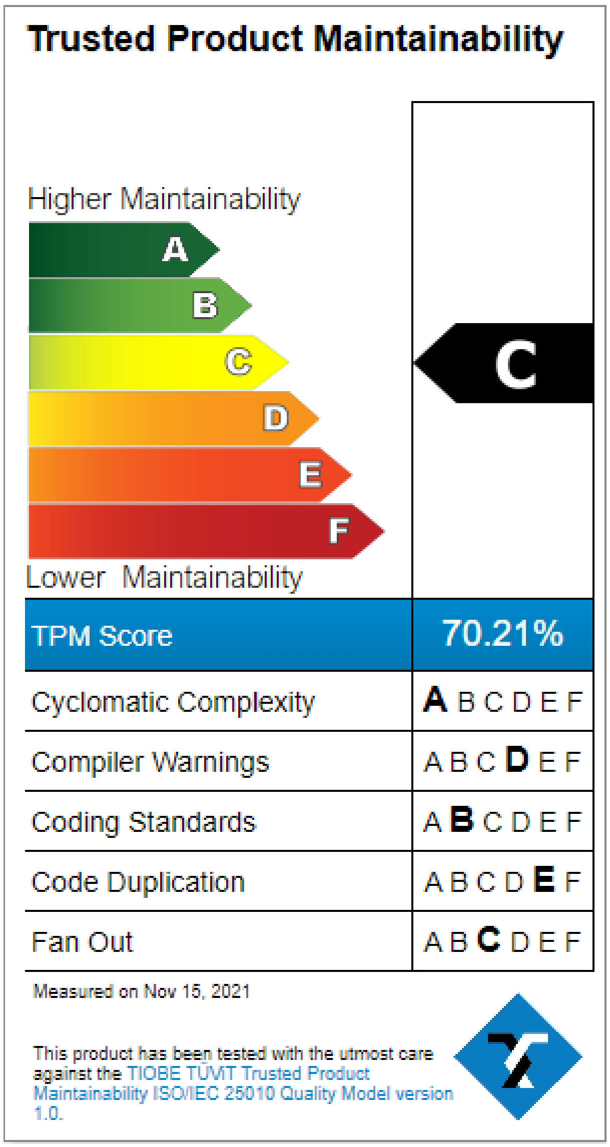
TIOBE TÜViT

Trusted Product Maintainability

ISO/IEC 25010 Quality Model



Trusted Product Maintainability

TPM Score	70.21%
Cyclomatic Complexity	**A** B C D E F
Compiler Warnings	A B C **D** E F
Coding Standards	A **B** C D E F
Code Duplication	A B C D **E** F
Fan Out	A B **C** D E F

Measured on Nov 15, 2021

This product has been tested with the utmost care against the TIOBE TÜViT Trusted Product Maintainability ISO/IEC 25010 Quality Model version 1.0.

Document ID: TIOBE-20211016.1
Version: 1.2 (approved)
Date: 16-Dec-2021
Author: Paul.Jansen@tiobe.com

Approved by TIOBE: Benjamin.Jurg@tiobe.com
Approved by TÜViT: M.Bueser@tuvit.de

Table of Contents

1 Introduction

Complex software systems tend to be very large nowadays. For instance, a modern car contains more than 100 million lines of code [1] and it is expected to increase to 1 billion lines the next few years due to autonomous driving and security measures in the code [13]. Important quality characteristics for such large software systems are Reliability and Security. But there is more. Imagine that 100 million lines of code is a book of about 1.8 million pages of text if printed out [2]. Who is able to maintain such a vast amount of pages? With this in mind, it shouldn't come as a surprise that the maintenance costs of a software system take about 90% of the total software life time costs [3]. So Maintainability is another important quality characteristic that needs to be taken seriously.

The goal of this document is to define a computational and qualification model for software maintenance based on the ISO/IEC 25010 standard about software product quality [4]. The defined approach is based on scientific evidence, more than 20 years of experience in this field, and the analysis of more than 1 billion lines of commercial software code that are checked each day. The model is approved both by TIOBE and TÜViT and is called the "TIOBE TÜViT Trusted Product Maintainability ISO/IEC 25010 Quality Model" (TPM) model. The objective of the model is to have a standard that allows certification of products with respect to software maintainability based on its static code quality. Products that meet the criteria and have been evaluated by a recognized evaluation body can receive a Trusted Product Maintainability certificate from the certification body of TÜViT.

The next section starts with the definition of software maintainability according to the ISO/IEC 25010 standard together with its implications. Section 3 describes how 5 maintainability metrics are selected to measure software maintainability according to the ISO/IEC 25010 definition. After that, in section 4, the selected software metrics are defined in more detail together with their relation to the software maintainability subcharacteristics. The subject of section 5 is to define how the metrics are calculated and how their scores are defined. Finally, section 6 defines the overall score of the Trusted Product Maintainability standard.

2 ISO/IEC 25010 Quality Model

In this section, it is discussed in more detail what quality model is used to measure Trusted Product Maintainability.

