Software Improvement Group

SIG/TÜViT Evaluation Criteria Trusted Product Maintainability: *Guidance for producers*

Version 11.0

September 19, 2019

GETTING SOFTWARE RIGHT

Colophon

Reinier Vis MSc
+31 20 314 0950
r.vis@sig.eu

Dennis Bijlsma MSc
+31 20 314 0950
d.bijlsma@sig.eu

dr. Haiyun Xu
+31 20 314 0950
h.xu@sig.eu

Table of contents

1 Introduction

This document is a companion to the SIG/TÜViT Evaluation Criteria Trusted Product Maintainability [SIG 2019]. The *SIG/TÜViT Evaluation Criteria* for the quality mark *TÜViT Trusted Product Maintainability* are intended for the standardized evaluation and certification of the technical quality of the source code of software products. The purpose of such evaluation and certification is to provide an instrument to developers for guiding improvement of the products they create and enhance, and to acquirers for comparing, selecting, and accepting pre-developed software.

This guidance document provides explanation to software producers about the measurement method of SIG applied for evaluation. For each measurement area, the threshold measurement values are provided that are required for eligibility of certification at the level of 4 stars.

2 Definitions

The measurements are performed at different aggregation levels, as depicted in the figure below. This chapter defines the terminology used to describe those levels.

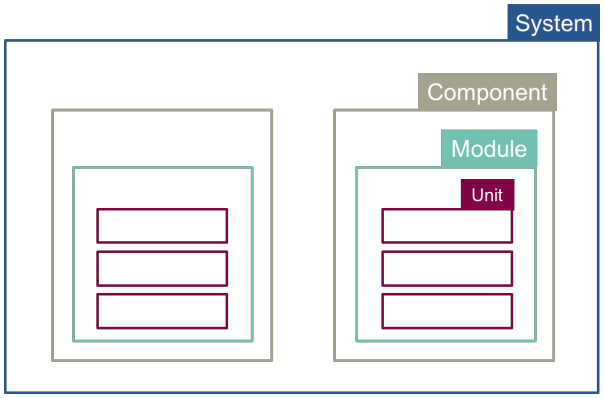

Figure 1: Aggregation levels at which measurements are performed

2.1 System

The software system for which certification is sought. The system boundaries are defined by the System Owner, the organization that holds property rights over the software system.

2.2 Component

A component is a subdivision of a system in which source code modules are grouped together based on a common trait. Often components consist of modules grouped together based on a shared technical or functional aspect. Top-level components are the first subdivision of a software system as seen by the developers, and are therefore visible in the source code, typically through the directory structure or through naming conventions. Where in the remainder of this document we refer to component, it is intended to mean top-level component.

2.3 Module

The notion of module is defined as a delimited group of declaration. This may correspond to different terms in each programming language, but generally it corresponds to a file. For example, in object-oriented languages, declarations are grouped into classes, which in turn usually correspond to a file.

2.4 Unit

The notion of unit is defined as the smallest named piece of executable code. This may correspond to different terms in each programming language. For example, for object-oriented languages (e.g. Java, C#), *methods* are regarded as units whereas for procedural languages (e.g. C), units correspond to *procedures*, or *functions*. Constructs *without* a name cannot be re-executed by calling them, and are therefore not considered units.

3 Guidance for producers

For each measurement area, the threshold measurement values are provided that are required for eligibility of certification at the level of 4 stars. Note that these clarifications are meant as guidance for software producers, providing sufficient conditions for satisfying the measurement model that is used during evaluation by SIG.

3.1 Volume

Larger software products are deemed to be harder to maintain. To maximize the rating of a product with respect to volume, the producer should strive to keep its source code concise.

For the evaluation of the volume property, the software product's rebuild value is estimated on the basis of the number of lines of source code. To make the lines of code of software artefacts written in different programming languages comparable to each other, they are normalized on the basis of industry averages.

To be eligible for certification at the level of 4 stars, the total rebuild value of the product should not exceed 26 man-years. The following table lists the number of lines of code that is produced on average for some industry standard technologies.

LANGUAGE	LOC IN 26 MAN YEARS
C++	207,000
C#	240,000
Java	234,000
JavaScript	197,000
Python	165,000
Ruby	142,000
TypeScript	193,000

3.2 Duplication

Software products with less (textual) duplication are deemed to be easier to maintain. To maximize the rating of a product for the duplication property, the software producer should avoid multiple occurrences of the same fragments of code.

For the evaluation of the duplication property, the percentage of redundant lines of code is estimated. A line of code is considered redundant if it is part of a code fragment (larger than 6 lines of code) that is repeated literally (modulo white-space) in at least one other location in the source code.

To be eligible for certification at the level of 4 stars, the percentage of redundant lines of code for each programming language used should not exceed 4.8%.

3.3 Unit size

Software products where more source code resides in large units are deemed to be harder to maintain. To maximize the rating of a product for the unit size property, the software producer should avoid large units.

The size of the units of a software product is determined by counting the number of lines of code within each unit.

