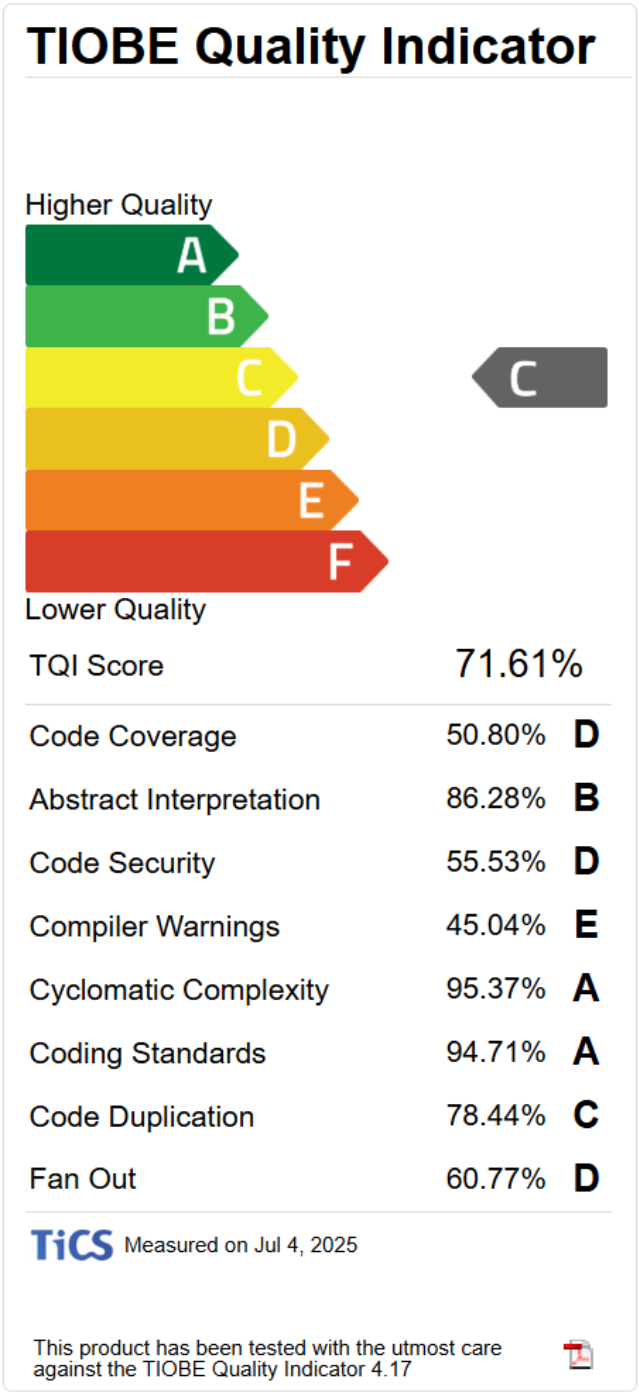
The TIOBE Quality Indicator

A pragmatic way of measuring code quality



Document ID: TIOBE-20120718.1
Version: 5.2 (Authorized)
Date: 17-October-2025
Author: Paul Jansen (paul.jansen@tiobe.com)

Contents

Contents 2
1 Introduction 2
2 Software Quality Characteristics 2
3 Software Metrics 3
4 Mapping Software Metrics to Quality Characteristics 4
5 Compliance Factor 8
6 Measuring and Judging Metric Values 9
7 TIOBE Quality Indicator 14
7.1 Scope 15
7.2 Recommended TQI Levels 15
8 Conclusions 16
Appendix A References 17
Appendix B Reviewers 18

1 Introduction

The proverb "the proof of the pudding is in the eating" applies perfectly to software quality. Because only after a software product has been shipped, the true quality of a software product reveals itself. Software quality is determined by

- the number of defects found after release
- the severity of these defects
- the effort needed to solve these defects

More than 30 years ago, software engineer Barry Boehm already observed that the costs of repairing defects increase exponentially if they are found later on in the software development process [1]. This means that if it is possible to have a way to measure the software quality of a system before release, it will potentially save a lot of money.

The goal of this document is to define such a software quality measurement system based on a pragmatic approach. The focus is on code quality (as opposed to e.g. quality of requirements or the architecture). The defined approach is based on more than 20 years of experience in this field and the analysis of more than 2 billion lines of industrial production software code that are checked each day.

2 Software Quality Characteristics

There is an ISO definition of software quality, called ISO 25010 [2]. This standard defines 8 main quality characteristics and a lot of sub-characteristics. The 8 main quality characteristics are:

- **Functional suitability**. The degree to which the product provides functions that meet stated and implied needs when the product is used under specified conditions.