Various quality models have been proposed to define the maintainability of software systems, of which the most well known is the ISO/IEC 25010 standard on software product quality [4]. This standard defines 8 main quality characteristics, maintainability being one of them. The standard defines maintainability as "The degree of effectiveness and efficiency with which the product can be modified." The maintainability characteristic is divided into the following sub-characteristics in the ISO/IEC 25010 standard:

- **Modularity**. 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 in more than one system, or in building other assets.
- **Analysability**. The degree of effectiveness and efficiency with which it is possible to assess the impact on a product or system of an intended change to one or more of its parts, or to diagnose a product for deficiencies or causes of failures, or to identify parts to be modified.
- **Modifiability**. The degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality.
- **Testability**. The degree of effectiveness and efficiency with which test criteria can be established for a system, product or component and tests can be performed to determine whether those criteria have been met.

The ISO/IEC 25010 standard helps as a starting point to determine quality. It has 2 main drawbacks, however:

- The standard does not specify how to measure quality characteristics. Note that there is an ISO standard 25023 [5] that defines metrics to measure the ISO/IEC 25010, but most of these metrics are at behavioral level and not at software code level. Recently, ISO/IEC 5055 [6] has been introduced, which is much better in this respect, but is not sufficiently balanced, e.g. complexity of the code is at the same level as missing default statements for a switch statement.
- There is no qualification model. Even if you would be able to measure the quality characteristics, you still need to have some qualification scheme, i.e. what is considered good or bad? When do you qualify?

The TIOBE / TÜViT Trusted Product Maintainability ISO/IEC 25010 Quality Model definition aims to overcome these shortcomings.

3 Selected Software Metrics

Measuring ISO/IEC 25010 characteristics can be done in various ways. Examples are performing user tests, conducting manual reviews, and applying software metrics. Whatever method is selected, it should adhere to the following sound scientific criteria. The applied methods should be (see [14][15][16][17][18][19][20]):

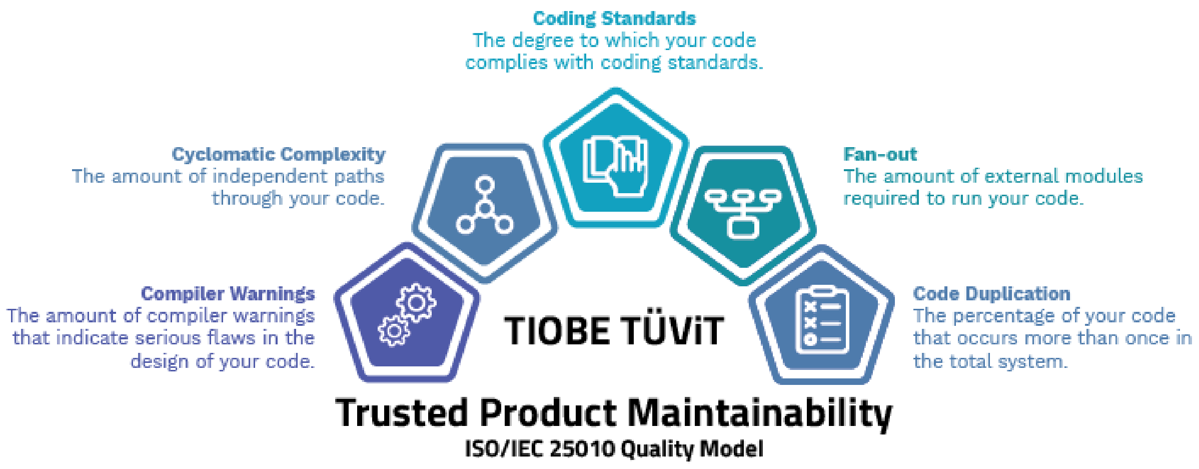
- *Consistent/objective/repeatable*. This means that the same circumstances should result in the same measurement value. For instance, manual code review is not consistent/objective/repeatable because different persons will review the same code in a different way.
- *Economical: easy to compute and can be automated*. Measurement computation should be easily

understood, the method of computing the measurement should be clearly defined. If the measurements can be measured in an automated way measurement collection is economical.

- *Predictive*. There should be a correlation between the measurement and the characteristic it is supposed to predict. Such a correlation can be determined statistically. Note that this criterion is very hard to determine for a software measurement.
- *Support all kinds of software/programming languages*. If a measurement is only available for one programming language it can't be applied in general. For instance, defining a measurement that counts the number of try-with-resources statements in the code will only be applicable for Java and will be of no use in case multiple languages are involved.
- *Understandable/Simple*. Software measurements that are hard to explain and understand are also hard to comply with because users don't know what to improve. Infamous examples of these measurements are Halstead complexity [21] and language scope [22].

