PLCopen®

for efficiency in automation

PLCopen Training Guidelines

Guideline on Software Quality Metrics

Version 1.0 - Official Release

Date: November 07, 2023

total number of pages: 65

The following paper

Guideline on Software Quality Metrics

is an official PLCopen document. It summarises the results of the working group, containing contributions of all its members.

The present specification was written thanks to the members of the working group:

Name	**Company**
Eva-Maria Neumann	Technical University of Munich
Dr.-Ing. Juliane Fischer	Technical University of Munich
Univ.-Prof. Dr.-Ing. Birgit Vogel-Heuser	Technical University of Munich
Bernhard Reiter	CODESYS Group
Sebastian Diehm	Schneider Electric Automation GmbH
Christian Keupp	Schneider Electric Automation GmbH
Mohua Ghosh	Schneider Electric
Denis Chalon	Schneider Electric
Antonio Giusto	Tetra Pak
Chris Freiberg	BHP Group
Bugra M. Yildiz	Software Improvement Group
Michael Stiller	Fraunhofer IKS
Bill Lydon	PLCopen
Eelco van der Wal	PLCopen

Change Status List:

Version number	**Date**	**Change comment**
V 0.1	Nov. 30, 2022	First release as proposed by E. M. Neumann
V 0.2	Dec. 22, 2022	After web meeting of December 22 and multiple inputs on terminology and use case description
V 0.3	Mar. 23, 2023	Release of textual descriptions of software development workflow, use cases, and initial draft for structuring recommendations
V 0.4	Apr. 11, 2023	Release of updated template to formulate recommendations
V 0.5	May 11, 2023	After the web meeting of May 11 and multiple inputs
V 0.6	May 25, 2023	Input on terms and definitions, use cases, and recommendations
V 0.7	Jun. 21, 2023	First textually complete draft
V 0.8	Jul. 20, 2023	Merge and incorporation of feedback on first textually complete draft
V 0.99	Oct. 06, 2023	Version for internal review in the WG to prepare V 1.0
--	Nov. 02, 2023	Finalized version of the WG for publication. Not released
V. 1.0	Nov. 07, 2023	Official release of the document

Table of Contents

Table of Figures

1. Motivation and Introduction

Companies operating in the machine and plant manufacturing sector are facing ever-increasing competitive pressure to design, develop, and implement high-quality control software. As software gains importance as a functionality carrier in production systems, the reuse of high-quality control software is a crucial factor in ensuring efficient development and, consequently, ensuring long-term competitiveness. Quality management of control software is therefore of increasing importance – both for internally developed software, such as to meet customer requirements, as well as for external software, such as to compare software from subcontractors. Furthermore, continuous quality management may support the maturation of development teams since it strengthens the awareness of software quality and how it is influenced by design decisions. In addition, the process of quality assessment over time, e.g., across software versions, can aid in identifying and reversing a deterioration in quality at an early stage, for instance, by compensating for an increase in complexity resulting from software evolution. This also has the potential to save long-term costs that would be incurred, e.g., by increased maintenance efforts due to high software complexity [1].

In the field of computer science, software quality metrics have emerged as a suitable means of objectively measuring and comparing the quality attributes of software. Approaches to measuring software characteristics using metrics also exist already for control software in machine and plant manufacturing, and many platform providers already support these approaches with commercial static code analysis tools in their programming environments. In practice, however, control software developers are still reluctant to incorporate such metrics into their development workflows and to measure the quality of their control software, e.g., due to a lack of knowledge of how to use such means in their daily work. Until now, quality assessment is mostly based on the experience of software developers. Quantitative quality indicators, therefore, hold great potential to support developers in their experience-based decision-making by objectifying code. Furthermore, from a management perspective, software metrics may serve as quantitative indicators to justify the initial higher cost of software quality or to establish the basis for a cross-company benchmark.

This guideline provides support for integrating a metric-based quality assessment of control software running on Programmable Logic Controllers (PLCs) into the daily industrial routine, supporting different stakeholders in the software engineering workflow in machine and plant manufacturing companies. Existing approaches from research and tool support by platform suppliers will be used and elucidated to be applicable for various use cases and company-specific boundary conditions. Thus, insights in software quality can be achieved with minimal effort in the daily practice of PLC software development and, at the same time, the greatest possible benefit.