- **Reliability**. The degree to which a system or component performs specified functions under specified conditions for a specified period of time.
- **Performance efficiency**. The performance relative to the amount of resources used under stated conditions.
- **Usability**. The degree to which the product has attributes that enable it to be understood, learned, used and attractive to the user, when used under specified conditions.
- **Security**. 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.
- **Compatibility**. The degree to which two or more systems or components can exchange information and/or perform their required functions while sharing the same hardware or software environment.
- **Maintainability**. The degree of effectiveness and efficiency with which the product can be modified.
- **Portability**. The degree to which a system or component can be effectively and efficiently transferred from one hardware, software or other operational or usage environment to another.

The ISO 25010 standard helps as a starting point to determine quality in an early stage. It has 2 main drawbacks, however:

- The standard does not specify how to measure quality characteristics. Some of the quality characteristics even seem unfit for objective measurement. Take "Usability" for instance, with sub-characteristics such as "User interface aesthetics" and "Learnability". How to measure this and what is the unit of measurement? Note that there is an ISO standard 25023 [15] that defines metrics to measure the ISO 25010, but most of these metrics are at behavioral level and not at software code level [16]. Recently, ISO 5055 [16] has been introduced, which looks promising, although it covers only 4 out of the 8 ISO 25010 quality characteristics.
- Most of the quality characteristics defined have different meanings in different contexts. Even if it is possible to measure a quality characteristic, it is impossible to define clear objective criteria for what is considered good or bad. "Performance efficiency" is a good example of such a quality characteristic. For some software systems a response within 1 second is sufficient, whereas others demand a response within 1 millisecond.

3 Software Metrics

We could try to define measurement systems for the ISO quality characteristics in a top-down, scientific way. However, this is too ambitious.

Can we measure anything at all? Yes, but we need to take a more pragmatic approach. Lots of metrics are applied to software code nowadays, but unfortunately there is insufficient proof (yet) whether these metrics contribute to better code. Examples of such metrics are cyclomatic complexity [3], code duplication [4] and all kinds of code coverage [5]. These metrics are approximations of some of the quality characteristics of the ISO 25010 standard.

To obtain a systematic way of measuring and qualifying these measurements, the 8 most commonly used software code quality metrics in industry today have been selected that can be measured in an automated way. These are:

1. Code coverage
2. Abstract interpretation [6]

3. Code security [9], [10]
4. Compiler warnings
5. Cyclomatic complexity
6. Coding standards [7]
7. Code duplication
8. Fan out [8]

4 Mapping Software Metrics to Quality Characteristics

We define the metrics of the previous section and map them on the quality characteristics of the ISO 25010 standard.

Code coverage. Before software engineers hand over their code to the next stage in the software development cycle, they usually perform unit tests. These are small automated tests that check a particular part of a program such as a single function. The actual results of these automated tests are compared to the expected results. Unit tests are a powerful way to check whether a program behaves like it is designed to behave at the lowest level. The code coverage metric indicates how many lines of code or executable branches in the code have been touched during the unit test runs. The lower the coverage, the lower the quality of the performed unit tests. Code coverage is an indicator of both "Functional Suitability" and “Reliability” of the ISO 25010 standard.

A simple example of the output of a code coverage tool is shown for the C# code below. Every line that is colored “green” is touched during at least one of the tests, whereas “red” lines are not touched by any test.

```
25: if (
26:   element.ElementType == ElementType.Class &&
27:   element.Declaration.Name.EndsWith("Class")
28: )
29: {
30:   addViolation(element, element.Declaration.Name, "Class");
31: }
32: else if (
33:   element.ElementType == ElementType.Struct &&
34:   element.Declaration.Name.EndsWith("Struct")
35: )
36: {
37:    addViolation(element, element.Declaration.Name, "Struct");
38: }
39: return true;
```

The output of the code coverage tool shows that all lines in this code sample are covered by (unit) tests, except for line 37.

Abstract Interpretation. A fairly new technology is to detect possible reliability issues in software programs by running abstract interpretation tools, also known as deep flow analysis tools. These tools are capable of automatically detecting all kinds of programming errors related to the control flow of a program. Examples are null pointer dereferences, divisions by zero and unclosed database connections. The advantage of these tools

is that they generate their results without actually running the programs. This is done by calculating all possible paths through a program in an efficient way. Errors found by abstract interpretation are severe programming errors that may result in crashes. This metric is mapped to the "Reliability" characteristic of the ISO 25010 standard.

