and throughput rates when performing its function under stated conditions.
- **Resource Utilization**—the amounts and types of resources used when the software performs its function under stated conditions.
- **Performance Efficiency Compliance**—the degree to which the product adheres to standards or conventions relating to performance efficiency.

4.4.2 CISQ Performance Efficiency Measure Elements

The CISQ Performance Efficiency measure is composed from 16 Quality Measure Elements that quantify violations of 15 Quality Rules affecting 6 Quality Issues related to Performance Efficiency. Table 2 enumerates these Quality Measure Elements, Rules, and Issues.

Table 2. Performance Efficiency Issues, Rules, and Measure Elements

Issue	Quality Rule	Quality Measure Element
Issue 1: In a local versus remote environment, software that does not maintain redundancy of data (cache) and code increases the time with which they are accessed	Rule 1: Software centralizes client requests (incoming and data) to reduce network traffic	Measure 1: # of instances in which client requests are not handled thorough a central mechanism
Issue 2: Some SQL Query and Data Access constructs cause unnecessary resource consumption	Rule 2: Avoid SQL queries that require sequential searches	Measure 2: # of SELECTS done through sequential searches
	Rule 3: Avoid overly complex queries on very large tables (e.g., joins too many tables, has too many sub-queries, inefficient order of joins)	Measure 3: # of complex queries on very large tables, where complex query = ≥ 5 joins, sub-queries ≥ 3 , (check on order of joins) (check on scoring for total of the above to determine a complexity threshold)
	Rule 4: Avoid overly large indices and too many indices per table on very large tables	Measure 4: # of indices in very large tables with range ³ 10 Measure 5: # of very large tables with ³ 3 indices

Issue 3: Poorly designed data structures and algorithms can increase computation and network traffic	Rule 5: Choose the right parser when memory is constrained (e.g., avoid use of DOM; use alternatives such as SAX)	Measure 6: ratio of # child objects of DOM compared to child objects of SAX ($DOM / SAX < 1.0$)
Issue 4: Expensive computations in loops or other repetitive operations increase the computational load especially as the data or usage grows	Rule 6: Avoid placing expensive operations inside a loop (e.g., OPEN/CLOSE, instantiation or initialization of objects, SQL queries, cursors, initialization of database connections inside a loop, remote calls)	Measure 7: # of loops with expensive operations
	Rule 7: Avoid instantiations inside static blocks - use lazy initialization instead	Measure 8: # of initializations inside static blocks
Issue 5: Failure to use database stored procedures or database functions versus middle tier components reduces performance	Rule 8: # of SQLs in middle tier should not be > 2	Measure 9: # of non-database functions with SQLs > 2
	Rule 9: # of SQLs > 2 should be in stored procedures	Measure 10: # of SQLs in the function > 5 not in stored procedures
Issue 6: Poor coding practices can cause unnecessary use of resources	Rule 10: Avoid unmatched use of allocation and de-allocation of memory for objects, instead, CHAR/VARCHAR in database, system.out.print, etc.)	Measure 11: # of unmatched allocation/de-allocations of memory for objects
	Rule 11: Avoid creation of additional immutable objects (e.g., strings during concatenation in JAVA--use string buffers	Measure 12: # of additional immutable objects
	Rule 12: Avoid keeping reference to objects that are of no use, then garbage collection will not be able to identify the object for garbage collection	Measure 13: # of object references that lack a destructor/finalize function
	Rule 13: Avoid heavy objects in session for better user experience	Measure 14: # objects having aggregation of > 5 objects
	Rule 14: Avoid using static variables/objects (if used, should be used as singleton, and in multi-threaded environments)	Measure 15: # of static variables/collections/objects that are not singletons
	Rule 15: Avoid static connections rather than use of connection pools	Measure 16: # of static connections

4.5 CISQ Security Measure

4.5.1 Definition of Security

According to ISO/IEC 25010, Security concerns “the degree of protection of information and data so that unauthorized persons or systems cannot read or modify them and authorized persons or systems are not denied access to them.” This definition is consistent with ISO/IEC 12207-2008 Systems and Software Engineering—Software Lifecycle Processes. In ISO/IEC 25010 Security consists of the following sub-characteristics.