2. How to Use This Document

This guideline is geared towards professionals in the fields of machine and plant manufacturing, who are looking for a starting point for incorporating software quality metrics into their workflows.
This guideline focuses on IEC 61131-3-compliant control software running on industrial controllers like PLCs, with a primary focus on the boundary conditions in machine and plant manufacturing. It is anticipated that certain aspects may be adapted to other fields of PLC applications, such as building automation or construction machinery.

2.1. Aspects to Consider When Using This Guideline

Please note the following hints and disclaimers when applying the guideline:

- The focus of this guideline is on measuring a specific set of software quality attributes as highlighted: *maintainability, reusability, testability, efficiency,* and *reliability.* These attributes have been proven to be particularly relevant for PLC software in machine and plant manufacturing. However, this guideline does not claim that focusing on only these five attributes guarantees an all-encompassing quality optimization of the software.

- The objectives of this guideline are to identify metrics that are included in industrially available static code analysis tools, thereby facilitating the implementation of the recommendations in industrial settings. In this guideline we focus on the following static code analysis tools for software metrics: *CODESYS Static Analysis*, *Schneider Electric EcoStruxure Machine Code Analysis, Schneider Electric – Control Engineering Verification* (formerly Itris PLC Checker), and *Software Improvement Group Sigrid* are in focus.

- This guideline introduces typical use cases in an industrial software development workflow in machine and plant manufacturing. However, depending on the company-specific boundary conditions, there may be deviations from the presented reference workflow in practice. The focused use cases represent examples of scenarios, in which metric application is considered helpful. There is no claim for all possible use cases, only a few highly relevant and popular ones in an industrial setting.

- This guideline provides recommendations on how to quantify quality attributes using suitable metrics in the respective platforms. It is intended to provide an introduction to quality management for software developers in machine and plant manufacturing. It is not intended to cover all possible boundary conditions in all areas of application, instead it is a guide for software developers to gain valuable insights and advice for improving software quality.

- Metric results only lead to benefits when they are correctly interpreted by practitioners. Hence, this guideline does not advocate the optimization of PLC software solely based on numerical values, but rather supports the identification of outstanding values or timely deterioration of values. It is advisable to automate the metric calculation and to review the outcomes to deliberate on anomalous metric results and inform all pertinent relevant stakeholders (such as responsible software developers, quality managers, or project managers).

2.2. Structure of the Guideline

In Section 3, the theoretical background, terms, and definitions used in this guideline are introduced.

Section 4 provides a reference workflow for software development in machine and plant manufacturing, including specific scenarios in which software metrics are expected to be beneficial.

Section 5 constitutes the fundamental component of the guideline and presents concrete recommendations for selecting appropriate metrics to quantify the fulfillment of a specific software quality attribute.

Lastly, in Section 6, the metric application is demonstrated for concrete industrial code examples.

Appendix 1 provides a mapping between the available metrics and software quality attributes.

3. Software Quality Measurement Using Metrics - Terms and Definitions

This section describes the theoretical basis for this guideline.

Section 3.1 provides the definition of software quality used in this guideline and the focused software quality attributes.

Section 3.2 motivates how metrics can support the objective measurement of these quality attributes.

Finally, Section 3.3 lists the terms and definitions used in this guideline.

3.1. Software Quality and Software Quality Attributes

Software quality models define the characteristics of software quality as well as their interconnections [2]. According to the standard IEEE 1061, software quality attributes refer to "characteristic[s] of software, or a generic term applying to quality factors, quality subfactors, or metric values". In the field of computer science, numerous quality models have been established over the past decades, such as the quality model of Boehm [3] or the Dromey's model for software product quality [4]. Among the most established quality models are the software quality model of McCall [5] and the ISO 25010 [6], which will be outlined in the following.

McCall et al. define software quality as a "general term applicable to any trait or characteristic, whether individual or generic, a distinguishing attribute which indicates a degree of excellence or identifies the basic nature of something." Based on a literature review, different quality attributes have been gathered and grouped into three main categories, namely *product revision, product transition,* and *product operation* (cf. Figure 1).