To be eligible for certification at the level of 4 stars, for each programming language used the percentage of lines of code residing in units with more than 15 lines of code should not exceed 41.3%. The percentage in units with more than 30 lines of code should not exceed 17.9%. The percentage in units with more than 60 lines should not exceed 5.2%.

3.4 Unit complexity

Software products where more source code resides in units with high logical complexity are deemed to be harder to maintain. To maximize the rating of a product for the unit complexity property, the software producer should avoid units with high complexity.

The complexity of each unit is determined in terms of the McCabe cyclomatic complexity number. This number represents the number of non-cyclic paths through the control-flow graph of the unit and can be calculated by counting the number of decision points that are present in the source code.

To be eligible for certification at the level of 4 stars, for each programming language used the percentage of lines of code residing in units with McCabe complexity number higher than 5 should not exceed 20.2%. The percentage of lines of code in units with McCabe complexity number higher than 10 should not exceed 6.8%. The percentage of lines of code in units with McCabe complexity number higher than 25 should not exceed 0.7%.

3.5 Unit interfacing

Software products where more source code resides in units with large interfaces are deemed to be harder to maintain. To maximize the rating of a product for the unit interfacing property, the software producer should avoid units with large interfaces.

The size of the interface of a unit can be quantified as the number of parameters (also known as formal arguments) that are defined in the signature or declaration of a unit.

To be eligible for certification at the level of 4 stars, for each programming language used the percentage of lines of code residing in units with 3 or more parameters should not exceed 13.7%. The percentage in units with 5 or more parameters should not exceed 2.8%. The percentage in units with 7 or more parameters should not exceed 0.7%.

3.6 Module coupling

Software products where more source code resides in modules that are strongly coupled with other modules are deemed to be harder to maintain. To maximize the rating of a product for the module coupling property, the software producer should avoid having strong coupling between modules. Modules are groupings of units, such as classes that group methods or files that group functions.

The coupling of the modules of a software product can be quantified as the number of incoming dependencies, such as invocations, per module.

To be eligible for certification at the level of 4 stars, for each programming language used the percentage of lines of code residing in modules with a number of incoming dependencies above 10 should not exceed 15.7%. The percentage in modules with a number of incoming dependencies

above 20 should not exceed 8.9%. The percentage in modules with a number of incoming dependencies above 50 should not exceed 3.6%.

3.7 Component balance

The number of components of a system and their relative size compared to each other can give an indication of the breakdown of the system in functional modules. If functionality has not properly been divided across components, it becomes harder to locate functionality when making changes. If too much functionality is centralized into a single component, it also becomes harder to work on the system in parallel, or to make changes to a component without affecting other components.

The component balance can be quantified on the basis of the deviation from the situation where all the components have the same size. Systems get a lower score when components are more unequally sized relative to each other. For this calculation, the adjusted Gini coefficient on the size of all components in the system measured in LOC is used. A special case is made when the system has not divided into components at all, and consists of one monolithic component. In these cases, Component Balance will receive the lowest possible rating.

To be eligible for certification at the level of 4 stars the adjusted Gini coefficient of the sizes should not exceed 0.7.

3.8 Component independence

Each module in the system is contained within a component. Based on the dependencies between modules residing in other components the module is classified as either hidden or interface. A module is hidden if there are no incoming dependencies from modules in other components. Otherwise the module is classified as interface.

Ideally, the percentage of code that is classified as hidden is maximized as to ensure that changes within the components are not propagating to other components. Therefore, component independence is calculated as the percentage of code that is contained in modules that are classified as hidden.

To be eligible for certification at the level of 4 stars the percentage of code residing in modules with incoming cross-component dependencies should not exceed 9.4%.

3.9 Component entanglement

Each module in the system is contained within a component. Communication lines between components are expected to respect certain rules. Some of these rules are specific to the domain in which the system is used, and therefore specific and relate to the functionality provided by the system. Other communication rules are related to (technical) best practices, and apply regardless of the functionality of the system.

Component Entanglement is the percentage of communication lines between components that does not conform to these best practices. It is calculated on the system level, as the percentage of communication lines that contains anti-patterns. It considers three of such anti-patterns:

- *Cyclic dependencies occur* when component A has a dependency on component B, but component B also has dependency on component A.
- *Indirect cyclic dependencies* occur when components do not have direct cyclic dependencies, but indirectly communicate so that every component is dependent on every other component.

> *Bypassing communication layers* occurs when a component has both direct and indirect dependencies on another component.

The percentage of component entanglement is then calculated as the percentage of communication lines affected by those anti-patterns, divided by the total number of communication lines. In this calculation, each communication line has a weight based on the number of underlying dependencies.

To be eligible for certification at the level of 4 stars the this percentage should not exceed 10%.

4 References

[SIG 2019] Joost Visser, *SIG/TÜViT Evaluation Criteria Trusted Product Maintainability, Version 11.0*, Software Improvement Group, 2019.

Software Improvement Group
Fred. Roeskestraat 115
1076 EE Amsterdam
The Netherlands

+31 20 314 09 50
info@sig.eu
@sig_eu
www.sig.eu

GETTING SOFTWARE RIGHT