The idea is to select a sufficient large set of independent and well-known software metrics that have a clear correlation with the 5 maintainability sub characteristics. The following scientific papers have been input to come to a selection: [23][24][25][26][27][28][29][30][31][32][33][34][35][36][37][38][39][40][41][42]. It is clear from these papers that there is still a lot of research to be done to come to the best possible set of software metrics to measure maintainability of software. Based on the current state of affairs the following metrics have been selected:

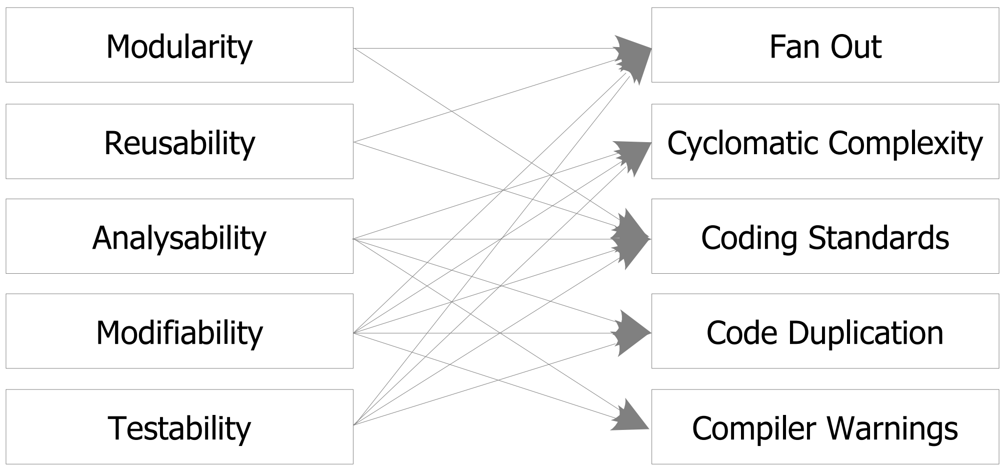
1. Cyclomatic complexity [7]
2. Compiler warnings [8]
3. Coding standards compliance [9]
4. Code duplication [10]
5. Fan out [11]


Coding Standards
The degree to which your code complies with coding standards.
Cyclomatic Complexity
The amount of independent paths through your code.
Fan-out
The amount of external modules required to run your code.
Compiler Warnings
The amount of compiler warnings that indicate serious flaws in the design of your code.
TIOBE TÜViT
Code Duplication
The percentage of your code that occurs more than once in the total system.
Trusted Product Maintainability
ISO/IEC 25010 Quality Model

One of the metrics that is often quoted in this context is size of the software. The reasoning is that the larger the size of a software system, the less maintainable it becomes. There are a couple of reasons why size of the software doesn't qualify for this quality model:

- The metric is not compositional. Suppose you have a system of 4 million lines of code that consists of 4 subsystems of 1 million lines of code each. If you assess the total system, it will be considered less maintainable than if you assess the maintainability of the individual subsystems.
- Size is not conceptually indicating a lack of maintainability. If all the software metrics mentioned above have a good score, e.g. the system has low fan out (high modularity) and low complexity, it can be perfectly maintainable.

The ISO/IEC 25010 maintainability sub characteristics are mapped to these software metrics in the following way.



The selected software metrics and their relation to the various ISO/IEC 25010 Maintainability sub characteristics are explained in more detail in the next section.

4 Definition Software Metrics

This section defines the 5 selected software metrics for maintainability in more detail, including its relation to maintainability.

4.1 Cyclomatic Complexity

One of the oldest software metrics is cyclomatic complexity: it has been proposed by Thomas J. McCabe in 1976. 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.

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

```
 1: public int getValue(int param1)
 2: {
 3:   int value = 0;
 4:   if (param1 == 0)
 5:   {
 6:     value = 4;
 7:   }
 8:   else
 9:   {
10:     value = 0;
11:   }
12:   return value;
13: }
```

The cyclomatic complexity of the function “getValue” at line 1 is 2 (one path through “then” and one through “else”).

Cyclomatic complexity has impact on the following ISO/IEC 25010 maintainability sub characteristics:

- Analysability. If code is complex, it becomes hard to understand for human beings.
- Modifiability. It is hard to judge what the consequences are if complex code is changed, because software engineers can't keep all possible paths in their minds.
- Testability. The more paths through the code, the more test cases need to be written.

4.2 Compiler Warnings

In order to execute a software program on a computer, it first must 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 maintainability issues.

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

```
 1: int func(int i) {
 2:   if (i = 0) {
 3:     return -1;
 4:   }
...
58: }
```

Most compilers will complain about the assignment in the if condition at line 2. Probably a comparison was meant instead, but because this is not for certain, it could be a problem for software engineers that have to maintain this code.

Compiler warnings have impact on the following ISO/IEC 25010 maintainability sub characteristics:

- Analysability. There are compiler warnings that warn about code that is probably not intended as it is stated. If such a compiler warning is left in the code, it will make others wonder what is exactly meant.
- Modifiability. There are compiler warnings that warn about strange situations in the code. If somebody modifies such a piece of code, then this might have unexpected consequences.

4.3 Coding Standards Compliance

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.

An example of a coding standard violation is shown below.

```
 1: int abs(int i) {
 2:   int result = 0;
 3:
 4:   if (i < 0) {
 5:     goto end;
 6:   }
 7:   result = i;
 8: end:
 9:   return result;
10: }
```

Any C coding standard will complain about the goto statement used at line 5. It is considered bad practice to use goto statements.