- **Confidentiality**—the degree or protection from unauthorized disclosure of data or information, whether accidental or deliberate.
- **Integrity**—the degree to which a system or component prevents unauthorized access to, or modification of, computer programs or data.
- **Non-Repudiation**—the degree to which actions or events can be proven to have taken place, so that the events or actions cannot be repudiated later.
- **Accountability**—the degree to which the actions of an entity can be traced to an entity.
- **Authenticity**—the degree to which the identity of a subject or resource can be proved to be the one claimed.
- **Security Compliance**—the degree to which the product adheres to standards, conventions, or regulations in laws and similar prescriptions relating to security.

4.5.2 CISQ Security Measure Elements

The CISQ Security measure is composed from 21 Quality Measure Elements that quantify violations of 20 Quality Rules affecting 19 Quality Issues related to Security. The 19 Quality Issues are taken from the top 25 issues of the Common Weakness Enumeration (CWE). Six issues of the top 25 were not included because they were not defined in a way that Quality Measure Elements related to these CWEs could be detected in the source code. Table 3 enumerates these Quality Measure Elements, Rules, and Issues. Quality Issues related to Security are labeled with their CWE identifier.

Table 3. Security Issues, Rules, and Measure Elements

Issue	Quality Rule	Quality Measure Element
CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	Rule 1: Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid, such as Microsoft's Anti-XSS library, the OWASP ESAPI Encoding module, and Apache Wicket.	Measure 1: # of instances where output is not using library for neutralization

CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	Rule 2: Use a vetted library or framework that does not allow SQL injection to occur or provides constructs that make this SQL injection easier to avoid or use persistence layers such as Hibernate or Enterprise Java Beans.	Measure 2: # of instances where data is included in SQL statements that is not passed through the neutralization routines.
CWE-22: Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	Rule 3: Use a vetted library or framework that does not allow path traversal to occur or provides constructs that make it easier to avoid.	Measure 3: # of path manipulation calls without validation mechanism
CWE-434: Unrestricted Upload of File with Dangerous Type	Rule 4: Assume all input is malicious. Use an "accept known good" input validation strategy, i.e., use a whitelist of all file types that the system can safely accept. Reject any input that does not strictly conform to specifications, or transform it into something that does. When performing input validation, consider all potentially relevant properties, including length of files, type of input files, the full range of acceptable values, missing or extra extensions, syntax, and conformance to business rules (just because a file type is OK in one context, does not mean it will be ok in all contexts throughout the system). For example, limiting filenames to alphanumeric characters can help to restrict the introduction of unintended file extensions.	Measure 4: # of upload opportunities not passed to sanitization calls Note: This might be very hard to do automatically - detection may be incomplete

CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	Rule 5: Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid. For example, consider using the ESAPI Encoding control or a similar tool, library, or framework. These will help the programmer encode outputs in a manner less prone to error.	Measure 5: # of shell statements or OS calls executed by the system without proper neutralization routines
CWE-798: Use of Hard-coded Credentials	Rule 6: For outbound authentication: store passwords, keys, and other credentials outside of the code in a strongly-protected, encrypted configuration file or database that is protected from access by all outsiders, including other local users on the same system. Properly protect the key (CWE-320). If you cannot use encryption to protect the file, then make sure that the permissions are as restrictive as possible. In Windows environments, the Encrypted File System (EFS) may provide some protection.	Measure 6: # of remote authentication calls that use literal or fixed values as a user name or password.
CWE-706: Use of Incorrectly-Resolved Name or Reference (also covers CWE-98 Improper Control of Filename for Include/Require Statement in PHP Program ('PHP File Inclusion'))	Rule 7: Use a vetted library or framework that does not allow user input to determine the names of resources to be used for execution or provide constructs that make it easier to avoid.	Measure 7: # of names with user input that aren't validated / number of names with user input