A simple example of an abstract interpretation issue is shown in the Java code below.

```
159: public Order getOrder() {
160: // Only return orders with a valid date
161:   if (orderDate.isValid()) {
162:     return order;
163:   } else {
164:     return null;
165:   }
166: }
...
227: public List<Order> getOrderPackages() {
228:   return getOrder().getCorrespondingOrderPackages(company);
229: }
```

Abstract interpretation tools will flag a possible null pointer dereference at line 228, because the function “getOrder” can return null in case the order has no valid date. If this situation occurs an exception will be thrown, possibly resulting in program abortion.

Code Security. Code security of software is about how vulnerable code is to get unauthorized access to data and how easy it is to make changes to the software by exploiting security leaks. Examples of such leaks are buffer overflows (to let the program crash) and exposure of sensitive data (thus giving users information to get unauthorized access).

An example of a security leak is given in the following C code.

```
318: char buf[8];
319: sprintf(buf, “some_evil_program_code”);
```

At line 319 a very long string of characters is written to an array called “buf” that can only hold 8 characters. The characters that don't fit in “buf” are saved somewhere else, possibly overwriting code that is supposed to do the program execution. By making abuse of this hole, one can run another program than the one that is intended to run. The corrected example is

```
318: char buf[8];
319: snprintf(buf, 8, “some_evil_program_code”);
```

By using “snprintf” instead of “sprintf” the number of characters written to the buffer is restricted by the second argument.

Compiler warnings. In order to execute a software program on a computer it must first be compiled or interpreted. Compilers/interpreters generate errors and warnings. Errors must be fixed otherwise the program cannot run. Warnings on the other hand do not necessarily need to be solved. However, some compiler warnings indicate serious program flaws. Leaving these unresolved will probably impact the "Reliability" of the code. Apart from reliability issues, most compilers also warn about portability issues. So this metric can also mapped to “Portability” in most cases.

A simple example of a compiler warning is shown in the C code below.

```
31: int func(int i) {
```

```
32:   if (i = 0) {
33:     return -1;
34:   }
...
58: }
```

Most compilers will complain about the assignment in the if condition at line 32 (probably a comparison was meant instead).

Cyclomatic complexity. One of the oldest software metrics is cyclomatic complexity. Cyclomatic complexity counts the number of independent paths through a program. For instance, each "if" statement adds one extra code path. The higher the cyclomatic complexity the harder it is to understand a program. Moreover, the more paths there are, the more test cases need to be written to achieve a decent code coverage. The average cyclomatic complexity per function is an indicator that enables comparisons of complexity between programs. It is part of the "Maintainability" characteristic of the ISO 25010 standard.

The C# code below shows a simple example of how cyclomatic complexity is calculated.

```
123: public int getValue(int param1)
124: {
125:   int value = 0;
126:   if (param1 == 0)
127:   {
128:     value = 4;
129:   }
130:   else
131:   {
132:     value = 0;
133:   }
134:   return value;
135: }
```

The cyclomatic complexity of the function “getValue” at line 123 is 2 (one path through “then” and one through “else”).

Coding standards. Software maintenance is one of the most time consuming tasks of software engineers. One of the reasons for this is that it is hard to understand the intention of program code long after it has been written, especially if it has been updated a lot of times. A way to reduce the costs of software maintenance is to introduce a coding standard. A coding standard is a set of rules that engineers should follow. These coding rules are about known language pitfalls, code constructions to avoid, but also about naming conventions and program layout. Since coding standards usually contain many different rules they can be mapped to most quality characteristics. Most rules concern "Maintainability" and “Reliability”, but there are also rules available for “Portability” and “Performance Efficiency” of the ISO 25010 standard.

An example of a coding standard violation is shown below.

```
31: int abs(int i) {
32:   int result;
33:
34:   if (i < 0) {
35:     result = -i;
```

```
36:     goto end;
37:   }
38:   result = i;
39:   end:
40:   return result;
41: }
```

Any C coding standard will complain about the goto statement used at line 36. It is considered bad practice to use goto statements.

Code duplication. Sometimes, it is very tempting for a software engineer to copy a piece of code and make some small modifications to it instead of generalizing functionality. The drawback of code duplication is that if one part of the code must be changed for whatever reason (solving a bug or adding missing functionality), it is very likely that the other parts ought to be changed as well. But who is to notice? If nobody does, code duplication will lead to rework in the long term. This has a negative effect on the "Maintainability" characteristic of the ISO 25010 standard.