Coding standard violations have impact on the following ISO/IEC 25010 maintainability sub characteristics:

- Modularity. Most coding standards have rules about how to make sure code remains modular, e.g. don't use global variables, encapsulate objects correctly, use constants as much as possible, etc.
- Reusability. There are coding standard rules that support reusability, e.g. hide implementation details in the interface, write proper documentation, avoid in-out parameters, etc.
- Analysability. One of the powers of a coding standard is to avoid code that is not comprehensible. Rules such as don't use goto statements, avoid multiple side effects in the same statement, use proper exception handling, add sufficient logging, etc. help to create code that is understandable.
- Modifiability. Coding standards usually have rules that increase the ease of modifying code without too much risk. Rules in this respect are keeping functions small with only one intention, don't hide variables with the same name in a smaller scope, don't use break or continue statement, etc.
- Testability. Some rules of a coding standard make it easier to test code. Examples of such rules are to limit the number of arguments of a function, having only one return value (no in/out parameters), and have as few exit points from a function as possible.

4.4 Code Duplication

Sometimes, it is very tempting for a software engineer to copy some 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 new functionality), it is very likely that the other parts ought to be changed as well. The chances are high that such copies are forgotten to be changed.

Code duplication has impact on the following ISO/IEC 25010 maintainability sub characteristics:

- Analysability. If there is a lot of code duplication, more code must be read to understand what is happening. Moreover, it might be the case that one is looking at the wrong copy to analyse a certain situation because there is no single point of truth any more in case of much code duplication.
- Modifiability. Code duplication has most impact on modifiability of the code. It might be the case that a defect has been fixed, but the defect is still in because the 4 other copies have not been changed.
- Testability. The more code duplication, the more code paths must be tested. If code duplication is resolved by abstraction, less code needs to be tested.

4.5 Fan Out

Software programs consist of parts that interact with each other. Depending on the used programming language, application domain, and the company, these parts might have different names. Most commonly used names are components, modules and subsystems. These parts might exists of a single file each, or groups of files, directories, etc. In order to avoid any confusion, the Trusted Product Maintainability methodology takes files as atomic entities to have dependencies between.

The fan out metric indicates how many different other files are used by a certain file. If a file needs a lot of other modules to function correctly (high fan out), there is a high interdependency with other files, thus making code less modifiable.

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

```java
package com.acme.plugins.eclipse.analyzer;

import java.io.IOException;
import java.util.Map;

import org.apache.commons.exec.CommandLine;
import org.apache.commons.exec.DefaultExecutor;
import org.apache.commons.exec.ExecuteException;
import org.apache.commons.exec.ExecuteResultHandler;

import com.acme.plugins.eclipse.console.IConsole;
import com.acme.plugins.eclipse.console.Console;
import com.acme.plugins.eclipse.util.EclipseUtils;

public class Analyzer implements IAnalyzer {
```

The example contains 9 dependencies on other files. There are 6 dependencies that make use of third party libraries (so called external fan out), whereas 3 dependencies start with "com.acme" and are referring to own code (so called internal fan out), provided that the company is named "acme".

Fan out has impact on the following ISO/IEC 25010 maintainability sub characteristics:

- Modularity. The fan out metric contributes most to the modularity subcharacteristic. If a file contains a lot of dependencies to other files, it is not easy to move this file, because all dependencies should be resolved in the new situation.
- Reusability. If a file contains a lot of dependencies to other files, then this file can only be reused if all the other files are also available. Moreover, the dependencies of all these other files must be taken into account as well and might even require even more files to be taken into scope.
- Modifiability. If a software system contains a high interconnection between files, changing one part might impact other parts and vice versa. A system with a high fan out is hard to change.
- Testability. One of the keys of testability is that parts can be test in isolation. If the fan out is high and there are a lot of dependencies, testing becomes harder. Think about an extra database connection mock that simulates a connection including data transfer.

5 Measuring and Evaluating Metric Values

This section defines how the 5 metrics of the previous section are measured. In order to quantify the results of code quality measurements, a scale between 0% and 100% (called the *score*) is introduced. This is an absolute score to enable comparisons. There are 6 different quality levels with respect to maintainability distinguished associated with the score, ranging from A (high) to F (low).

5.1 Cyclomatic Complexity

The definition of cyclomatic complexity has been given by McCabe [7]. This definition is also used in this document. The most important design decision that has been taken here is that the *average* cyclomatic complexity *per function* is used. This is explained below:

- Average. Every software system will have some complex functions, e.g. functions that implement state machines or long algorithms. That is no problem as long as this is compensated by a sufficient number of simple functions. However, if every function has many paths and is hard to understand, the code is too complex. Taking the average cyclomatic complexity reflects this reasoning.
- Per function. A function is the smallest semantic entity of a program. If you have to take the average for cyclomatic complexity, functions are the most obvious choice.

If the *absolute* cyclomatic complexity for a software system would have been taken instead of the *average*, this metric would become similar to calculating the lines of code of a system. It has already been explained in section 3 why size of the software is not a good indicator of maintainability.

