Software Improvement Group

SIG/TÜV NORD CERT EVALUATION CRITERIA TRUSTED PRODUCT MAINTAINABILITY: GUIDANCE FOR PRODUCERS

Version 17.0 (March 12, 2025)

SOFTWAREIMPROVEMENTGROUP.COM

Colophon

This document was reviewed and approved by the following people:

Reinier Vis
Head of evaluation laboratory
r.vis@sig.eu

Dennis Bijlsma
Head of Evaluators
d.bijlsma@sig.eu

Haiyun Xu
Quality Manager
h.xu@sig.eu

TABLE OF CONTENTS

1. INTRODUCTION

This document is a companion to the SIG/TÜV NORD CERT Evaluation Criteria Trusted Product Maintainability [SIG 2025]. The *SIG/TÜV NORD CERT Evaluation Criteria* for the quality mark *TÜV NORD CERT 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.

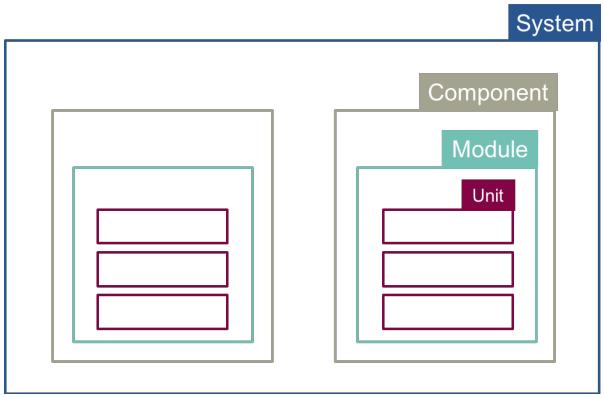


Figure 1 Aggregation levels at which measurements are performed

2.1 SYSTEM

The system consists of all software needed to achieve the overall functionality of the product.

The system consists of one or multiple components, which may or may not be deployed separately, and may be maintained by different development teams. The classification into systems and components does not depend on the independence or reusability of components in other systems.

The system owner, which is the organization that holds property rights and that applies for certification, is responsible for defining the exact system boundary during the scoping phase.

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.

Systems may contain multiple levels of components, but only top-level components are considered during the maintainability evaluation. 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. A larger system needs a more complex design and a larger team. To be eligible for certification at the level of 4 stars with a total violation count of 10. 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 3.9 person-years. The following table lists the number of lines of code that is produced on average for some industry standard technologies.

Language	Lines of code in 3.9 person years
C++	32,000
C#	39,000
Java	35,100
JavaScript	36,300
Python	25,000
Ruby	24,600
TypeScript	31,600

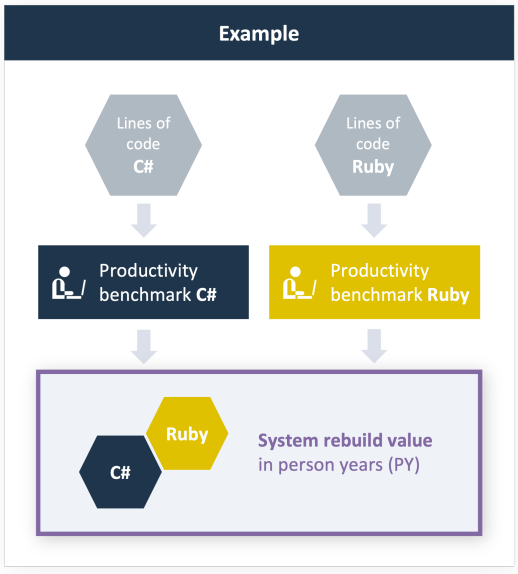


Figure 2 Example of system volume determined from lines of code for different technologies

3.2 DUPLICATION

Software products with less (textual) duplication are deemed to be easier to maintain. Any change in a piece of duplicated code (likely) needs to be made in each of the duplicate occurrences. If this is not done correctly, it leads to the risk of causing unexpected differences in behavior for different occurrences of similar functionality. Bugfixes in pieces of duplicated code are especially risky, as they can lead to the continued existence of the bug if it's not correctly applied to all duplicate occurrences. 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 code fragment is considered duplicated if it is at least 6 lines of code long and repeated literally (modulo white-space) in at least one other location in the source code. With optimal code reuse the fragment would still occur in the codebase exactly once. So the amount of redundant code is the amount of occurrences of the duplicated fragment minus one times the length of the fragment.