CWE-129: Improper Validation of Array Index	Rule 8: Assume all input is malicious. When accessing a user-controlled array index, use a stringent range of values that are within the target array. Make sure that you do not allow negative values to be used. That is, verify the minimum as well as the maximum of the range of acceptable values.	Measure 8: # of array accesses with user input that IS NOT range checked
CWE-754: Improper Check for Unusual or Exceptional Conditions	<p>Rule 9: Check the results of all functions involving system resources that return a value and verify that the value is expected. Notes: Checking the return value of the function will typically be sufficient, however beware of race conditions (CWE-362) in a concurrent environment. Rule 2: If using exception handling, catch and throw specific exceptions instead of overly-general exceptions (CWE-396, CWE-397). Catch and handle exceptions as locally as possible so that exceptions do not propagate too far up the call stack (CWE-705). Avoid unchecked or uncaught exceptions where feasible (CWE-248). Notes: Using specific exceptions, and ensuring that exceptions are checked, helps programmers to anticipate and appropriately handle many unusual events that could occur.</p>	<p>Measure 9: # of function calls involving system resources that do not check return values</p> <p>Measure 10a: # of overly broad exceptions thrown. This will require language specific analysis of potential exceptions</p> <p>Measure 10b: # of overly broad exceptions caught. This will require language specific analysis of potential exceptions</p>
CWE-131: Incorrect Calculation of Buffer Size	Rule 10: Perform input validation on any numeric input by ensuring that it is within the expected range. Enforce that the input meets both the minimum and maximum requirements for the expected range.	Measure 11: # of allocations with tainted input AND no range check

CWE-327: Use of a Broken or Risky Cryptographic Algorithm	Rule 11: Select a well-vetted algorithm that is currently considered to be strong by experts in the field, and select well-tested implementations. As with all cryptographic mechanisms, the source code should be available for analysis. For example, US government systems require FIPS 140-2 certification.	Measure 12: Determine the version and type of libraries being used, and verify that they are well vetted implementations and are up to date. For example, FIPS 140-2 has a list of validated implementations. This metric is binary (uses a good algorithm or not)
CWE-134: Uncontrolled Format String	Rule 12: Ensure that all format string functions are passed a static string which cannot be controlled by the user and that the proper number of arguments is always sent to that function as well. If at all possible, use functions that do not support the %n operator in format strings.	Measure 13: #format strings with user input
CWE-456: Missing Initialization	Rule 13: Supply an initial value for all non-static variables	Measure 14: # of non-static variables that do not supply an initial value
CWE-672: Operation on a Resource after Expiration or Release	Rule 14: Once resources have been released, references to the resource should be cleared and they should not be accessed again.	Measure 15: # of resources used after they are released (free, file close, socket close, etc.)
CWE-834: Excessive Iterations	Rule 15: Do not use user-controlled data for loop conditions. Rule 16: Limit the number of recursive calls to a reasonable number.	Measure 16: # loop conditions that are specified by a user without some kind of range check or neutralization process Measure 17: # of recursive functions that do not move toward a base case on each call
CWE-681: Incorrect Conversion between Numeric Types	Rule 17: Type casting should only be performed between compatible types	Measure 18: # of type casting between incompatible types
CWE-667: Improper Locking	Rule 18: Use industry standard APIs to implement locking mechanism.	Measure 19: # of shared resources accessed without synchronization in concurrent context
CWE-772: Missing Release of Resource after Effective Lifetime	Rule 19: When the software no longer needs a resource, such as a file, network connection, or memory, it should be released back to the system.	Measure 20: # of resources allocated and not released within the same module

CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer	Rule 20: When moving in-memory data never exceed the bounds of the buffer and ensure that the source and the destination have compatible sizes	Measure 21: # of functions that move in-memory data between buffers of incompatible sizes
---	---	--

4.6 CISQ Maintainability Measure

4.6.1 Definition of Maintainability

According to ISO/IEC 25010, Maintainability concerns “the degree to which the product can be modified.” This definition is consistent with ISO/IEC/IEEE 24765-2010 which provides a common vocabulary for software and systems engineering. In ISO/IEC 25010 Reliability consists of the following sub-characteristics.