The definition of the cyclomatic complexity score is based on the following empirical 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%

Given these assumptions, the average cyclomatic complexity (cc) per function is mapped on a normative scale by using the formula

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

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

Cyclomatic Complexity	Score	Quality 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 1: Cyclomatic Complexity Scores

Empirical data. The average cyclomatic complexity of more than 1 billion lines of industrial software code as checked by TIOBE is 3.88 (level D). The distribution of all these industrial projects is as follows:

Cyclomatic Complexity	% Projects
A	35.1%
B	13.3%
C	6.2%
D	10.1%
E	9.1%
F	26.2%

Table 2: Industrial Averages Cyclomatic Complexity

5.2 Compiler Warnings

Compiler warnings are measured by running the compiler used by the software system 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, i.e. the union is taken. 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, 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 appendix A. The compiler warnings that are applied including their severity levels are published on the website of TIOBE.

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

score = max(100 - 50 * log10(101 – compliance_factor(compiler_warnings)), 0)

This formula is based on the observation that most compilers have lots of different warnings and most of these warnings don't occur in the source code because they will only trigger in very specific cases. Hence, the compliance will be high, in most of the cases. That's why a logarithmic function is used.

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

Compliance Factor	Score	Quality Level
>= 99.42%	>= 90%	A
>= 98.49%	>= 80%	B
>= 97.02%	>= 70%	C
>= 91.00%	>= 50%	D
>= 85.15%	>= 40%	E
< 85.15%	< 40%	F

Table 3: Compiler Warning Scores

Empirical data. The average compiler warnings score of more than 1 billion lines of industrial software code as checked by TIOBE is 77.80% (level C). The distribution of all these industrial projects is as follows:

Compiler Warnings	% Projects
A	66.7%
B	12.6%
C	4.9%
D	7.2%
E	2.1%
F	6.6%

Table 4: Industrial Averages Compiler Warnings

5.3 Coding Standards Compliance

A coding standard is a document that contains a set of rules a software engineer should follow. These rules might vary in nature, ranging from naming conventions and lay out rules to rules about program design, type correctness, and pitfalls of the used language to avoid.

In order to standardize, the TIOBE TÜViT Trusted Product Maintainability ISO/IEC 25010 quality model will apply “standard” coding standards, one for each language. These “standard” coding standards are published on the TIOBE website. The standards are based on the industry (e.g. the Philips C# coding standard or the ASML C coding standard) or on a sensible selection of rules defined by leading code checkers (e.g. the default configuration of eslint to check TypeScript code).

Note that naming conventions and layout rules are not selected. This is because these style guide related rules might differ between projects, and none of them is superior, it is just a matter of convention.

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 appendix A). Coding standards are mapped on a normative scale by using the formula:

score = compliance_factor(coding_standard_violations)

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

Compliance Factor	Score	Quality Level
>= 90%	>= 90%	A
>= 80%	>= 80%	B
>= 70%	>= 70%	C
>= 50%	>= 50%	D
>= 40%	>= 40%	E
< 40%	< 40%	F

Table 5: Coding Standard Scores

Empirical data. The average coding standard score of more than 1 billion lines of industrial software code as checked by TIOBE is 75.69% (level C). The distribution of all these industrial projects is as follows:

Coding Standards	% Projects
A	55.9%
B	15.2%
C	9.8%
D	11.2%
E	2.2%
F	5.7%

Table 6: Industrial Averages Coding Standards

5.4 Code Duplication

This metric is calculated by counting the number of semantically equivalent chains of 100 tokens. 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 number of 100 tokens have been chosen, because experiments have shown that less tokens, e.g. 50, result in too many false positives.

The following token chains are excluded from code duplication:

- Comments

- Spacing
- C/C++ header files, since these could contain duplications because of interface inheritance
- C# using directives

The score definition for code duplication 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 to get the best possible distribution.

Based on this code duplication is mapped on a normative scale by using the formula

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

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

Code Duplication	Score	Quality 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

Empirical data. The average coding standard score of more than 1 billion lines of industrial software code as checked by TIOBE is 8.48% (level E). The distribution of all these industrial projects is as follows:

Code Duplication	% Projects
A	29.7%
B	5.4%
C	7.9%
D	24.3%
E	20.4%
F	12.3%

Table 8: Industrial Averages Code Duplication

5.5 Fan Out

The fan out is measured by counting the *average* number of imports per file. 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. 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. That's why we demand for C# to count the actual number of unique dependencies per file.

There is also a difference 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. Based on experiments and checking available data, it has been decided to let internal imports have 4 times more negative impact on the score 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 levels 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, thus multiplying by the average of 8 and 2, which equals 5.

Fan Out	Score	Quality 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 9: Fan Out Scores

Empirical data. The average coding standard score of more than 1 billion lines of industrial software code as checked by TIOBE is 8.19 (level C). The distribution of all these industrial projects is as follows:

Fan Out	% Projects
A	32.7%
B	18.4%
C	13.1%
D	22.0%
E	6.7%
F	7.1%

Table 10: Industrial Averages Code Duplication

6 Global Definition TPM Qualification

In this section, the qualification system for TIOBE TÜViT Trusted Product Maintainability (TPM) is defined. In the first subsection, the scope of what software code is assessed is defined. This is also known as the target of evaluation. After that in section 2, the impact of failed measurements are discussed, followed by a definition of weights of the TPM metrics in section 6.3. Finally in section 6.4, the qualification itself is defined including aggregation of scores.

