TECHNICAL DOCUMENT 3304
April 2016

Establishing Qualitative Software Metrics in Department of the Navy Programs

Christopher E. Johnson

SSC Pacific
San Diego, CA 92152-5001

SSC Pacific
San Diego, California 92152-5001

K. J. Rothenhaus, CAPT, USN
Commanding Officer

C. A. Keeney
Executive Director

ADMINISTRATIVE INFORMATION

The work described in this report was performed during Fiscal Year 2015 by the Command Intelligence Systems Division (Code 53200), Space and Naval Warfare Systems Center Pacific (SSC Pacific), San Diego, CA. The Naval Innovative Science and Engineering (NISE) Program at SSC Pacific provided funding for this Applied Research project.

Released under authority of J.
J. M. Simonetti, Head
Command Intelligence
Systems Division

SB

CONTENTS

Figure

Tables

The Department of the Navy is dedicated to provide the highest quality software to its users. In accomplishing this goal, a need exists for a formalized set of software quality metrics. This document establishes the validity of those necessary quality metrics. In our approach, we collected the data of more than a dozen programs from previous tests, analyzed current states of the software, derived formulas by weighting to provide necessary results, investigated tool sets to provide the necessary variable data for our formulas, and tested the formulas for validity.

PURPOSE

Space and Naval Warfare Systems Center Pacific (SSC Pacific) seeks to establish and provide a set of software quality metrics, measured from common static code analysis tools, which the Department of the Navy can use to measure quality. These metrics provide quality and maturity data through all stages of software development to further ensure that the software delivered meets government-specific requirements. Carefully chosen metrics can direct attention to problems, providing diagnostic value and influence developers' behavior, and offset post-delivery maintenance costs.

SOFTWARE QUALITY CHARACTERISTICS

Software developers can use common static code analysis tools to obtain various measurable metrics of various categories from every software component. This document identifies software qualities and their indicators that affect DoD 5000.02 program areas, primarily cost, schedule, and risk. For software quality measures, the following abilities associated with any software are considered:

- Reusability
- Portability
- Maintainability
- Security
- Extensibility
- Reliability
- Testability
- Scalability

SOFTWARE QUALITY MEASURMENT

Table 1 defines a matrix for determining a score for reusability. It is an example for the other abilities measured in this document. The columns in the tables represent the software attributes our research proved as the most relevant to determine quality for software we acquire. The rows in the table define a range of values to score the software. The project team selected these attributes based on various documents, studies, academic research, industry findings, and empirical data of locally developed programs as listed in [1–24].

The test looked at over 40 software applications from more than a dozen different developers, including government and contractor. Sizes of the applications varied in source lines of code (SLOC), modules, complexity, dependencies, and program languages. We reviewed 15 tools in our current laboratory to determine the best-fit tool for the measures and attributes tested. All the applications selected were previously tested, and in many cases, operationally fielded. From operational use, empirical data was available to support a familiarity of the actual quality of the software prior to the test. This experience enabled us to

refine the formulas through a series of test, formula refinement, and retest, to adjust the formulas and weighting and provide us the expected results.

After determining the tools, we tested software applications to generate the necessary quality attribute data. The data provided from the tools enabled us to create the formulas and weighting necessary to achieve overall qualitative measurement of software.

To achieve the overall score of a particular ability, we selected the combined measures from the table. The corresponding Grade 1–5 was selected for each attribute. The grade number for each attribute was then multiplied by the corresponding weighting in the formula. The numbers for each attribute were then added to arrive at a final score. That score, using the same overall grade as the individual attributes, was used to determine the overall quality.

REUSABILITY

Software abstractness drives reusability. Abstract software can be inherited, which allows for increased reuse. In addition to abstract software, modularity improves software reuse because smaller, more abstract components can be reused and put together like LEGO® blocks to create new functionality.

The formula provides heavier weighting on abstractness (0.5) over the other contributing variables in the formula. Modularity provides the next higher weighting *N* (0.3), which accounts for the sizes of the modules and number of modules used for the application. For this measure, smaller module sizes with more modules are preferred, and provide the associated formula weighting. Complexity and architecture provide the final attribute measures, and are weighted identical based on the ability once module sizes are reduced, best engineering process would dictate decreasing the complexity of the modules and help in achieving the modularity values desired.

Related Metrics:

- Modularity
  - Number of module score
  - Module size score
- Abstractness
  - Weighted Methods per Class (WMC)
  - Number of Children per Class (NOC)
- Complexity
  - Cyclamate Complexity
  - Coupling
- Open Architecture Assessment
  - Use open architecture = 1 if not 0

Table 1. Reusability score matrix.

Grade		Modules			Abstractness		Complexity
		Number of Modules	Coupling (%)	Size	WMC	NOC (%)	Cyclomatic Complexity
1	Very good	> 20	< 50	0–200	1–2	> 25	< 3
2	Good	15–20	51–60	201–300	3–5	20–25	3–6
3	Fair	11–15	61–75	301–400	6–9	10–20	7–10
4	Needs improvement	5–10	76–90	401–500	10–14	5–10	11–14
5	Poor	1–5	> 90	> 500	> 14	< 5	15–20

Modularity Calculation:

$Mo = (Mn \times .5) + (Ms \times .5)$

Abstractness Calculation:

$Ab = (W \times .5) + (N \times .5)$

Complexity Calculation:

$Co = (V \times .5) + (Cp \times .5)$

Reusability Calculation

$Re = Mo(.3) + Ab(.5) + Co(.1) + A(.1)$

Re = Reusability
Mo = Modularity
Mn = Number of Modules
Ms = Module Size
Ab = Abstractness
Co = Complexity
V = Cyclomatic Complexity
Cp = Coupling
A = Architecture
W = Weighted Methods per Class
N = Number of Children

PORTABILITY

Software portability entails the ability and effort required to produce a runnable application based on existing source code for a new environment. Software portability depends on the language used, the libraries, the dependency on native system calls, and the assumptions about the underlying hardware, including display, storage space, memory availability, and

permissions. Measuring portability is not a simple task. The portability metric is useful, but it is critical to first review the software architecture to determine the availability of dependent libraries as well as hardware assumptions. The key features for software portability are:

- Use of popular high-level language (not assembly language or platform-specific language)
- Keeping platform-specific code in modules separate from the cross-platform code and bbuilding application on a platform-abstraction layer
- Use of standardized and widely available APIs (e.g., OpenGL, X) and cross-platform network APIs, protocols, and data representations (e.g., XML, JSON, CORBA, ASN.1, Unicode); pay attention to byte-ordering, structure-packing, and native character set issues
- Use of cross-platform libraries and Open Source libraries that have multiplatform support
- Use of a cross-platform virtual machine or interpreter (e.g., Java™, Smalltalk, Python, Perl).

Related Metrics

- Programming languages
  - Software is not portable if it is written using platform-specific language or language that is not supported on the targeted platform
  - Java™, C, C++, Python™ = 2, other high-level language = 1
- Architecture assessment
  - Interview the system architect or lead programmer and check off the architecture features for score
- Modularity
  - Number of modules
  - Module size score
- Complexity
  - Cyclomatic complexity
  - Coupling

A careful examination of all hardware dependencies is an important first step, as hardware dependencies present the biggest challenge in portability. Whether it is a smart card, a display device, a storage device, or some specialized hardware, when the targeted platform does not have hardware support, the project is not portable and the grade for the portability category is Poor for all categories.

Portability Pass/Fail questions:

- Is there any critical hardware dependency where support does not exist on the targeted platform?
- Is there platform-specific language in the software?

Table 2. Portability score matrix.

Grade		Programming Languages	Open Architechture Score	Modules			Complexity
				Number of Modules	Coupling (%)	Size	Cyclomatic Complexity
1	Very good	Java C, C++, Python, Pearl	= 1	> 20	< 50	0–200	< 3
2	Good		1–2	15–20	51–60	201–300	3–6
3	Fair	Other high-level language	2–3	11–15	61–75	301–400	7–10
4	Needs improvement		3–4	5–10	76–90	401–500	11–14
5	Poor	Platform-specific language	4–5	1–5	> 90	> 500	15–20

After passing the portability questions, the developer can determine the architecture score for portability by reviewing the software architecture or interviewing the system designer. The following questions should be answered:

- Does the project use a cross-platform virtual machine or primarily use interpreted language (e.g., Java™, Smalltalk, Python, and Perl)?

  [Yes = Very good portability, Grade = 1] (100% of final grade)

- If a cross-platform virtual machine or interpreter is not used, how well is the platform-specific code separated from the cross-platform code?

  [Estimate using the very good, good, fair, needs improvement, and poor grades.] (33% of final grade)

- Use standardized and widely available APIs (e.g., OpenGL, X) and cross-platform network APIs, protocols, and data representations (e.g., XML, JSON, CORBA, ASN.1, Unicode). Pay attention to byte-ordering, structure-packing, and native character set issues.

  [Estimate using the very good, good, fair, needs improvement, and poor grades.] (33% of final grade)

- Use cross-platform libraries and Open Source libraries that have multiplatform support.

  [Yes = Very good portability, Grade = 1

  No = Poor portability, Grade = 5] (33% of final grade)

Programming languages score:

Score assignment for Java™, C, C++, python or Perl is “Very Good”. Use “Fair” for other high-level languages.

Calculations:

Modularity Calculation:

$$Mo = (Mn \times .5) + (Ms \times .5)$$

Complexity Calculation:

$$Co = (V \times .5) + (Cp \times .5)$$

Portability Calculation:

$$P = Oa(.3) + Mo(.2) + Pl(.2) + Co(.1)$$

P = Portability
Oa = Open Architecture Assessment
Mo = Modularity
Mn = Number of Modules
Ms = Module Size
Cp = Coupling
Pl = Programming Languages
Co = Complexity
V = Cyclomatic Complexity

Maintainability

As technology, security risks, and hardware requirements increase, software must evolve to continue to function optimally. The need to maintain the software becomes a critical expenditure to ensure regular updates and revisions that correct any issues, improve efficiency and maintain security.

Additionally, the maintainability score is defined by modularity. Software that is modular can easily be decomposed into smaller, more maintainable parts.

Software maintainability is inversely proportional to both the effort required to make a change and the risk of breaking other functionality. The key targets in improving software maintainability are:

- Improve source code readability with comments and self-documented names
- Use a common programming language
- Keep software complexity low
- Use loose coupling and high cohesion
- Isolate software functions using modularization techniques

Related metrics:

- Comment Percentage in Code
- Modularity
- Number of Modules Score
- Module Size Score
- Cyclomatic Complexity

- Duplicate/Dead Code
- Number of Instances

Table 3. Maintainability score matrix.