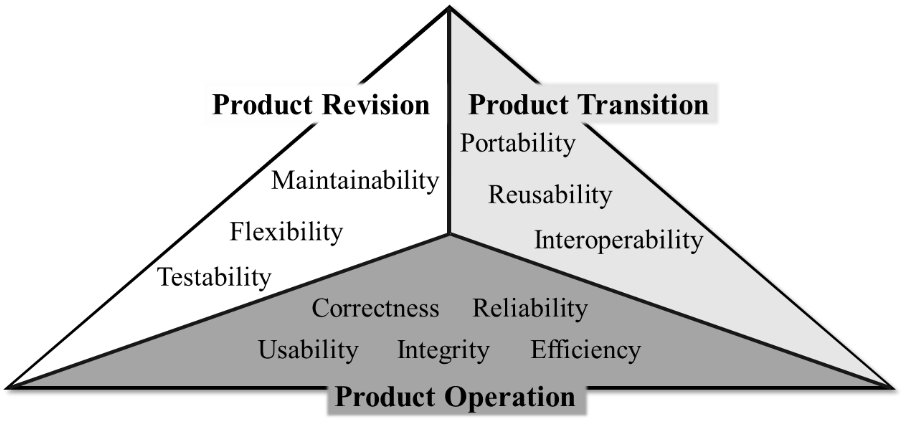


Figure 1. Overview of McCall's software quality model [5].

According to the ISO/IEC 25010, software quality is defined as "the degree to which the system satisfies the stated and implied needs of its various stakeholders, and thus provides value". The standard proposes eight quality attributes with different sub-characteristics, as visualized in Figure 2.

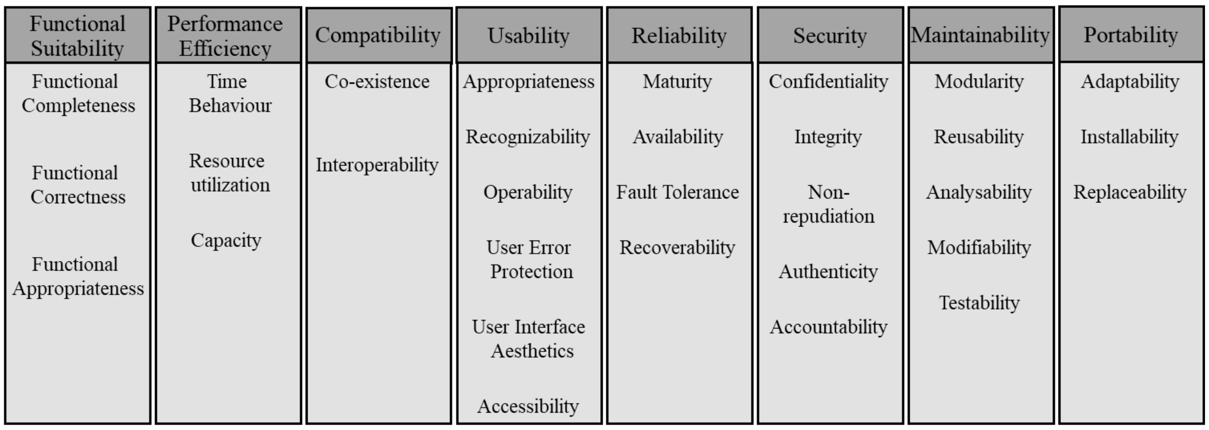


Figure 2. Overview of the software quality model according to ISO/IEC 25010 [6].

Previous investigations and code analyses conducted in various companies in the machine and plant manufacturing industry have revealed that the quality attributes *reliability, efficiency, maintainability*, *reusability*, and *testability* hold significant importance in the development of future technological trends, while simultaneously remaining adaptable for several decades [7]. The comparison of the respective definitions as per McCall and ISO 25010 in Table 1 demonstrates that they exhibit significant overlaps and complement each other. Throughout this guideline, the combination of both definitions is used.

Table 1. Comparison of definitions for selected quality attributes according to McCall [5] and ISO 25010 [6].