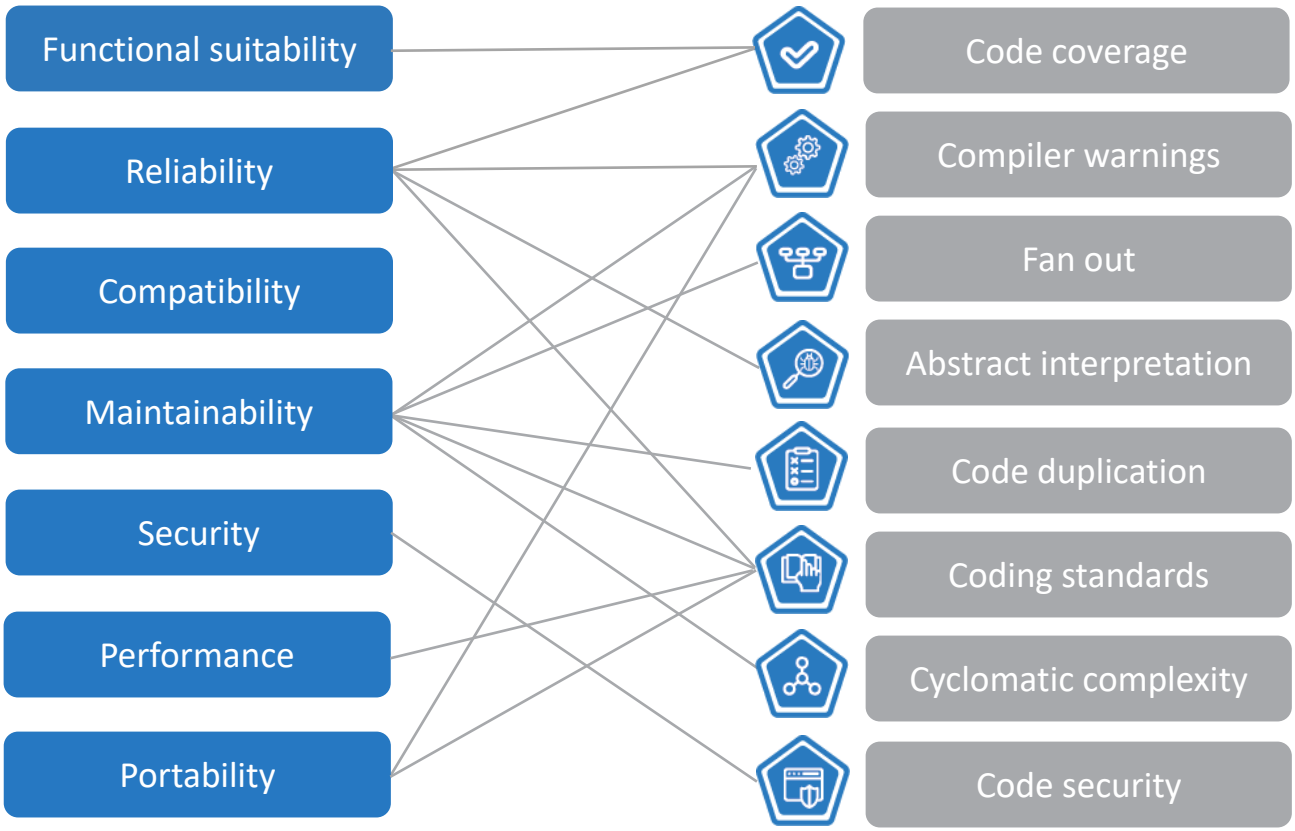
Fan out. Software programs are structured in terms of modules or components. These modules and components “use” each other. The fan out metric indicates how many different modules are used by a certain module. If modules need a lot of other modules to function correctly (high fan out), there is a high interdependency between modules, which makes code less modifiable. Hence, fan out is mapped to the "Maintainability" ISO 25010 characteristic.

An example of a high fan out is shown in the Java code below.

```
 1: package com.tiobe.plugins.eclipse.analyzer;
 2:
 3: import java.io.IOException;
 4: import java.util.Map;
 5:
 6: import org.apache.commons.exec.CommandLine;
 7: import org.apache.commons.exec.DefaultExecutor;
 8: import org.apache.commons.exec.ExecuteException;
 9: import org.apache.commons.exec.ExecuteResultHandler;
10: import org.apache.commons.exec.ExecuteWatchdog;
11: import org.apache.commons.exec.Executor;
12: import org.apache.commons.exec.PumpStreamHandler;
13: import org.apache.commons.exec.environment.EnvironmentUtils;
14: import org.apache.commons.io.output.NullOutputStream;
15: import org.eclipse.core.resources.IProject;
16: import org.eclipse.core.resources.IResource;
17:
18: import com.tiobe.plugins.eclipse.console.ITICSConsole;
19: import com.tiobe.plugins.eclipse.console.TICSConsole;
20: import com.tiobe.plugins.eclipse.util.EclipseUtils;
21:
22: public class TICSAnalyzer implements ITICSAnalyzer {
```

In this article we have adopted the simple definition of fan out to measure the number of import statements. Hence, the fan out of the Java file above is 16.