6.1 Scope

The TIOBE TÜViT Trusted Product Maintainability is only measured for *manually written own production* code. Production code is defined as all code that will be part of the product that is created. This means that the following code is **excluded** from the scope and is not part of the target of evaluation:

- Generated code. This code type is excluded because it is not manually written. Generated code is not supposed to be changed. Moreover, generated code doesn't need to be maintainable or at least can have a lower level of maintainability because it is never updated by human beings.
- External/third party code. This code type is excluded because only own code is subject to this assessment. The reason for this is that external/third party code is a given and can't be changed because there are most probably copy rights involved. So it doesn't need to be maintained.
- Test code. Test code is excluded because it will not be part of the final product. Usually, other quality constraints hold for test code if compared to production code. Maintainability of test code is important in order to be able to understand and adapt tests quickly. However, the product needs to be certified, not its test code.

The product description shall describe the target of evaluation (what files are in scope) and indicate the product name and its version number.

6.2 Failing Measurements

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

6.3 Metric Weights

The 5 code quality metrics defined in this document all help to get a complete picture of product maintainability. The metrics are combined by weighing them. The 5 metrics are weighted as follows.

Metric	Weight
Cyclomatic Complexity	20%
Compiler Warnings	20%
Coding Standards	20%
Code Duplication	20%
Fan Out	20%

Table 11: Weights of Trusted Product Maintainability metrics

All metrics have an equal weight. There are several reasons for this:

- Based on the relations between the ISO/IEC 25010 subcharacteristics for maintainability and the metrics as shown in section 3, all metrics contribute almost equally. Some have less relations, but that is compensated by having a stronger relation.
- Having equal weight keeps the model simple.
- There is insufficient research in the field of "software metric to ISO/IEC 25010 subcharacteristic mapping" to justify another distribution of weights. By using the quality model as defined in this document, data will be collected to improve these weights.

6.4 Aggregation and Qualification

Now that all ingredients have been defined, the overall qualification can be calculated. For this, the scores of all metrics for all programming languages should be combined, resulting in an overall score. Note that the defined quality model is independent of any programming language, so any set of programming languages can be used.

The overall score is associated with a level. See the table below.

Quality Level	Score
A	>= 90%
B	>= 80%
C	>= 70%
D	>= 50%
E	>= 40%
F	< 40%

Table 12: Mapping Scores to Levels

Empirical data. The average Trusted Product Maintainability score of more than 1 billion lines of industrial software code as checked by TIOBE is 66.46% (level D). The distribution of all these industrial projects over the various metrics is as follows:

Software Metric	Level
Cyclomatic Complexity	D
Compiler Warnings	C
Coding Standards	C
Code Duplication	E
Fan Out	C

Table 13: Industrial Averages per Metric

The 1 billion lines of industrial software code consists of almost 5,000 different projects. These are distributed in the following way:

Trusted Product Maintainability	% Projects
A	44.0%
B	13.0%
C	8.4%
D	15.0%
E	8.1%
F	11.6%

Table 14: Industrial Averages Trusted Product Maintainability

It might feel inconsistent that the average score of all projects is level D, whereas more than half of the projects have a score of either level A or level B. The reason for this is that table 13 is normalized for project size. Large projects tend to have a lower maintenance score.

Qualification

A software project qualifies for TIOBE TÜViT Trusted Product Maintainability if it has an *overall score of at least 70% (level C).*

7 Conclusions

The TIOBE TÜViT Trusted Product Maintainability ISO/IEC 25010 Quality Model is a pragmatic way to get an overview of the maintainability of software code before release, or even earlier, during the software development process. The indicator combines well-known code quality metrics by defining how they are measured and how the outcome of the resulting measurements should be evaluated. Based on this, a software system is labelled between A and F. These levels represent descending quality levels in terms of software maintainability.

Appendix A Compliance Factor

Some of the metrics discussed in this document 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 well, whereas if it is 50% this is considered bad programming practice.

However, for 2 of the 5 TIOBE TÜViT Trusted Product Maintainability metrics there is no such straightforward mapping. These are:

- Compiler warnings
- Coding standards violations

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 one 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 coding standard rules.

In the definition of the compliance factor, the notion of "severity level" has been formalized. The most severe issues are at severity level 1, less important issues are at severity level 2, etc. There is no upper bound to severity levels. Usually coding standards and compiler warnings are defined in a range from 1 to 10. The severity levels are determined by the coding standard document or the compiler. More information about severity levels can be found in [43].

The formal definition of the compliance factor is as follows:

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

where the definition of weighted violations is:

weighted violations = $$\sum_{i=1}^{maximum\ severity\ level} violations(i)/4^{(i-1)}$$

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

$$\textbf{average rules per level} = \frac{\sum_{i=1}^{maximum\,severity\,level} rules(i)}{maximum\,severity\,level}$$

A detailed explanation of this formula is outside the scope of this document. It can be found in a separate document [12]. TIOBE has used this definition for more than 20 years in all their code quality monitoring and assessment projects and it appears to work well for this kind of maintainability metrics.