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 5.6%.

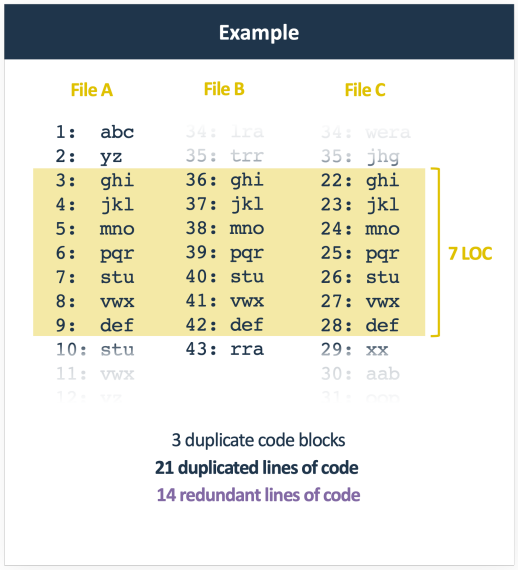


Figure 3 Example of duplication between files

3.3 UNIT SIZE

Software products where more source code resides in large units are deemed to be harder to maintain. Short units generally have a single responsibility, making them easier to test and reuse. Shorter units also allow for an easier and better overview and understanding of their inner workings. 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 47.1%.
- The percentage in units with more than 30 lines of code should not exceed 23.1%.
- The percentage in units with more than 60 lines should not exceed 8.3%.

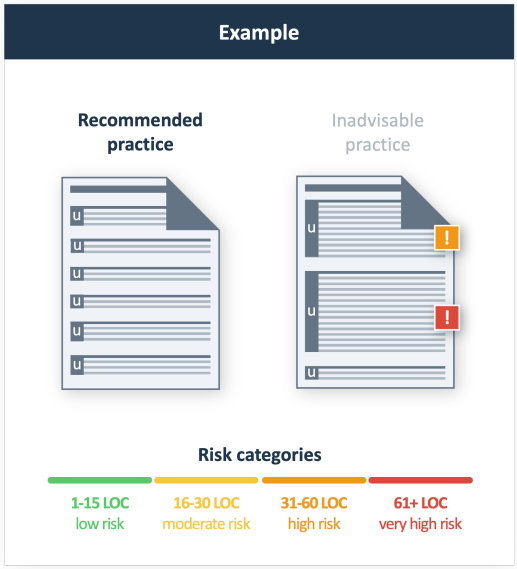


Figure 4 Overview of Unit Size risk categories; it is a recommended practice to keep units short

3.4 UNIT COMPLEXITY

Software products where more source code resides in units with high logical complexity are deemed to be harder to maintain. Complex units are harder to understand, increasing the difficulty of modifying them. A complex unit has a high number of possible execution paths and therefore requires more test cases to fully test it. 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 residing in units with McCabe complexity number higher than 10 should not exceed 7.3%.
- The percentage of lines of code residing in units with McCabe complexity number higher than 25 should not exceed 1.1%.

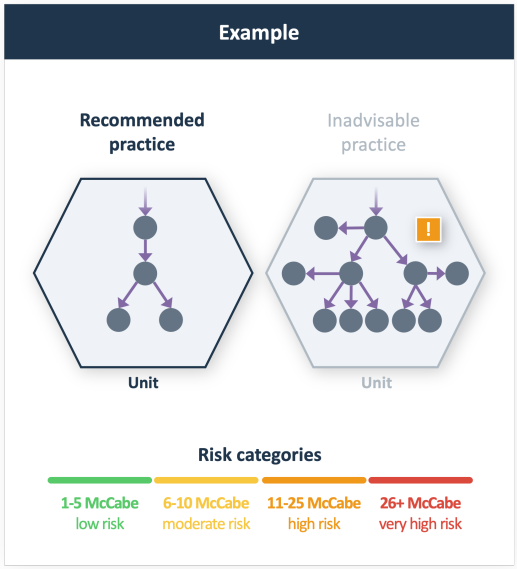


Figure 5 Overview of Unit Complexity risk categories and visualization of execution paths through a simple and a complex unit

3.5 UNIT INTERFACING

Software products where more source code resides in units with large interfaces are deemed to be harder to maintain. Units with a large interface often indicate a unit with multiple responsibilities which is harder to modify. Large interfaces are also error prone, especially if multiple parameters have the same type and can thus be accidentally supplied in the wrong order. This also impedes the speed of development. 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 15.0%.
- The percentage in units with 5 or more parameters should not exceed 3.3%.
- The percentage in units with 7 or more parameters should not exceed 0.9%.