A summary of the mapping between TQI metrics and ISO 25010 characteristics is given below.



5 Compliance Factor

Some of the metrics discussed in the previous section can be easily mapped to some qualification. For instance, if a file has a code duplication of 0% then this is considered to be very good, whereas if it is 50% this is considered bad programming practice.

However, for four of the eight TQI metrics there is no such straightforward mapping. These are:

- Abstract interpretation
- Compiler warnings
- Coding standards
- Code Security

For instance, if there are 3,000 coding standard violations left in your code is that all right or plain wrong? Whether this is a good or bad thing depends on 3 additional factors:

1. How many coding rules are measured? If a coding standard has more rules than another coding standard, the chances are higher that there will be more violations. But this doesn't mean that that code has less code quality.

2. What is the severity level of the rules that have been violated? If only unimportant rules are violated the code quality is better than in case the same amount of blocking rules are violated.

3. What is the size of the software? If there are 3,000 violations in a system consisting of 10 million of lines of code then this is less severe if compared to a system that has the same amount of violations and only contains 1,000 lines of code.

In order to solve this issue the notion of "compliance factor" is introduced. The compliance factor expresses how much some piece of software code complies to a certain set of rules. This could be for instance a set of compiler warnings or a set of code security rules.

The formal definition of the compliance factor is as follows:

$$\textbf{compliance factor} = \frac{100}{\text{weighteds violations}/(\text{average rules per level} * (\text{loc}/1{,}000)) + 1}$$

where the definition of weighted violations is:

$$\textbf{weighted violations} = \sum_{i=min\ severity\ level}^{max\ severity\ level} \frac{\text{violations}(i)}{4^{i-1}}$$

and where the definition of average number of rules per severity level is

$$\textbf{average rules per level} = \left(\sum_{i=min\ severity\ level}^{max\ severity\ level} \text{rules(i)}\right)/\text{maximum severity level}$$

A detailed explanation of this formula is outside the scope of this document. It can be found in a separate document [11]. TIOBE has used this definition for more than 20 years in many projects and it appears to work well in practice.

6 Measuring and Judging Metric Values

This section defines how the 8 metrics of the previous section are measured. It also specifies how the obtained metric values are to be judged on a scale between 0 and 100 (called the score). The formulas that are used to calculate the scores for metrics have been determined empirically, based on analyzing the more than 2 billion lines of code that are checked by TIOBE Software each day.

There are 6 different categories distinguished based on the normative system. These are similar to the European Union energy labels [12]. See the table below.

Category	Name	TQI Score
A	Outstanding	>= 90%
B	Good	>= 80%
C	Fairly Good	>= 70%
D	Moderate	>= 50%
E	Weak	>= 40%
F	Poor	< 40%

The category "Moderate" is a bit larger than the other categories to create a Gaussian-like normal distribution.

It might be the case that some metric can't be measured for some code. Possible reasons for this are the lack of appropriate tooling or crashes of the applied tools. The percentage of lines of code that is measured for a metric is called the metric coverage. The score for a metric for some code

for which a metric fails is 0. For instance, if the metric score for coding standards is 84.00% but only 80.00% of the lines of code could be checked for coding standards, then the final score will be 80.00% of 84.00% is 67.20%.

The 8 metrics are measured and valued in the following way.

Code coverage. Code coverage is measured by taking the average of the available decision, branch and statement coverage figures. At least one of these three coverage types should be available. This is based on research done by Steve Cornett [13]. Function coverage is not taken into account because it is too easy to achieve high code coverage scores for this coverage type.

The following formula is applied to value code coverage:

score = min(0.75 * test_coverage + 32.5, 100)

The definition of the code coverage score is based on the fact that a code coverage above 90% is perfect (score = 100). Improving the code coverage if it is above 90% is not worthwhile. On the other hand, if the code coverage is above 10%, i.e. if at least some tests are performed, the score should be in category E (score = 40). Between 40 and 100, the score is evenly distributed.

C/C++ header files are excluded from code coverage scores because they usually don't contain any code and would otherwise have an unfair negative impact.

According to this formula the mapping to the code quality categories is as follows.