Grade		Modules	*Size	Complexity (vG)	Duplicate/ Unused Code	Languages
1	Very good	> 20	0–200	> 3	< 10	< 5
2	Good	15–20	201–300	3–6	11–25	5–7
3	Fair	11–15	301–400	7–10	26–40	7–9
4	Needs improvement	5–10	401–500	11–14	41–50	10-12
5	Poor	1–5	> 500	15–20	> 50	> 12

The Maintainability score indicates how easy or hard it will be to upkeep the software. This score is determined mostly on software complexity, which is measured by identifying cyclomatic complexity. Software with higher complexity requires additional effort to upkeep or modify. This complexity is based on two attributes: (1) more effort is required to understand complex software and requires additional documentation as well as additional expertise, and 2) additional effort is required to test because more paths that are independent require testing. Complex software is typically more prone to inherent defects, and repairing these defects can increase sustainment costs.

Calculation:

Modularity Calculation:

$Mo = (Mn \times .5) + (Ms \times .5)$

Complexity Calculation:

$Co = (V \times .5) + (Cp \times .5)$

Maintainability Calculation:

$M = Co(.5) + Mo(.3) + Dp(.1) + Pl(.1)$

M = Maintainability
Co = Complexity
Cp = Coupling
V = Cyclomatic Complexity
Pl = Programming Languages
Dp = Duplicate/Dead Code

Security

Software Security is the measure of open vulnerabilities within the application code. The Application Security and Development (ASD) Security Technical Implementation Guide (STIG) provides the baseline requirements for government off-the-shelf (GOTS) applications and may be used to evaluate custom-developed applications and commercial off-the-shelf (COTS) software. Software developers can also use a static analysis tool output such as Common Weakness Enumeration (CWE)/SANS Top 25 vulnerabilities to measure software security.

The Security Metric formula is based on multi-tier weighting.

We implemented the multi-tier weighting to account for the disparate attributes associated with the security formula as well as the differing severity vulnerability rating scales provided by automated tool output. This formula assigns the highest value to Category I (CAT I) findings, followed by CAT II and CATIII, and other potential issues within the system that may be elevated to CAT level in the future.

The project team defined the CAT formula to properly weight the associated severity of each classification of defects and assist in prioritizing vulnerabilities addressed. This formula does not however represent the application's overall risk. Risk assessment methodology in accordance with DoDI 8510.01, Risk Management Framework (RMF) for DoD Information Technology (IT) and NIST SP 800-30, Guide for Conducting Risk Assessments, and Navy Guidance, should be used for risk management decisions.

Systems developed for Department of Navy use are typically required to possess zero CATI findings to field. CATII findings can be present, but only with proper mitigation and a plan of action to mitigate or remediate those items during a defined time. CATIII findings are low risk and are allowed; however, every effort should be made to remedy these accordingly. Based on these criteria, each CAT finding classification is weighted as listed in the formula with CATI items weighted as (0.5) the total value, CATII weighted at (0.3) the total value, and CATIII weighted at (0.2) the total CAT value. Once the sum of these values is calculated, the CAT attribute is weighted for the Overall Security value.

Since Defect Density represents risk for potential issues, it makes up a significant attribute to define the overall security posture of a software application, it is imperative that it be given individual weighting and an attribute score in the Overall Security value. For the purpose of the formula, Defect Density was weighted at (0.25) of the Overall Security value. This weight is due the increasingly large numbers of software defects that are found throughout the software we tested. This weight showed that the Defect Density could be considered very high. However, closer analysis revealed that the vast majority, approximately 80 % of those defects, are trivial or minor in scope, and focused on coding style issues. These defects are believed to not impact the ability of the software to be secure and withstand cyber-related attack. Based on the premise that a large number of defects can be prevalent, it is not suggested that large numbers indicate proportionately large numbers of critical defects, but suggests the associated weighting of this attribute at the appropriate 0.25 score.

Related Metrics:

- Open Web Application Security Project (OWASP) Top 10 and CWE

Table 4. Security score matrix.

Security Score Matrix Observed			Source Code	Number of Defects per KLOC	Priority	ASG STIGMap/CVSSv2 Qualitative Severity Rating Scale	
Grade		Severity	Scans				
			Severity %				
1	Very Good	None	> 1	< 1		None	0
2	Good	Low/CATT III	1–2.5	2.5–1	4	Low/ CAT III	0.1–3.9
3	Fair	Medium/Low	2.5–4	4–2.5	3	Medium/ CAT III	4.0–6.9
4	Needs Improvement	Medium/CAT II	4–10	10–4	2	High/ CAT I	7.0–8.9
5	Poor	Critical/High/CATI	> 10	> 10	1	Critical	9.0–10

Calculation:

Security Calculation:

$$S = (((CATI\#s(.5) + CATII\#s(.3) + CATIII\#s(.2)) * .75) + ((D / LOC) * .25)$$

S = Security
LoC = Lines of Code
D =# Defects
CAT I = Any vulnerability, the exploitation of which will directly or immediately result in the loss of Confidentiality, Availability, or Integrity
CAT II = Any vulnerability, the exploitation of which has potential to result in loss of Confidentiality, Availability, or Integrity.
CAT III = Any vulnerability, the existence of which degrades measures to protect against loss of Confidentiality, Availability, or Integrity.

EXTENSIBILITY

Extensibility can be confused with re-usability. Software extensibility describes how much effort is required to extend and change the software to provide new functionality that may not have been originally planned. Extensible design avoids software development issues such as low cohesion and high coupling.