	Definition McCall [5]	**Definition of ISO 25010 [6]**
Reliability	Extent to which a program satisfies its **specifications** and fulfills the **user's mission objectives** *(sub-characteristic of product operation)*	Degree to which a system, product or component performs specified functions under specified conditions for a specified period of time. • Sub-characteristic **maturity**: Degree to which a system, product or component meets needs for **reliability under normal operation.**
Efficiency	The amount of **computing resources** and code required by a program to perform a function *(sub-characteristic of product operation)*	Performance relative to **the amount of resources** used under stated conditions. • Sub-characteristic **Time behavior**: Degree to which the response and processing times and throughput rates of a product or system, when performing its functions, meet requirements.
Maintain-ability	Effort required to **locate** and **fix an error** in an operational program *(sub-characteristic of product revision)*	Degree of **effectiveness** and **efficiency** with which a product or system can be **modified** to **improve** it.
Reusability	Extent to which a program can be **used in other applications** – related to the packaging and scope of the functions that programs perform.	**Reusability**: 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**. **Modularity**: 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.

	Definition McCall [5]	Definition of ISO 25010 [6]
Testability	**Effort** required to **test a program** to ensure it performs its intended function	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. *(sub-characteristic of maintainability)*

3.2. Software Quality Metrics

In computer science, software quality metrics have been proven to be a suitable means to quantify the fulfillment of software quality attributes based on the number and distribution of software properties. According to the IEEE 1061 standard [8], a software metric is defined as a "function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which software possesses a given attribute that affects its quality."

Metrics can be applied to different granularities of the software architecture. The established metrics suite from Kemerer and Chidamber [9] focuses on the structural design of object-oriented software. This implies that the metrics do not solely focus on the source code within individual Program Organization Units (POUs, as defined in Section 3.3), but rather on their structural interrelationships, such as the coupling between objects. On the other hand, some metrics focus on the internal design of a software unit's implementation (e.g., a POU), like regarding its complexity (e.g., McCabe's Cyclomatic Complexity [10] or Halstead's complexity metrics [11]). All metrics have in common that the fulfillment of a certain quality attribute can be derived based on a certain combination and number of software properties. The degree of fulfillment of a quality attribute, such as testability, is indicated by a software metric as a numerical indicator (cf. Figure 3).

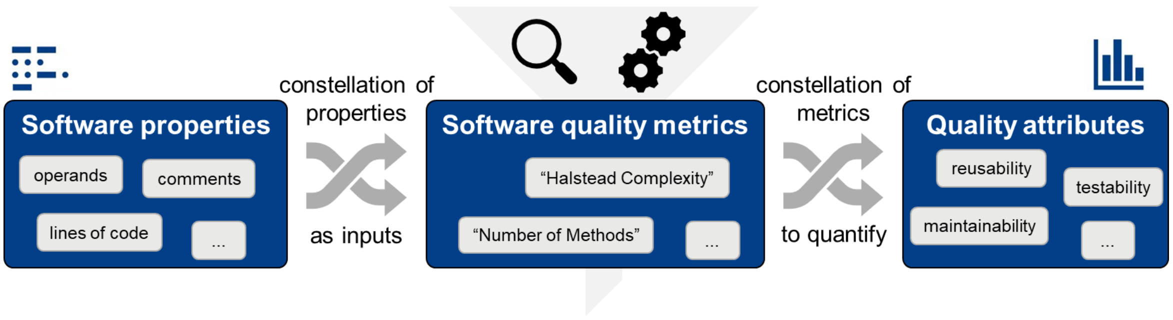


Figure 3. Interrelationship between software properties, quality attributes, and software quality metrics [12].

Within the scope of this guideline, metrics are grouped into a list of categories given below. This list of categories is derived from the available metrics in commercial static code analysis tools, as well as based on literature:

- ***Size metrics***, i.e., metrics that measure the scope of a POU, e.g., by counting parts of their implementation or considering the required memory size.
- ***Variables and POU interfaces***, i.e., metrics referring to a POU's declaration part.
- ***Code documentation***, i.e., metrics referring to comments in a POU's implementation or declaration part.
- ***Information exchange,*** i.e., metrics referring to calls and information flows to and from a POU.
- ***Software complexity,*** i.e., metrics referring to different aspects that increase the software's complexity.