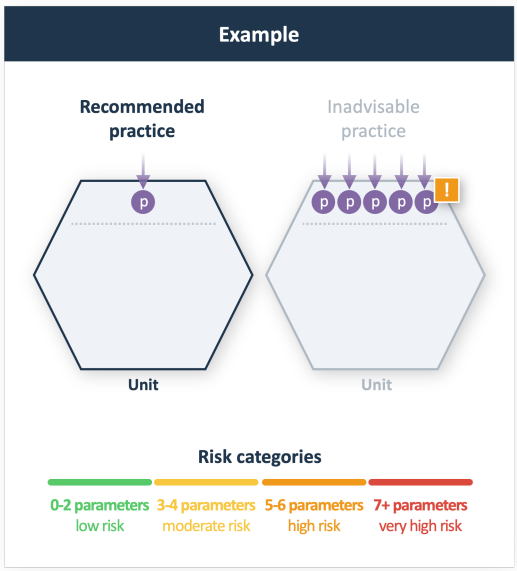


Figure 6 Overview of Unit Interfacing risk categories; it is a recommended practice to keep interfaces small

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.

Moreover, for modules that do require high(er) coupling, the software producer should keep them as small as possible, for example through abstraction of inner logic to other modules.

High module coupling is often a sign of lacking separation of concerns into separate modules. A good separation of concerns leads to modules that have a single responsibility which are easier to analyze, test, and modify. Changes to larger modules with tight coupling tend to cause a ripple effect through the codebase. Keeping these modules small through abstraction, for example by using interfaces, mitigates this effect.

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 10.0%.
- The percentage in modules with a number of incoming dependencies above 20 should not exceed 5.6%.
- The percentage in modules with a number of incoming dependencies above 50 should not exceed 1.9%.

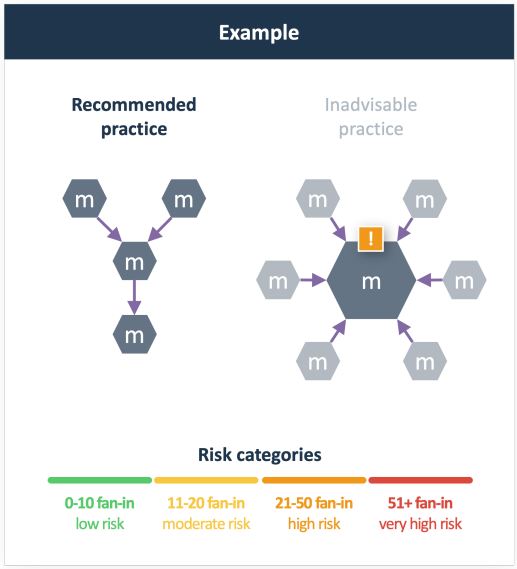


Figure 7 Overview of Module Coupling risk categories; It's a recommended practice to keep modules loosely coupled

3.7 COMPONENT ENTANGLEMENT

Each module in the system is contained within a component. Component Entanglement considers both these components and the communication lines defined between components; these are the directed lines between components over which dependencies go from one component to the other. The weight of a communication line is equal to the total number of dependencies it consists of. Every additional communication line adds additional complexity to the architecture. This makes it harder to make changes in isolation and to extend the architecture.

In addition to the *amount* of communication lines, certain *patterns* formed by them can also make the architecture more complex and consequently development more difficult.

Therefore, inter-component communication should be limited and is expected to respect certain rules. Some of these rules are specific to the domain in which the system is used, 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 measured as the combination of two numbers:

- Communication density
- Communication violation degree

The latter is computed from three violations:

- Cyclic dependencies
- Indirect cyclic dependencies
- Transitive dependencies

Below are the definitions for these terms.

Communication density is calculated by dividing the number of communication lines between components by the number of connected components. Connected components are components with at least one incoming or outgoing communication line. Components that do not feature any communication at all are thus excluded, as these do not participate in the communication.