Extensibility measures how easy or hard it will be to add to software's capability. Extensibility is impacted by various factors equally. These factors include software modularity, coupling/cohesion, complexity, and open architecture. Software that is modular

can be easily extended because less code requires modification. Therefore, based on these values we equally weight the attributes for the Extensibility formula.

Related Metrics:

- Modularity
- Number of Modules Score
- Module Size Score
- Weighted Method per Class (WMC)
- Coupling/Cohesion
- Complexity
- Cyclomatic Complexity
- Architecture

Table 5. Extensibility score matrix.

Grade		Modules	*Size	WMC	Cohesion on (LOCM %)	Coupling (CB0)	Architecture
1	Very good	> 20	0–200	1–2	> 90	> 2	< 5
2	Good	15–20	201–300	3–6	60–89	2–4	5–7
3	Fair	11–15	301–400	> 14	40–59	4–6	7–9
4	Needs improvement	5–10	401–500	14–20	25–39	6–8	10–12
5	Poor	1–5	> 500	> 20	> 25	> 8	> 12

Calculations:

Modularity Calculation:

$Mo = (Mn \times .5) + (Ms \times .5)$

Coupling/Cohesion Calculation:

$Cc = (Cp \times .5) + (Ch \times .5)$

Complexity Calculation:

$Co = (V \times .5) + (Cp \times .5)$

Extensibility Calculation:

$E = Mo(.25) + Cc(.25) + Co(.25) + Oa(.25)E = Extensibility$

Mo = Modularity
Mn = Number of Modules
Ms = Module Size
Cc = Coupling/Cohesion
Cp = Coupling

Ch = Cohesion
Co = Complexity
V = Cyclomatic Complexity
Oa = Open Architecture Assessment

Reliability

Software reliability is the measure of how well the software will work when a particular functionality is required. Issue density and software complexity are two key drivers impacting this metric. Software with fewer issues is less likely to break down when it executed. Software with lower complexity is typically easier to fix and test, requires less downtown time to fix any issues found, which improves availability and, in turn, increases reliability.

Our formula weighted the Software Issue Density and Cyclomatic Complexity values identical. Both contribute equally to the ability of a software application to maintain reliable operational use. While Cyclomatic Complexity produces the majority of the observed Software Issue Density due to risks associated with complex software, it also adds significant time in the repair of the associated defects encountered.

Related Metrics:

- Software Issue Density
- Cyclomatic Complexity

Calculations:

Reliability Calculation:

$R = Dd(.5) + V(.5)$

R = Reliability
Dd = Defect Density
V = Cyclomatic Complexity

Table 6. Reliability score matrix.

Grade		Bug Density in Modules (%)	Cyclomatic Complexity
1	Very good	> 5	> 3
2	Good	5–10	3–6
3	Fair	10–15	7–10
4	Needs Improvement	15–20	11–14
5	Poor	> 20	≥ 5

Testability

We assume that the software artifact contains faults; the Testability metric estimates the probability that testing will uncover the faults. If the testability of the software artifact is high, then finding existing faults through testing is easier. Figure 1 shows a simple, hypothetical model of Testability. If "A" is the range of all the possible inputs and "a" is the subset of inputs that causes the software to fail, then Testability in percentage equals to a/A * 100.

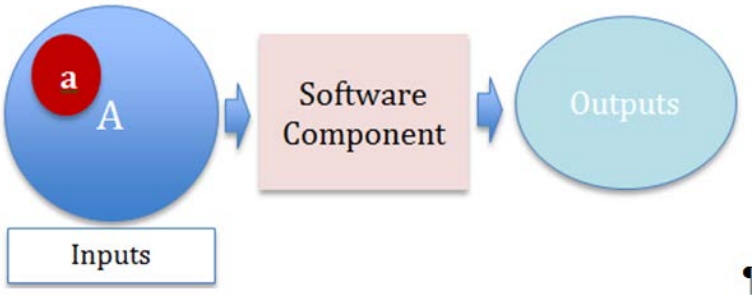

Figure 1. Testability % = (a/A)*100.

Finding "a" from "A" is impractical, as "A" is usually an infinite set. Software tests usually focus on the boundary conditions and the code coverage.

Testability is high when a high degree of controllability enables the injection of any input combination and the invoking of any possible state or combination of state. To uncover faults, the ability to observe the state and behavior is another desirable characteristic. Complexity, modularity, and size are also important factors in testability estimation.

Modular software was far easier to test because smaller, less complex modules require less test paths through the source code to execute complete test coverage. We weighted the formula based on this fact. For this formula, we weighted Modularity as half (0.5) the established value of Testability.

A higher score in Testability also relies on a lower Cyclomatic Complexity value (0.4). A lower Cyclomatic Complexity score depicts less test paths necessary to exercise fully all branches through the software, which supports full test automation. This score is associated with the Modularity weighting in that smaller modules, by necessity, would lend it into having greater numbers, of less complex modules, and would then reduce the complexity value. An additional note is Dead or Unused Code. This value accounts for a small variable (0.1) in the formula since higher complex modules exhibit some amount of Dead or Unused Code in the tested modules or files.

Related Metrics:

- Modularity
- Coupling/Cohesion
- WMC
- Number of modules
- Module size score
- Cyclomatic Complexity
- Reachable Code/Dead Code
- Number of Dead Code instances

Calculation:Modularity Calculation:

$Mo = (Mn \times .5) + (Ms \times .5)$

Testability Calculation:

$T = Mo(.5) + V(.4) + Dp(.1)$

T = Testability
Mo = Modularity
Mn = Number of Modules
Ms = Module Size
V = Cyclomatic Complexity
Dp = Duplicate/Dead Code

Scalability

The term “scalability” can encompass a wide range of meanings. For the purposes of this software quality model, scalability refers to how well the software performs given more users and data on a system representative of the production environment. The intent is to measure how the system adapts as the workload increases.