Appendix B References

[1] Doug Newcomb, "*The Next Big OS War Is in Your Dashboard*", Wired, March 12, 2012, obtainable from https://www.wired.com/2012/12/automotive-os-war/.

[2] Jeff Desjardins, "*How Many Millions of Lines of Code Does It Take?*", Visual Capitalist, February 8, 2017, obtainable from https://www.visualcapitalist.com/millions-lines-of-code/.

[3] Sayed Mehdi Hejazi Dehaghani and Nafiseh Hajrahimi, "*Which factors affect software projects maintenance cost more?*" in Acta Inform. Med. 21, 1 (2013), 63.

[4] 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.

[5] 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.

[6] 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.

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

[8] Alex Allain, "*Why Compiler Warnings are Your Friends?*", Cprogramming.com, 2019, obtainable from https://www.cprogramming.com/tutorial/compiler_warnings.html.

[9] Wikipedia, "*Coding Conventions*", obtained from https://en.wikipedia.org/wiki/Coding_standard in October 2021.

[10] Wikipedia, "*Duplicated Code*", obtained from https://en.wikipedia.org/wiki/Duplicate_code in October 2021.

[11] Wayne P.Stevens, Glenford J. Myers, and Larry LeRoy Constantine, "*Structured design*", IBM Systems Journal. 13 (2): 115–139. doi:10.1147/sj.132.0115, June 1974.

[12] 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.

[13] Jaguar Land Rover (2019) *Jaguar Land Rover finds the teenagers writing the code for a self-driving future*. Available from https://media.jaguarlandrover.com/news/2019/04/jaguar-land-rover-finds-teenagers-writing-code-self-driving-future.

[14] IEEE (1998) *1061-1998 - IEEE Standard for a Software Quality Metrics Methodology*.

[15] Cem Kaner and Walter P. Bond (2004) "Software Engineering Metrics: What Do They Measure and How

Do We Know?" in *10th Internation Software Metrics Symposium, Metrics 2004*. Available from http://www.kaner.com/pdfs/metrics2004.pdf.

[16] Capers Jones (2017) *A Guide to Selecting Software Measures and Metrics,* Auerbach Publications.

[17] Danish Thakur (2020) *Software Metrics in Software Engineering*. Available from http://ecomputernotes.com/software-engineering/software-metrics.

[18] Rational Software (2001) *Measurement Plan Guidelines*. Available from https://sceweb.uhcl.edu/helm/RationalUnifiedProcess/process/modguide/md_metri.htm.

[19] Elvira-Maria Arvanitoua, Apostolos Ampatzogloua, Alexander Chatzigeorgioub and Paris Avgeriou (2016) "Software metrics fluctuation: a property for assisting the metric selection process" in *Information and Software Technology Volume 72 Number 4*. Available from http://www.cs.rug.nl/search/uploads/Publications/arvanitou2016ist.pdf.

[20] Pankaj Patel (2020) *Software Measurement and Metrics*. Available from https://www.geeksforgeeks.org/software-measurement-and-metrics/.

[21] Maurice H. Halstead (1977) *Elements of Software Science* Amsterdam: Elsevier North-Holland.

[22] Helmar Kuder (2008) *HIS Source Code Metrics version 1.3.1.*

[23] Joren Wijnmaalen, Cuiting Chen, Dennis Bijlsma, Ana-Maria Oprescu (2019) "The Relation between Software Maintainability and Issue Resolution Time: A Replication Study". In: Anne Etien (eds.): *Proceedings of the 12th Seminar on Advanced Techniques Tools for Software Evolution*, Bolzano, Italy, July 8-10 2019. Available from http://ceur-ws.org/Vol-2510/sattose2019_paper_11.pdf.

[24] Peggy Skiada, Apostolos Ampatzoglou, Elvira-Maria Arvanitou, Alexander Chatzigeorgiou, Ioannis Stamelos(2019) "Exploring the Relationship between Software Modularity and Technical Debt". Available from https://ruomo.lib.uom.gr/bitstream/7000/121/3/skiada2018seaa.pdf.

[25] Reinier Vis, Dennis Bijlsma and Haiyun Xu (2020) "SIG/TÜViT Evaluation Criteria Trusted Product Maintainability", version 12.0. Available from https://www.softwareimprovementgroup.com/wp-content/uploads/2020-SIG-TUViT-Evaluation-Criteria-Trusted-Product-Maintainability.pdf.

[26] Yanja Dajsuren, Mark G.J. van den Brand and Alexander Serebrenik (2013) "Modularity Analysis of Automotive Control Software". Available from https://ercim-news.ercim.eu/en94/special/modularity-analysis-of-automotive-control-software.

[27] Siti Rochimah, I Made B. Gautama and Rizky J. Akbar (2019) "Refactoring the Anemic Domain Model using Pattern of Enterprise Application Architecture and its Impact on Maintainability: A Case Study". In: *IAENG International Journal of Computer Science, 46:2, IJCS_46_2_16*. Available from http://www.iaeng.org/IJCS/issues_v46/issue_2/IJCS_46_2_16.pdf.