Cyclic dependencies occur when component A has a dependency on component B, but component B also has dependency on component A. This makes it harder to maintain components in isolation: changes to one of the components can have an impact on the other and vice versa.
The weight of the violation is equal to the weight of the lowest weighted communication line involved in the violation, as removing this communication line, will remove the cyclic dependency in a way that most likely requires the lowest effort. So, in the example below the weights of the communication lines involved are 1 and 10, which means that the violation will receive a weight of 1.

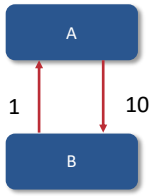


Figure 9 Example of cyclic dependencies

Indirect cyclic dependencies occur when a set of components does not have direct cyclic dependencies, but the communication lines between the involved components form a cycle. Resulting in every involved component being either directly or indirectly dependent on every other involved component. The problems associated with maintaining such components are similar to “regular” cyclic dependencies.
Indirect cyclic dependencies are considered a less severe violation than direct cyclic dependencies. The reason is that the maintenance problem is not directly bidirectional: a change to B will impact A, but a change to A will not automatically have direct impact on B. Compared to direct cyclic dependencies, in this situation it is somewhat easier to maintain the components independently.

As the goal of finding violations is to find communication lines that are most likely not intended in the we do not want to consider the communication lines that are already indicated as unintended in the “regular” cyclic dependency violations when looking for indirect cyclic dependencies. Therefore, as the first step is to remove all lightest weight communication lines in “regular” cyclic dependencies from the graph, if both communication lines in a specific cyclic dependency are equally weighted, we remove both. Given the resulting subgraph, the approach towards detecting indirect cyclic dependency violations, and the approach for calculating the violation weight, are identical to the cyclic dependency detection described above.

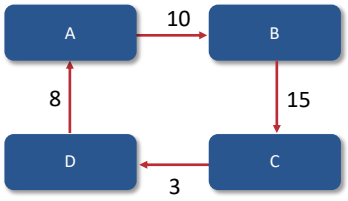


Figure 10 Example of indirect cyclic dependencies

Transitive dependencies occur when a component has both indirect and direct (transitive) dependencies to another component. Three criteria need to be met for the dependencies through a specific communication line to be considered as violating transitive dependencies:

- The communication line is transitive. In the example below, component B has a direct dependency to component D, but it also has an indirect dependency via component C. This makes the direct dependency transitive.

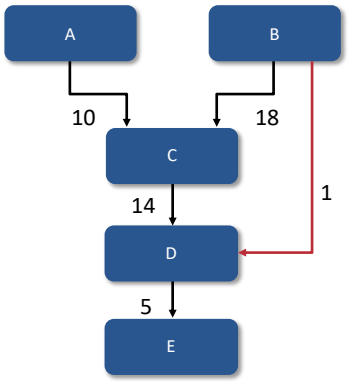


Figure 11 Example of transitive dependencies, with the violating dependency appearing in red

- The transitive communication line is to a component that also has at least one outgoing communication line. In the previous example, the dependency from component B to component D is considered a violation. Whereas a dependency from component B to component E would not have been considered a violation, because component E does not have outgoing dependencies. The reason for this exception is that components without outgoing dependencies are typically intended for code reuse, such as libraries or components containing common code.

- The transitive communication line must be the lowest weighted communication line of all distinct communication lines on all paths from the first to the second component connected by the transitive communication line.

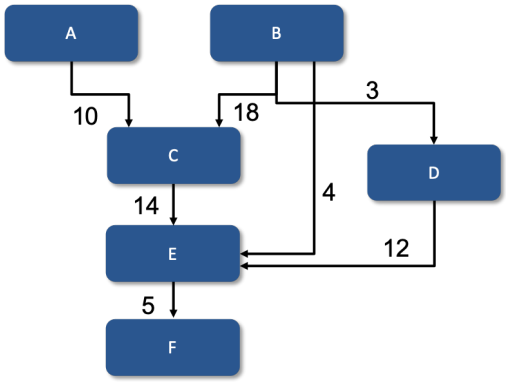


Figure 12 Example of transitive dependencies, without violations

In the example in Figure 12, the dependencies from B to E are not considered violating as the weight of the communication line from B to D, is lower than the weight of the direct communication line from B to E. This exception is made as a transitive dependency violation is seen as a communication line bypassing the normal code flow. By extension we expect the normal code flow through the indirect path(s) to be higher or equally weighted than the bypassing direct communication line. As the weight of the flow through a path of multiple

communication lines is limited by the lowest weighted communication line in that path, the weight of the transitive dependency should be lower than or equal to this lowest weight communication line to be violating.