Without instrumenting the code or running performance tests, a quick measure of the scalability of the system depends on how well the software takes advantage of thread level and data level parallelism in addition to use of modular design. Software that exhibits more parallelism may have fewer dependencies and less coupling. Open architectures such as service oriented architectures (SOA) divide the system into composable parts that can adapt to varying demands.

Scalability should only be measured dynamically by monitoring resource utilization growth with increasing load. Depending on the intended platform, analysis tools such as Intel® VTune™ Amplifier XE, HP® Loadrunner, and Apache JMeter™, can dynamically assess the software scalability.

Quality to Metrics Dependency Matrix

Table 7 provides the quality characteristics to the dependency matrix..

Table 7. Quality characteristics to metrics dependecy matrix.

Software Quality Characteristics	Modularity	Cyclomatic Complexity	Duplicate Code	Dead Code	Lines of Code/ Comments	Coupling/ Cohesion	Abstractness	Defect Density	OWASP Top Ten	Weight Method per Class (WMC	Number of Classes	Size of Class	Number of Children per Class
Reliability	x	x				x	x			x	x	x	x
Portability	x					x					x	x	

Table 7. Quality characteristics to metrics dependecy matrix. (continued)

Software Quality Characteristics	Modularity	Cyclomatic Complexity	Duplicate Code	Dead Code	Lines of Code/ Comments	Coupling/ Cohesion	Abstractness	Defect Density	OWASP Top Ten	Weight Method per Class (WMC	Number of Classes	Size of Class	Number of Children per Class
Maintainability	x	x	x	x	x						x	x	
Security									x				
Extensibility	x	x				x	x			x	x	x	
Reliability		x						x					
Testability	x	x		x		x					x	x	
Scalability													

SOFTWARE METRICS DEFINITION

MODULARITY

Modularity separates a software system in independent and collaborative modules for organization in software architecture [1]. Modular software has several advantages such as maintainability, manageability, and comprehensibility.

Five attributes are closely related to modularity in software systems: size, coupling/ dependency, complexity, cohesion, and information hiding. The first attribute is the size of the module as well as the system that contains each module. It should not be too large. Additional system features should be translated as the addition in the module of the system. The second attribute is coupling/dependency, which consists of a direct/syntactic achieved through composition, method signatures, class instantiations, inheritance, and semantic or indirect coupling. Developers can measure the third attribute, complexity, by using software metrics such as McCabe's Cyclomatic Complexity or Halstead's Software Metrics. The fourth attribute is cohesion, which measures the integrity of the code inside each of module. The terms used to measure cohesion qualitatively are high cohesion or low cohesion. The last attribute is information hiding, which involves hiding the details of implementation from external modules. An ideal modular software system should have the following attributes [2]:

- **Small Size:** Each module (package) and many modules in the system should be small. Each module/package should only be responsible for a simple feature, and the more complex features should be composed of many of these simple features. The possible software metrics to measure size are Non-Comment Lines of Code (NCLOC), Lines, or Statements.

- **Low Coupling/Dependency:** Minimization or standardization of coupling/ dependency occurs through standard format, that is, published application programming interfaces (APIs), elimination of semantic dependencies, etc. The possible software metrics to measure coupling are Afferent Coupling, Efferent Coupling, or RFC (Response for a Class).
- **Low complexity:** A hierarchy of modules prefers flatter rather than taller dependency. The most popular software metrics to measure complexity is Cyclomatic Complexity. [3]
- **High Cohesion:** High integrity of the internal structure of software modules are usually stated as either high cohesion or low cohesion. The better measure of cohesion in object-oriented programming such as Java™ is LCOM4 or Lack of Cohesion Metrics version 4, proposed by Hitz and Montazeri.
- **Open for Extension and Close to Modification:** Capability of the existing module is extended to create a more complex module to avoid changing already debugged code. The creation of new modules should be encouraged using available extension and not modifying the already tested module. [2]

DEPENDENCIES

Almost all software systems have components that are identifiable as data items, data types, subprograms, or source files. A dependency exists between two components if a change to one may have an impact that will require changes to the other.

CYCLOMATIC COMPLEXITY

The cyclomatic complexity of a section of source code is the number of linearly independent paths within it. For instance, if the source code contains no control flow statements (conditionals or decision points), such as IF statements, the complexity is 1, since there is only a single path through the code. If the code has one single-condition IF statement, there are two paths through the code: one where the IF statement evaluates to TRUE and another one where it evaluates to FALSE, so complexity is 2 for single IF statement with single condition. Two nested single-condition IFs, or one IF with two conditions, produces a complexity of 4, 2 for each branch within the outer conditional. Thomas J. McCabe, Sr., developed cyclomatic complexity in 1976. [3]

One of McCabe's original applications was to limit the complexity of routines during program development; he recommended that developers should count the complexity of the modules they are developing, and split them into smaller modules whenever the cyclomatic complexity of the module exceeds 10. This practice was adopted by the National Institute of Standards and Technologies (NIST) Structured Testing methodology, with an observation that since McCabe's original publication, the figure of 10 has received substantial corroborating evidence, but that in some circumstances it may be appropriate to relax the restriction and permit modules with a complexity as high as 15. As the methodology acknowledged occasional reasons for going beyond the agreed-upon limit, it phrased its recommendation as follows: "For each module, either limit cyclomatic complexity to [the agreed-upon limit] or provide a written explanation of why the limit was exceeded." [4]

ABSTRACTNESS

Robert Martin proposed a widely used metric suite in 1994. Abstractness was included and is the ratio of the number of abstract classes versus the total number of classes. A value of 0 would mean no abstract classes were present and a maximum value of 1 would mean that all of the classes are abstract. [5]

Abstractness is important to maintain stability within the source code [6]. Abstractions allow the implementation to change without modifying the interfaces, so that dependent code does not break. Abstractions also may indicate the use of design patterns.