Code Coverage	TQI Score	Level
>= 76.7%	>= 90%	A
>= 63.3%	>= 80%	B
>= 50%	>= 70%	C
>= 23.3%	>= 50%	D
>= 10%	>= 40%	E
< 10%	< 40%	F

Table 1: Code Coverage Scores

Abstract Interpretation. Abstract interpretation results are measured by taking all errors found by the abstract interpreter. The resulting set of errors is mapped to a scale between 0% and 100% via the TIOBE compliance factor definition (see section 5). The following formula is applied to the compliance factor to get the scores for abstract interpretation:

score = 100 - 50 * log10(100 – 0.99 * compliance_factor(abstract_interpretation))

Abstract interpretation errors are considered to be very important, so its perfect score (score = 100) means there are no abstract interpretation errors at all. A compliance less than 85% indicates that there are too many abstract interpretation errors, thus having a poor score (score < 40). The score is distributed logarithmically.

According to this formula the mapping to the code quality categories is as follows.

Abstract Interpretation Compliance	TQI Score	Level
>= 99.41%	>= 90%	A
>= 98.47%	>= 80%	B
>= 96.99%	>= 70%	C
>= 90.91%	>= 50%	D
>= 85.00%	>= 40%	E
< 85.00%	< 40%	F

Table 2: Abstract Interpretation Scores

Code Security. Code security is simply measured as the compliance to the available code security rules in the used code checkers. It is assumed that the code security rules have been categorized in severity levels. The resulting set of code security rule violations is mapped to a scale between 0 and 100 via the TIOBE compliance factor definition (see section 5) using the following formula:

score = 100 - 50 * log20(400 – 3.99 * compliance_factor(security_violations))

According to this formula the mapping to the code quality categories is as follows.

Security Compliance	TQI Score	Level
>= 99.79%	>= 90%	A
>= 99.42%	>= 80%	B
>= 98.74%	>= 70%	C
>= 95.24%	>= 50%	D
>= 91.13%	>= 40%	E
< 91.13%	< 40%	F

Table 3: Code Security Scores

Compiler warnings. Compiler warnings are measured by running the compiler used at the highest possible warning level. If more than one compiler is used (e.g. because code is generated for multiple platforms), the warnings of all compilers are combined. If a file fails for one of the compilers in case multiple compilers are used, there will be no penalty, provided that the file compiles at least for one of the compilers.

Since different compilers check for different compiler warnings, it is not sufficient to use the number of compiler warnings as input for the score. Hence, the set of compiler warnings should be normalized, based on the number of different checks a compiler performs and the severity of these checks. TIOBE uses its compliance factor for this (see section 5), which is a figure between 0 (no compliance) and 100 (complete compliance, i.e. no compiler warnings). A brief summary of the way the TIOBE compliance factor is calculated is given in the next section.

Once the compliance factor is known, the following formula is applied to determine the score for compiler warnings:

score = 100 - 50 * log10(100 – 0.99 * compliance_factor(compiler_warnings))

This rather complex formula is based on the observation that most compilers have lots of different warnings and most of these warnings don't occur in the software. Hence, the compliance will be high most of the time. That's why a logarithmic function is used.

According to this formula the mapping to the code quality categories is as follows.

Compiler Warning Compliance	TQI Score	Level
>= 99.41%	>= 90%	A
>= 98.47%	>= 80%	B
>= 96.99%	>= 70%	C
>= 90.91%	>= 50%	D
>= 85.00%	>= 40%	E
< 85.00%	< 40%	F

Table 4: Compiler Warning Scores

Coding standards. It is important to make sure that as many coding standard rules as possible are automated by code checkers. For this metric only automated rules are taken into account. It is assumed that the coding rules have been categorized in severity levels. The resulting set of coding rule violations is mapped to a scale between 0 and 100 via the TIOBE compliance factor definition (see section 5). Coding standards are mapped on a normative scale by using the formula:

score = compliance_factor(coding_standard_violations)

Coding Standard Compliance	TQI Score	Level
>= 90%	>= 90%	A
>= 80%	>= 80%	B
>= 70%	>= 70%	C
>= 50%	>= 50%	D
>= 40%	>= 40%	E
< 40%	< 40%	F

Table 5: Coding Standard Scores

Cyclomatic complexity. The definition of cyclomatic complexity has been given by McCabe [14]. This definition is also used in this article. For the TQI score, the average cyclomatic complexity (cc) per function is taken. This is to reflect that it is no problem if some functions are complex, provided that this is a minority of the total number of functions. The average cyclomatic complexity is mapped on a normative scale by using the formula:

score = 6400 / (cc^3 – cc^2 - cc + 65)

The definition of the cyclomatic complexity score is based on the following assumptions:

- If the average cyclomatic complexity is 1 the score should be 100
- If the average cyclomatic complexity is 3 the score should be 80
- If the average cyclomatic complexity is 5 the score should be 40
- If the average cyclomatic complexity is infinite the score should be 0

The average cyclomatic complexity of more than 2 billion lines of industrial software code as checked by TIOBE is 4.52.