The weight of the violation is equal to the weight of the transitive communication line. So, in the first example above the weight of the violation would be 1.

Communication violation degree is calculated by adding up the weights for all violations and dividing this by the total weight of all communication lines in the component graph. Note that the above violations are checked in order, and a communication line can only be marked as a single type of violation.

For the final calculation of component entanglement, the values for communication density and communication violation degree are normalized by dividing by the maximal values for each in our benchmark and then combined to compute the final score. To maximize the rating of a product for the component entanglement property, the software producer should minimize the communication violation degree and minimize the communication density.

To be eligible for certification at the level of 4 stars, the value for Component Entanglement (calculated as described in the previous paragraph) should not exceed 0.077.

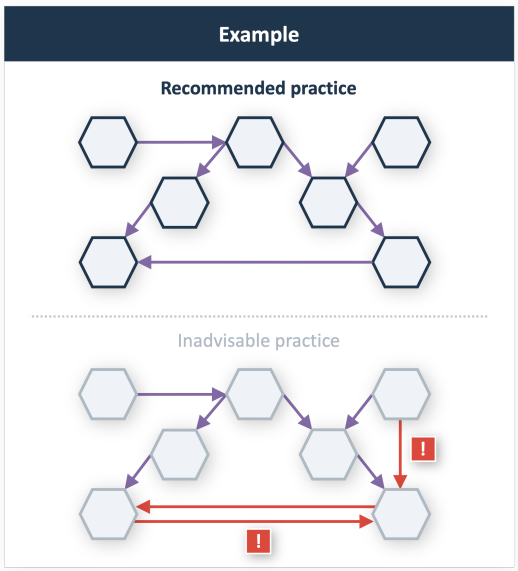


Figure 13 Example Component Entanglement; It is a recommended practice to avoid complex structures in the dependency graph and to minimize the number of communication lines

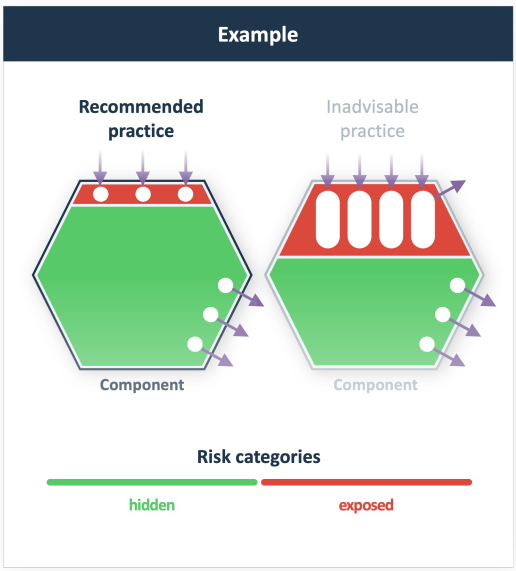
3.8 COMPONENT INDEPENDENCE

All modules in a component are classified as either hidden or exposed. A module is hidden if there are no incoming dependencies from modules in other components. Otherwise, the module is classified as exposed.

Component independence is calculated as the percentage of code that is contained in modules that are classified as hidden. Maximizing the percentage of code that is classified as hidden ensures that changes within the components are not propagating to other components, which makes maintenance easier. Good isolation of components also allows for distribution of maintenance responsibility for separate components among different teams. To maximize the rating of a product for the component independence property, the software producer should maximize the percentage of code classified as hidden.

To be eligible for certification at the level of 4 stars the percentage of code residing in modules *without* incoming cross-component dependencies should not drop below 93.7%.

Figure 8 Component Independence risk categories; It is recommended to keep the percentage of exposed code (i.e., with incoming dependencies) in components low. Note that outgoing dependencies do not influence the metric.



4. REFERENCES

[SIG 2025] Reinier Vis, *SIG/TÜV NORD CERT Evaluation Criteria Trusted Product Maintainability, Version 17.0*, Software Improvement Group, 2025.

Software Improvement Group

Fred. Roeskestraat 115
1076 EE Amsterdam
The Netherlands

www.softwareimprovementgroup.com
marketing@softwareimprovementgroup.com

Getting software right for a healthier digital world