- **Modularity**—the probability of executing faults in the software.
- **Availability**—the degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components.
- **Reusability**—the degree to which an asset can be used more than once in a software system, or in building other assets.
- **Analyzability**—the ease with which the impact of an intended change on the rest of the software can be assessed, or the software product can be diagnosed for deficiencies or causes of failures in the software, or the parts to be modified can be identified.
- **Changeability**—the degree to which an asset can be used more than once in a software system, or in building other assets.
- **Modification Stability**—the degree to which an asset can be used more than once in a software system, or in building other assets.
- **Reliability Compliance**—the degree to which the product can avoid unexpected affects from modifications of the software.

4.6.2 CISQ Maintainability Measure Elements

The CISQ Reliability measure is composed from 21 Quality Measure Elements that quantify violations of 18 Quality Rules affecting 7 Quality Issues related to Maintainability. Table 4 enumerates these Quality Measure Elements, Rules, and Issues.

Table 4. Maintainability Issues, Rules, and Measure Elements

Issue	Quality Rule	Quality Measure Element
Issue 1: In a layered architecture functions should be strictly allocated to layers and maintain a strict hierarchy of calling between layers (utility layer excepted)	Rule 1: Functions only communicate (exchange data) with functions belonging to an adjacent layer. Functions do not directly exchange data with functions that are not in adjacent layers (no layer skipping/bridging).	Measure 1: # functions that span layers Measure 2: # of layer-skipping calls
	Rule 2: Avoid too many horizontal layers	Measure 3: # of layers (threshold $4 \leq \# \text{ Layers} \leq 8$)

Issue 2: Duplicated code requires changes be made to all instances creating greater opportunity for error and more effort for maintenance.	Rule 3: Long code segments should only exist in one place (control or eliminate copy-paste) to avoid code duplication	Measure 4: # files that contain 100+ consecutive duplicate tokens Measure 5: # of unreachable functions
	Rule 4: limit the inheritance depth of a class	Measure 6: # of classes with inheritance levels ≥ 7
	Rule 5: limit the number of children of a class	Measure 7: # of classes with ≥ 10 children
	Rule 6: avoid multiple inheritance	Measure 8: # of instances of multiple inheritance of concrete implementation classes (threshold > 1)
Issue 3: Data access encapsulation (Proportion of public to private variables) to avoid accidental corruption of values	Rule 7: Data update and insert should be encapsulated	Measure 9: # of methods that are directly using fields from other classes
	Rule 8: Avoid declaring data members public	Measure 10: # of variables declared public
Issue 4: Highly and tightly coupled modules (fan-out, intermediaries,) cause excessive propagation of modifications	Rule 9: Limit the fan-out on the class (Threshold value should be configured ≤ 5)	Measure 11: # of functions that have a fan-out ≥ 10
	Rule 11: Limit the tightness with which functions are coupled with other functions ($X < Y$)	Measure 12: # of objects with coupling > 7
	Rule 10: Avoid cyclic calls between packages that create hang ups through mutual dependencies that cannot be resolved (A calls B and B calls A)	Measure 13: # of cyclic calls between packages
	Rule 11: No commented out instructions	Measure 14: # of functions with > 2% commented out instructions
	Rule 12: No file over 1000 lines	Measure 15: # files > 1000 LOC
	Rule 13: No modification of indices within loops	Measure 16: # of instances of indexes modified within its loop
Issue 5: Code that does not follow the principles of structured programming (essential complexity, branching conventions, cyclomatic complexity, etc.) degrades program comprehensibility	Rule 14: Don't use GO TOs, CONTINUE, and BREAK outside the switch	Measure 17: # of GO TOs, CONTINUE, and BREAK outside the switch
	Rule 15: Limit Cyclomatic Complexity	Measure 18: # functions with cyclomatic complexity \geq a language specific threshold (table to be inserted)
	Rule 16: Check complexity based on number of Database/file operations	Measure 19: # of methods with ≥ 7 data or file operations