According to this formula the mapping to the code quality categories is as follows.

Cyclomatic Complexity	TQI Score	Level
<= 2.44	>= 90%	A
<= 3.00	>= 80%	B
<= 3.48	>= 70%	C
<= 4.43	>= 50%	D
<= 5.00	>= 40%	E
> 5.00	< 40%	F

Table 6: Cyclomatic Complexity Scores

Code duplication. This metric is calculated by counting the number of syntactically equivalent chains of 100 tokens (default for most tools). A token is the atomic building block of a programming language. Examples of tokens are identifiers (e.g. “status”), keywords (e.g. “return”), operators (e.g. “&&”) and delimiters (e.g. “{” or “;”). The total number of tokens that is part of a duplicated chain is taken and expressed as a percentage of the total number of tokens of the system.

The following token chains are excluded from code duplication:

- Comments
- Spacing
- Sequences of data literals, e.g. array initializations
- C/C++ header files, since these could contain duplications because of interface inheritance
- C# using directives

Code duplication is mapped on a normative scale by using the formula:

score = min(-40 * log10(code_duplication) + 80, 100)

This score definition is based on the assumption that less than 1% code duplication is considered to be good, whereas more than 10% code duplication is considered to be very poor. A logarithmic scale has been applied.

According to this formula the mapping to the code quality categories is as follows.

Code Duplication	TQI Score	Level
<= 0.56%	>= 90%	A
<= 1.00%	>= 80%	B
<= 1.78%	>= 70%	C
<= 5.62%	>= 50%	D
<= 10.00%	>= 40%	E
> 10.00%	< 40%	F

Table 7: Code Duplication Scores

Fan out. The fan out is measured by counting the *average* number of imports per module. This measurement is language dependent. For C and C++ the number of include statements is used, for Java the number of import statements. Wild cards in Java import statements appear to be difficult because these statements import several modules at once. That is why we choose to count these statements as 5. The situation is even more complex for C# because it uses a different import mechanism. The “using” statement in C# imports a complete name space, which could consist of hundreds of classes whereas only a few of these

are actually used. For this reason, we demand to count the actual number of unique dependencies per file for C#.

It is important to differentiate between external and internal fan out. External fan out concerns imports from outside the software system, whereas internal fan out is about references within the system itself. External imports are mainly applied to reuse existing software and is thus much better than internal imports. Hence, internal imports have 4 times more negative impact on the TQI for fan out than external imports.

The average fan out of a software system is mapped on a normative scale by using the formula:

score = 100/2^((8 * internal fan_out + 2 * external fan_out)/100)

According to this formula the mapping to the code quality categories is as follows. In order to get a general idea of the impact, it is assumed that the ratio between internal and external imports is 1:1.

Fan Out	TQI Score	Level
<= 3.04	>= 90%	A
<= 6.44	>= 80%	B
<= 10.29	>= 70%	C
<= 20.00	>= 50%	D
<= 26.43	>= 40%	E
> 26.43	< 40%	F

Table 8: Fan Out Scores

7 TIOBE Quality Indicator

The 8 code quality metrics defined in this article all help to get a complete picture of the code quality before release. However, not all code quality metrics are equally important. For instance, a low code coverage has much more impact on quality than a high fan out rate. This section defines how the 8 metrics are combined into one overall code quality figure, called the TIOBE Quality Indicator (TQI).

The metrics are combined by weighing them. The weights are based on empirical evidence. It is important to note that TIOBE has started research to correlate software defects to code quality metrics for the more than 2 billion lines of code it measures each day. Once this research has been finished, the weighing will be more solidly founded on statistical data.

The 8 metrics are weighted as follows.

Metric	Weight
Code Coverage	20%
Abstract Interpretation	20%
Code Security	15%
Compiler Warnings	15%
Cyclomatic Complexity	10%
Coding Standards	10%
Code Duplication	5%
Fan Out	5%

Table 9: Weights of TQI metrics

7.1 Scope

Not all code is subject to the TQI. The following kinds of code are excluded:

- Generated code
- External/third party code
- Test code

7.2 Recommended TQI Levels

What TQI level should one aim for? That depends on your application domain. Different domains have different quality constraints.