- ***OO-IEC elements,*** i.e., metrics referring to language elements defined by OO-IEC such as inheritance, interfaces and properties / methods.
- ***Language-specific elements***, i.e., metrics intended for the particularities of specific IEC 61131-3-language such as Function Block Diagram (FBD) or Sequential Fucntion Chart (SFC).
- ***Reuse indicators***, i.e., metrics that give insights regarding the reuse of individual POUs in a project or library.

3.3. Terms and Definitions

In the following section, the terms and definitions used in this guideline are introduced in an alphabetic order. The definitions are derived from the IEC 61131-3 [13], the ISO/IEC/IEEE 24765 [14], definitions provided by established PLC platform providers, and general software development practices.

Acceptance Tests: A test of a system or functional unit usually performed by the purchasers on their premises after installation with the participation of the vendor to ensure that the contractual requirements are met.

Action: A Boolean variable or a collection of operations to be performed, together with an associated control structure.

Agile software development: Software development methodologies centered around the idea of iterative development, where requirements and solutions evolve through collaboration between self-organizing cross-functional teams. Enables teams to deliver value faster, with higher quality and predictability, and greater aptitude to respond to change.

Application software: A machine-specific software project file running on one or more controllers, usually including an interface to a corresponding Human Machine Interface (HMI) and machine-specific hardware-control.

alias: Application project, machine project, machine-specific project

Call: A language construct causing the execution of a function, function block, or method.

Class: A Program Organization Unit consisting of

- the definition of a data structure,
- a set of methods (like subroutines) to be performed upon the data structure.

A class is an implementation — a concrete data structure and collection of subroutines — while a type is an interface.

Cohesion: The degree of relatedness or interdependence within one component (modules, classes, functions) in a software system.

Configuration: A language element corresponding to a programmable controller system (cf. Figure 4).

Continuous Integration / Continuous Delivery / Continuous Deployment

- **CI – Continuous Integration:** The practice of automating the integration of code changes from multiple contributors into a single software project.

- **CD – Continuous Delivery or Deployment:**
  - *Continuous delivery*: A developer's changes to an application are automatically bug tested and uploaded to a repository where they can then be deployed to a live production environment.
  - *Continuous deployment* (the other possible "CD"): Automatically releasing a developer's changes from the repository to production, where it is usable by customers or stakeholders.

Coupling: The degree of interdependence between different modules, components, or classes in a software system. Measures the extent to which changes in one module require modifications in other modules.

Declaration: A mechanism for establishing the definition of a language element.

Global variables: A variable whose scope is global: It can be used in each POU of the project.

IEC 61131-3 Software model: The basic high-level language elements and their interrelationships as illustrated in Figure 4. These elements are programmed using the languages defined in this standard, i.e., programs and function block types, classes, functions, and configuration elements, namely, configurations, resources, tasks, global variables, access paths, and instance specific initializations, which support the installation of programmable controller programs into programmable controller systems.