[28] Joost Visser, Sylvan Rigal, Rob Van Der Leek, Pascal Van Eck and Gijs Wijnholds (2016) "What is maintainability?". Available from https://www.oreilly.com/content/what-is-maintainability/.

[29] Michail D.Papamichail, Themistoklis Diamantopoulos and Andreas L. Symeonidis (2019) "Measuring the Reusability of Software Components using Static Analysis Metrics andReuse Rate Information" in *Journal of Systems and Software 2019-9*. Available from https://issel.ee.auth.gr/wp-content/uploads/2019/09/2019mpapamicJSS.pdf.

[30] Özlem Akalin and Feza Buzluca, (2018) "A quality model for evaluating maintainability of object-oriented software systems" in *International Conference on Advanced Technologies, Computer Engineering and Science (ICATCES'18), May 11-13, 2018 Safranbolu, Turkey*. Available from https://web.itu.edu.tr/buzluca/icatces_2018.pdf.

[31] Harald Foidl and Michael Felderer (2018) "Integrating software quality models into risk-based testing". In *Software Quality Journal volume 26, pages809–847(2018)*. Available from

https://link.springer.com/article/10.1007/s11219-016-9345-3.

[32] Muhlis Sul Aen and Alif Finandhita (2019) Quality Assessment of CV. Fredavelop Notarial Deed Fidusia Software. Available from https://elib.unikom.ac.id/files/disk1/736/jbptunikompp-gdl-mukhlissul-36797-9-unikom_m-s.pdf.

[33] Michael Stal (2014) "Refactoring Software Architectures". *In Agile Software Architecture - Aligning Agile Processes and Software Architectures*, pages 63-82. Available from *https://www.sciencedirect.com/science/article/pii/B9780124077720000034*.

[34] Joyce M. S. França and Michel S. Soares (2015) "SOAQM: Quality Model for SOA Applications based on ISO/IEC 25010" in *Proceedings of the 17th International Conference on Enterprise Information Systems (ICEIS-2015), pages 60-70*. Available from https://www.scitepress.org/papers/2015/53691/53691.pdf.

[35] Joyce Meire da Silva França (2017) *Software Architecture based on a Quality Model to Develop Service Oriented Applications*. Available from https://repositorio.ufu.br/bitstream/123456789/21334/1/SoftwareArchitectureBase.pdf.

[36] Manuel I. Capel, Anna Grimán and Eladio Garví (2017) *Design Patterns for Software Evolution Requirements*. Available from *https://biblioteca.sistedes.es/submissions/uploaded-files/JISBD_2017_paper_63.pdf*.

[37] M. Riaz, E. Mendes, and E. Tempero (2009) in "A Systematic Review of Software Maintainability Prediction and Metrics" in *3rd International Symposium on Empirical Software Engineering and Measurement (ESEM 2009), pp. 367 - 377.*

[38] B. Luijten, J. Visser, & A. Zaidman (2010) "Faster defect resolution with higher technical quality of software" in *Proc. of the 4th International Workshop on Software Quality and Maintainability (SQM'10).*

[39] M. Alshayeb, 2009, "Empirical investigation of refactoring effect on software quality" in *Information and Software Technology, vol. 51, no. 9, pp. 1319-1326.*

[40] I. Samoladas, G. Gousios, D. Spinellis, and I. Stamelos, (2008) "The SQO-OSS quality model: Measurement based open source software evaluation" in *Open Source Development, Communities and Quality, Vol. 275, pp. 237-248*.

[41] Stefan Burger and Oliver Hummel (2012) "Lessons Learnt from Gauging Software Metrics of Cabin Software in a Commercial Airliner", in *International Scholarly Research Notices / 2012 / Article ID 162305*. Available from https://www.researchgate.net/publication/258403375_Lessons_Learnt_from_Gauging_Software_Metrics_of_Cabin_Software_in_a_Commercial_Airliner.

[42] Saleh A. Almugrin (2015) *Definitions and Validations of indirect Package Coupling in an agile, object-oriented Environment*. Available from https://businessdocbox.com/Marketing/83964123-Definitions-and-validations-of-metrics-of-indirect-package-coupling-in-an-agile-object-oriented-environment-a-dissertation-submitted.html.

[43] Paul Jansen (2020) *Severity Levels for Coding Standards*. Available from https://www.tiobe.com/articles/2020/severity-levels-for-codingstandards/.

Appendix C Reviewers

The following persons have reviewed this document.

Name	Company
Manfred Büser	TÜV Informationstechnik GmbH
Benjamin Jurg	TIOBE Software
Christoph Sutter	TÜV Informationstechnik GmbH

Appendix D Contact Details

In case of any questions or remarks about this document please contact:

- **Paul Jansen**, CEO, TIOBE Software, Victory House II, Esp 401, 5633 AJ Eindhoven, the Netherlands, Phone: +31 40 400 2800, Email: Paul.Jansen@tiobe.com

- **Christoph Sutter**, Head of Certification Body, TÜV Informationstechnik GmbH, Am TÜV 1, 45307 Essen, Germany, Phone: +49 201 8999 582, Email: C.Sutter@tuvit.de