Application Domain	TQI Level Goal
A software bug might result in massive death (> 100)	A
A software bug might result in death	B
A software bug might result in considerable financial loss	C
Anything else	D

Table 10: Recommended TQI Levels

For instance, for avionics we recommend TQI level A but for administrative software we only require TQI level D. The recommended levels for various application domains can be found in the table below.

Application Domain	TQI Level Goal
Avionics, Defense	A
Space, Medical, Automotive	B
Semiconductors, Banking	C
Administrative	D

Table 11: Recommended TQI Levels Application Domains

8 Conclusions

The TIOBE Quality Indicator (TQI) is a pragmatic way to get an overview of the quality of software code before release or even before system testing. The indicator combines the most well-known code quality metrics by defining how they are measured and how the outcome of the resulting measurements should be judged. Based on this a software system is labelled between A (outstanding quality) and F (poor quality).

Appendix A References

[1] Boehm, Barry W.; Philip N. Papaccio, "*Understanding and Controlling Software Costs*", IEEE Transactions on Software Engineering, v. 14, no. 10, October 1988, pp. 1462-1477.

[2] ISO, "*Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models*", ISO/IEC 25010:2011, 2011, obtainable from https://www.iso.org/standard/35733.html.

[3] Wikipedia, "*Cyclomatic Complexity*", extracted July 2012, obtainable from https://en.wikipedia.org/wiki/Cyclomatic_complexity.

[4] Wikipedia, "*Duplicated Code*", extracted July 2012, obtainable from https://en.wikipedia.org/wiki/Duplicate_code.

[5] Wikipedia, "*Code Coverage*", extracted July 2012, obtainable from https://en.wikipedia.org/wiki/Code_coverage.

[6] Wikipedia, "*Abstract Interpretation*", extracted July 2012, obtainable from https://en.wikipedia.org/wiki/Abstract_interpretation.

[7] Wikipedia, "*Coding Conventions*", extracted July 2012, obtainable from https://en.wikipedia.org/wiki/Coding_standard.

[8] Henry, S.; Kafura, D., "*Software Structure Metrics Based on Information Flow*", IEEE Transactions on Software Engineering Volume SE-7, Issue 5, September 1981, pp. 510–518.

[9] OWASP, "OWASP top 10 - 2013, The ten most critical web application security risks", extracted December 2016, obtainable from https://www.owasp.org/index.php/Top_10_2013.

[10] CERT, "CERT Secure Coding", extracted December 2016, obtainable from https://www.cert.org/secure-coding/.

[11] Jansen, Paul; Krikhaar, Rene; Dijkstra, Fons, "Towards a Single Software Quality Metric – The Static Confidence Factor", 2006, obtainable from https://www.tiobe.com/content/paperinfo/DefinitionOfConfidenceFactor.html.

[12] Wikipedia, "*European Union energy label*", extracted July 2012, obtainable from https://en.wikipedia.org/wiki/European_Union_energy_label.

[13] Cornett, Steve, "*Code Coverage Analysis*", obtainable from https://www.bullseye.com/coverage.html.

[14] McCabe, Thomas J., "*A Complexity Measure*", IEEE Transactions on Software Engineering Volume SE-2, Issue 4, December 1976, pp. 308–320.

[15] ISO, "*Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Measurement of system and software product quality*", ISO/IEC 25023:2016, 2016, obtainable from https://www.iso.org/standard/35747.html.

[16] ISO, "*Information technology — Software measurement — Software quality measurement — Automated source code quality measures*", ISO/IEC 5055:2021, 2021, obtainable from https://www.iso.org/standard/80623.html.

Appendix B Reviewers

The following persons have contributed to this document.

Name	Company
Johan van Beers	Philips
Ron van Dijk	Glencore
Zhu Diqi	Huawei
Rick Everaerts	Philips
Rob Goud	TIOBE Software
Gultekin Gulcur	Conclusion
Rene van Hees	Thales
Laurens Jansen	TIOBE Software
Nicolas de Jong	TIOBE Software
Benjamin Jurg	TIOBE Software
Alfred Kamper	TIOBE Software
Kostas Kevrekidis	TomTom
Marco Louwerse	Océ Technologies
Randy Marques	Randy Marques Consultancy
Bart Meijer	Thermo Fisher
Goce Naumoski	ASML
Jan van Nunen	TIOBE Software
Ben Ootjers	Unisys
Joep Peeters	TIOBE Software
Guillaume Puthod	Precilog
Dennie Reniers	TIOBE Software
Ben van Rens	Océ Technologies
Minghze Shi	Technical University of Eindhoven
Bram Stappers	TIOBE Software
Maikel Steneker	TIOBE Software
Tiger Teng	TIOBE Software
Walfried Veldman	ASML