COUPLING

This metric shows how the source code depends on the strength with which classes, methods, and methods' parameters are connected to each other, and the degree to which each program module relies on each one of the others.

Low (loose) coupling means that source code is organized so that its methods and classes slightly address each other. Software developers do not write the source code optimally, but rather create independent methods and classes to solve separate tasks.

COHESION

This metric shows an average number of internal relationships per type in a package/namespace.

AFFERENT COUPLING

Afferent means incoming. Software developers apply this metric to packages and namespaces. It is the number of types outside a package or namespace that depend on types of the current package or namespace. High afferent coupling shows that the analyzed package/namespace is very important.

EFFERENT COUPLING

Efferent means outgoing. This metric is the number of types inside a package/namespace that depend on types of other types/packages. High efferent coupling shows the degree to which the measured package/namespace depends on external packages/namespaces.

The main idea of this metric is that the class has high cohesion when all its methods use all the fields of this class.

DUPLICATE CODE

Code that is similar or copy and pasted can be harmful because it can increase maintenance costs and inconsistent changes to duplicate code can lead to inconsistent behavior. The presence of similar code also indicates the presence of a missed opportunity for reuse. [7]

DEAD CODE

Dead code is code that is never used. This code includes unused methods and variables. Dead code can lead to difficulties in understanding the program, which can lead to bugs or an increase in maintenance costs. [8]

DEFECT DENSITY OR SOFTWARE ISSUE DENSITY

Defect Density is the number of confirmed defects detected in a software/component during a defined period of development/operation divided by the size of the software/component. [9]

Elaboration:

The “defects” are:

- Confirmed and agreed upon (not just reported)
- Dropped defects are not counted

The period might be for one of the following:

- Duration (the first month, the quarter, or the year).
- For each phase of the software life cycle
- For the whole of the software life cycle

The size is measured in one of the following:

- Function Points (FP)
- Source Lines of Code

WEIGHTED METHODS PER CLASS (WMC)

This metric provides a better measurement of class complexity. It is the sum of the complexities of all the class methods. A class having a high WMC is more complex and is harder to maintain, reuse, or extend. Complexity is not explicitly defined for the metric to be generic. In the special case when complexity is not considered, the WMC metric is the same as the number of methods in the class.

NUMBER OF CHILDREN PER CLASS (NOC)

In object oriented (OO) terminology, classes that inherit their functionality from other classes are called Children Classes. A high value for NOC indicates that the class is implemented in abstract manner since other classes can inherit from it and reuse it.

STATIC-CODE ANALYSIS TOOLS

Both industry and open-source developers have provided a wide array of useful static-code analysis tools.

ATOMIQ [10]

Summary: Atomiq is a free tool that finds duplicate and similar code.

Languages: C/C++, C#, VB.net®, ASPX, RUBY, Python™, Java™, ActionScript®, XAML

Metrics Supported: Duplicate Code

CHECKSTYLE [11]

Summary: Checkstyle is an open-source tool to help developers write Java™ code that adheres to a coding standard. There is a plug-in for Eclipse™, IntelliJ™ IDEA, Netbeans™, Jenkins, and others that notify developers on-the-fly of any violations.

Languages: Java™

Metrics Supported: Cyclomatic Complexity, Design For Extension, Presence of Javadoc Comments (packages, types, methods, variables), Magic Numbers, File Length, Method Length, Method Count

COUNT LINES OF CODE (CLOC) [12]

Summary: CLOC counts blank lines, comment lines, and physical lines of source code in many programming languages. Given two versions of a code base, CLOC can compute differences in blank, comment, and source lines. It is written entirely in Perl with no dependencies outside the standard distribution of Perl v5.6 and higher (code from some external modules is embedded within CLOC) and so is quite portable.

Languages:

Metrics Supported: Lines of Code, Lines of Comments, Lines of Blank Lines

CPPDEPEND [13]

Summary: CppDepend simplifies managing a complex C/C++ code base. You can analyze code structure, specify design rules, do effective code reviews, and master evolution by comparing different versions of the code. CppDepend counts the number of lines of code. It also comes with more than 80 other code metrics. Some of them are related to your code organization (the number of classes or namespaces, the number of methods declared in a class, etc.), some of them are related to code quality (complexity, percentage of comments, number of parameters, cohesion of classes, stability of projects, etc.), some of them are related to the structure of code (which types are the most used, depth of inheritance, etc.)

Languages: C++

Metrics Supported: Similar to NDepend

FINDBUGS™ [14]

Summary: Open-source tool written by the University of Maryland to find bugs in Java™ programs. A graphical user interface (GUI) is provided in addition to access by antenna.

Languages: Java™

Metrics Supported: Identifies code that follow common bug patterns for Java™, such as possible null pointer dereference or index out of bounds.

FIND SECURITY BUGS [15]

Summary: Open-source plugin for FindBugs™, providing security audits for Java™ Web applications.