Issue 6: Over-parameterization of methods makes programs less comprehensible	Rule 17: The number of parameters passed by a functions should be less than seven	Measure 20: # functions passing ≥ 7 parameters
Issue 7: Hard coding of literals, etc. makes program modification more difficult.	Rule 18: Do not hard code literals other than trivial ones	Measure 21: # of hard coded literals except (-1, 0, 1, 2, or literals initializing static or constant variables)

5. Software Quality Characteristic Calculation Process

5.1 Weighting Schemes

Violations of Quality Rules for each Quality Characteristic will be aggregated into a measure using scoring rules that are common to all Quality Characteristic measures. A count of total violations of Quality Rules was selected as the best alternative for measurement. Scoring at the component level was rejected because many critical violations of Quality Rules cannot be isolated to a single component, but rather involve interactions among several components often in different layers of an application.

Several weighting schemes were considered for scoring individual rule violations, some based on the cost to remediate violations and others based on severity of the business impact. Normalization techniques based on software product size were also considered as a way to enable better comparisons for benchmarking.

After extensive discussion among the participating member companies it was decided that no weighting scheme would be proposed as part of this specification since different schemes have different uses. There are too many different weighting schemes that have valid uses to include them in this specification. In addition, the rule violations included in each Quality Characteristic in this specification were all required to be high impact violations that would need to be remediated. Therefore, weightings based on severity would add little useful information to the measure since the variance among weights would be small.

Vendors may recalculate these Quality Characteristic measures using any weighting scheme they choose provided they also report the raw scores that are the subject of this specification. This provides the maximum tradeoff between broad comparability of results and the provision of unique insight that comes with a tailored weighting scheme. CISQ will produce future documents describing alternative weighting schemes that could be used in Quality Characteristic measures.

5.2 Calculation Formula

The formula for calculating each CISQ Quality Characteristic score is:

$$QC_j = \sum_{k=1}^n (\text{Quality Measure Element}_{jk})$$

QC_j = the Quality Characteristic score for one of the four Quality Characteristics (Reliability, Performance Efficiency, Security, and Maintainability). The number of rule violations for each Quality Measure Element forms the score for that Element, and these scores are summed to calculate the measure for the CISQ Quality Characteristic.

6. References

- Boehm, B.W., Brown, J.R., Kaspar, H., Lipow, M., MacCleod, G.J., & Merrit, M.J. (1978). *Characteristics of Software Quality*. Amsterdam: North Holland.
- Chidamber, S.R. & Kemerer, C.F. (1994). A metrics suite for object-oriented design. *IEEE Transactions on Software Engineering*, 20 (6), 476-493.
- Consortium for IT Software Quality (2010). <http://it-cisq.org/>
- Curtis, B. (1980). Measurement and experimentation in software engineering. *Proceedings of the IEEE*, 68 (9), 1103-1119.
- Gilb, T. (1976). *Software Metrics*. Winthrop.
- Halstead, M.E. (1976). *Elements of Software Science*. Amsterdam: Elsevier North Holland.
- Henry, S. & Kafura, D. (1981). Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, 7 (5), 510-518.
- International Organization for Standards (2007). *ISO/IEC 15939: Software Engineering - Software Measurement Process*. Geneva: ISO.
- International Organization for Standards (2009). *ISO/IEC 25010.3 Software Engineering - Software Product Quality Requirements and Evaluation (SQuaRE) Quality Model*. Geneva: ISO.
- McCabe, T.J. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, 2 (4), 308-320.
- McCall, J.A., Richards, P.K., & Walters, G.F. (1977). *Factors in Software Quality – Volumes I, II, & III*. NTIS AD/A-049 014, 015, 055.