Languages: Java™

Metrics Supported: It can detect 63 different vulnerability types with over 200 unique signatures with extensive references given for each bug patterns with references to OWASP Top 10 and CWE.

FORTIFY™ [16]

Summary: Fortify™ by Hewlett Packard® provides a comprehensive tool for detecting security vulnerabilities.

Languages: 21 languages

Metrics Supported: 500 types of vulnerability detection, including OWASP Top 10

GMETRICS [17]

Summary: The GMetrics project provides calculation and reporting of size and complexity metrics for Groovy source code. GMetrics scans Groovy source code, applying a set of metrics, and generating an HTML or XML report of the results.

Languages: Groovy

Metrics Supported: Cyclomatic Complexity, Afferent Coupling, Efferent Coupling, Lines per Method, Lines per Class, Number of Classes per Package, Number of Fields per Class

JARCHITECT [18]

Summary: JArchitect offers a wide range of features. It is often described as a Swiss Army Knife for Java™ developers. JArchitect comes with more than 80 other code metrics. Some of them are related to your code organization (the number of classes or Packages, the number of methods declared in a class, etc.), some of them are related to code quality (complexity, percentage of comments, number of parameters, cohesion of classes, stability of projects, etc.), and some of them are related to the structure of code (which types are the most used, depth of inheritance, etc.).

Languages: Java™

Metrics Supported: Similar to NDepend

MCCABE IQ [19]

Summary: McCabe IQ provides software analysis tools to measure the complexity and quality of code at the application and enterprise level.

Languages: Ada, ASM86, C/C++, C#, C++.net, COBOL, FORTRAN, Java™, JSP, Perl, PL1, VB, VB.net®

Metrics Supported: Cyclomatic Complexity (< 10), Module Design Complexity (< 7), Essential Complexity (< 4), Lack of Cohesion Methods (> 75), Object Integration Complexity, Maintenance Severity

NDEPEND [20]

Summary: NDepend offers a wide range of features to let the user analyze a code base. It is often described as a Swiss Army Knife for .netT developers.

Languages: .NET

Metrics Supported: Lines of Code, Lines of Comments, Afferent Coupling, Efferent Coupling, Abstractness, Instability, Lack of Cohesion of Methods, Cyclomatic Complexity (< 10)

PMD® [21]

Summary: PMD® is an open-source tool used to find defects, including possible bugs, dead code, suboptimal code, overcomplicated expressions, and duplicate code.

Languages: PMD® supports rulesets for Java™, Javascript™, JSP, PL/SQL™, Velocity Template Language, and XML/XSL. The PMD® Copy and Paste Detector can run with additional languages, including C++, C#, FORTR, Go, MATLAB®, etc.

Metrics Supported: Varies by language. For most languages, copy paste detection is provided. For Java™, additional metrics include: Source lines of code, Cyclomatic

Complexity (<10), Coupling Between Objects, Loose Coupling, Exception Handling, Unused Code (Dead Code)

SONARQUBE™ [22]

Summary: SonarQube™ is an open-source platform for managing code quality. The tool supports 20+ languages through plug-ins and can collect a variety of metrics in addition to allowing the creation of custom metric rules. It also supports a variety of plug-ins for other code analysis tools such as Checkstyle and PMD® that can extend the number of metrics it can collect.

Languages: Java™, C#, C/C++, PL/SQL™, Cobol, Advanced Business Application Planning (ABAP®) (20+ languages supported through plug-ins)

Metrics Supported: Duplicate Code, Failed Unit Tests, Insufficient Branch Coverage by Unit Tests, Insufficient Comment Density, Insufficient Line Coverage by Unit Tests, and Skipped Unit Tests

UNIFIED CODE COUNT (UCC) [23]

Summary: UCC is a comprehensive source lines of code counter produced by the USC Center for Systems and Software Engineering. It is an open-source tool that can be compiled with any ANSI standard C++ compiler.

Languages: C/C++, C#, Java™, VB, Assembly, and others

Metrics Supported: Source Lines of Code, Physical Source Lines of Code (PSLOC), Logical Source Lines of Code (LSLOC)

UNDERSTAND™ [24]

Summary: Understand™ is a robust static code analysis tool developed by Scientific Toolworks, Inc. supporting the generation of multiple kinds of reports and views of the data at different levels (project, class, object oriented metrics, program unit, file). Understand can perform dependency analysis in addition to code standards testing.

Languages: Ada, COBOL, Coldfire® 68K Assembly, C/C++, C#, FORTRAN, Java™, Jovial, Pascal, PL/M, Python™, VHDL, Javascript™, PHP, XML, HTML, CSS

Metrics Supported: Understand can check for adherence to published coding standards from Effective C++ (3rd Edition) by Scott Meyers, MISRA-C 2004, MISRA-C++ 2008, and any custom coding standards defined by the user. Understand also supports checks for Dead Code, Cyclomatic Complexity, Source Lines of Code, Coupling Between Objects, Lack of Cohesion in Methods, and Comment to Code Ratio.

TOOLS TO METRIX MATRIX

The tools applied to the dependency matrix are provided in Table 8.

Table 8. Tools to metrics matrix.

Tools	Cyclomatic Complexity	Duplicate Code	Dead Code	Lines of Code/ Comments	Coupling/ Cohesion	Abstractness	Defect Density	OWASP Top Ten	Number of Classes	Size of Class	Number of Children per Class
Atomiq		x		x							
Checkstyle	x										
CLOC				X							
CppDepend	x			x	x	x				x	x
FindBugs™							x				
Find Security Bugs							x	x			
Fortify™							x	x			
GMetrics	x			x	x				x	X	
JArchitect	x			x	x	x				x	X
McCabe IQ	x										
NDepend	x			x	x	x				x	X
PMD®		x	x		x		X				
SonarCube™ (no plug-ins)		x		X							
UCC			x								
Understand™	x	x	x	x	x		x				

CONCLUSION

We have identified software qualities, software analysis tools, and related metrics. This effort was based on existing data and analysis of that data, proofing a formula for use by Department of the Navy software development efforts to measure inherent quality of the software under development. However, each project is unique and requires a software quality model tailored for its individual needs.

This process is an ongoing effort for any organization and requires analysis of data and trends to determine the most effective implementation of metrics to achieve the highest fidelity of quality and provide for beneficial cost savings. In addition, evaluators need to create and calibrate cost functions for the cost of fixing the code that does not meet software code requirements. This activity will normalize the software model based on cost.

REFERNCES

1. E. Y. Nakagawa. 2008. “Software Architecture Relevance in Open Source Software Evolution: A Case Study.” *32nd Annual IEEE International Computer Software and Application Conference (COMSAC)* (pp. 1234–1239). July 2–August 1, Tunku, Finland.
2. W. R. Emanuel, R. Wardoyo, J. E. Istiyanto, and K. Mustofa. 2011. “Statistical Analysis on Software Metrics Affecting Modularity in Open Source Sofware,” *International Journal of Computer Science & Information Technology (IJCSIT)*, vol. 3, pp. 105–118.
3. T. J. McCabe, 1976. "A Complexity Measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 143–150.
4. A, H. Watson and T. J. McCabe. 1996. “Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric.” NIST Special Publication 500–235. McCabe Software.
5. Y. Suresh, J. Pati, and S. K. Rath. 2012. “Effectiveness of Software Metrics for Object-oriented System,” *Procedia Technology*, vol. 6, pp. 420–427.
6. R. Martin. 1994. “Object Mentor.” Available at http://www.objectmentor.com/resources/articles/oodmetrc.pdf. [Accessed Aug. 15, 2015].
7. E. Juergens, F. Deissenboeck, and B. Hummel, 2010. “Code Similarities Beyond Copy & Paste,” *Proceedings of the 14th European Conference on Software Maintenance and Reengineering* (pp. 78–87). March 15–18, Madrid, Spain.
8. Sonar Qube. “Detect Dead Code and Calls to Deprecated Methods with Sonar Squid,” [Online]. Available: http://www.sonarqube.org/detect-dead-code-and-calls-to-deprecated-methods-with-sonar-squid/ [Accessed August 14, 2015].
9. Software Testing Fundamentals. “Defect Density.” [Online]. Available: http://softwaretestingfundamentals.com/defect-density.
10. "Atomiq Code Similarity Finder," [Online]. Available: http://www.getatomiq.com/ . [Accessed July 31, 2015].
11. "Checkstyle Checks," [Online]. Available: http://checkstyle.sourceforge.net/checks.html. [Accessed July 31, 2015].
12. Github, Inc. “CLOC: Count Lines of Code.” Available: https://github.com/AlDanial/cloc. [Accessed October 9, 2015].
13. Codergears, "CppDepend," Codergears, [Online]. Available: http://cppdepend.com. [Accessed August 14, 2015].
14. "FindBugs Bug Descriptions," [Online]. Available: http://findbugs.sourceforge.net/bugDescriptions.html. [Accessed 31 July 2015].
15. P. Arteau, Open-source, open for contributions, [Online]. Available: http://h3xstream.github.io/find-sec-bugs.

16. Hewlett Packard. “Securing Your Enterprise Software.” Available: http://www8.hp.com/us/en/software-solutions/asset/software-asset-viewer.html?asset=1356157&module=1823975&docname=4aa4-2455enw&page=1823980. [Accessed August 29, 2015].

17. Gmetrics. Apache License V2.0, [Online]. Available: http://gmetrics.sourceforge.net/index.html

18. Codergears. “JArchitect.” [Online]. Available: http://www.jarchitect.com. [Accessed Oct. 9, 2015].

19. McCabe Software. “McCabe Software Metrics Glossary.” [Online]. Available: http://www.mccabe.com/iq_research_metrics.htm [Accessed July 31, 2015].

20. Codergears. “ndepend.” [Online]. Available: http://www.ndepend.com

21. SourceForge.net. “PMD Rulesets Index: Current Rulesets,” [Online]. Available: https://pmd.github.io/pmd-5.3.3/pmd-java/rules/index.html. [Accessed July 31, 2015].

22. SonarSource S.A. “SonarQube Manual Issues.” [Online]. Available: http://docs.sonarqube.org/display/SONAR/Manual+Issues [Accessed August 3, 2015].

23. USC Center for Systems and Software Engineering. “About UCC.” [Online]. Available: http://csse.usc.edu/ucc_wp/about/. [Accessed July 31, 2015].

24. Scientific Toolworks, Inc. “Understand Features.” [Online]. Available: https://scitools.com/features. [Accessed August 3, 2015]

INITIAL DISTRIBUTION

84300	Library	(2)
85300	Archive/Stock	(1)
53203	C. Johnson	(1)

Defense Technical Information Center
Fort Belvoir, VA 22060–6218 (1)

Approved for public release.

SSC Pacific
San Diego, CA 92152-5001