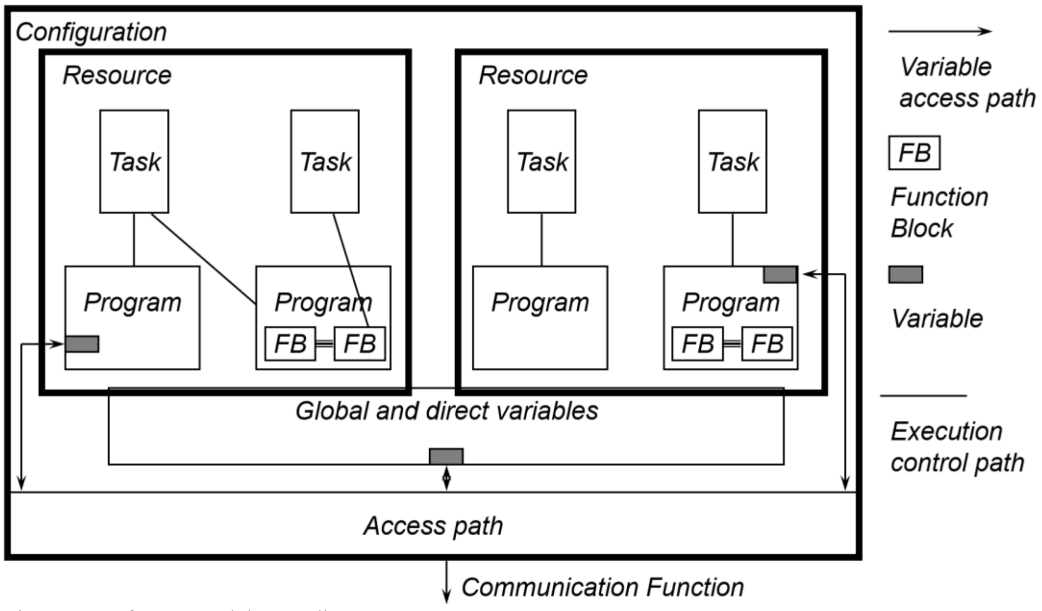


Figure 4. Software model according to IEC 61131-3 [13].

Information flow: The data exchange between POUs. Can be either direct data exchange via calls or indirectly via reading / writing global variables.
alias: data exchange; data flow

Inheritance: The creation of a new class, function block type or interface based on an existing class, function block type or interface, respectively.

Input variable: A variable which is used to supply a value to a program organization unit except for classes.

Instance: An individual, named copy of the data structure associated with a function block type, class, or program type, which keeps its values from one call of the associated operations to the next.

Instantiation: The creation of an instance.

Interface: A language element in the context of object-oriented programming containing a set of method prototypes.
Example: It resembles a motor flange in that it delineates the diameter, distance, and shaft size, albeit it is not a motor.

Maintenance activity: An activity to fix issues (corrective maintenance), to implement new features (perfective maintenance), to make improvements without changing observable behavior (preventative maintenance), or to modify the software to adapt to ever-changing environments (adaptive maintenance).

Method: A language element similar to a function that can only be defined in the scope of a function block or class type and with implicit access to instance variables of the function block instance or class instance.
Example: A boiler can possess a Fill method and a HeatUp method, each of which performs a distinct task.

Output variable: A variable which is used to return a value from the program organization unit except for classes.

PLC Project: The application software together with all referenced libraries (cf. Fig. 4).
alias: PLC program (to be distinguished from POU type program)

POU: A Program Organization Unit. Source code of a PLC program is written via POUs. POU consists of function, function block, class, or program (terms of IEC 61131-3).

Resource: A language element corresponding to a "signal processing function" and its "man-machine interface" and "sensor and actuator interface functions", if any (cf. Figure 4).

Software library: A collection of standardized, reusable code that often targets a coherent problem or functionality, such as:

- POUs, like function blocks or functions
- Interfaces and their methods and properties
- Data types such as enumerations, structures, aliases, unions
- Global variable, constants, parameter lists
- Text lists, image pools, visualizations, visualization elements
- External files (e.g., documentation)

Integration of a library into an application project enables the library modules to be used in a project in the same way other Function Blocks and variables are defined directly in the project.

Software Template: An editable, pre-designed software project file consisting of generic modules which can be adapted efficiently and quickly by software developers based on machine/system application requirements. A template can usually be run on its own and often serves as a starting point to develop machine-specific application projects.

Software Framework: A collection of interfaces and conditions that guide the developer in the implementation of software (in application projects or libraries). It usually represents a collection of tools, but it can also include libraries, example code, or non-executable support, such as programming guidelines. The guideline presented in this document is thus intended to be integrated into a given framework.

Task: An execution control element that provides for periodic or triggered execution of a group of associated program organization units (cf. Fig. 4).

Unit Test: The testing of individual routines and modules by the developer or an independent tester (automated or manually). A test of individual programs or modules to ensure that there are no errors (logical or programming).

Waterfall procedure model: A linear, non-iterative procedure model for software development organized in successive project phases. Results from a preceding project phase serve as binding specifications for the subsequent project phase.

4. Software Development Workflow

4.1. Typical workflow for PLC software development

In the following figure, an exemplary PLC software development workflow is introduced to illustrate different scenarios in industrial development where metrics can be beneficial by supporting the subjective experience of PLC software developers by quantitative indicators for the software quality attributes focused on in this guideline (cf. Section 3.1).

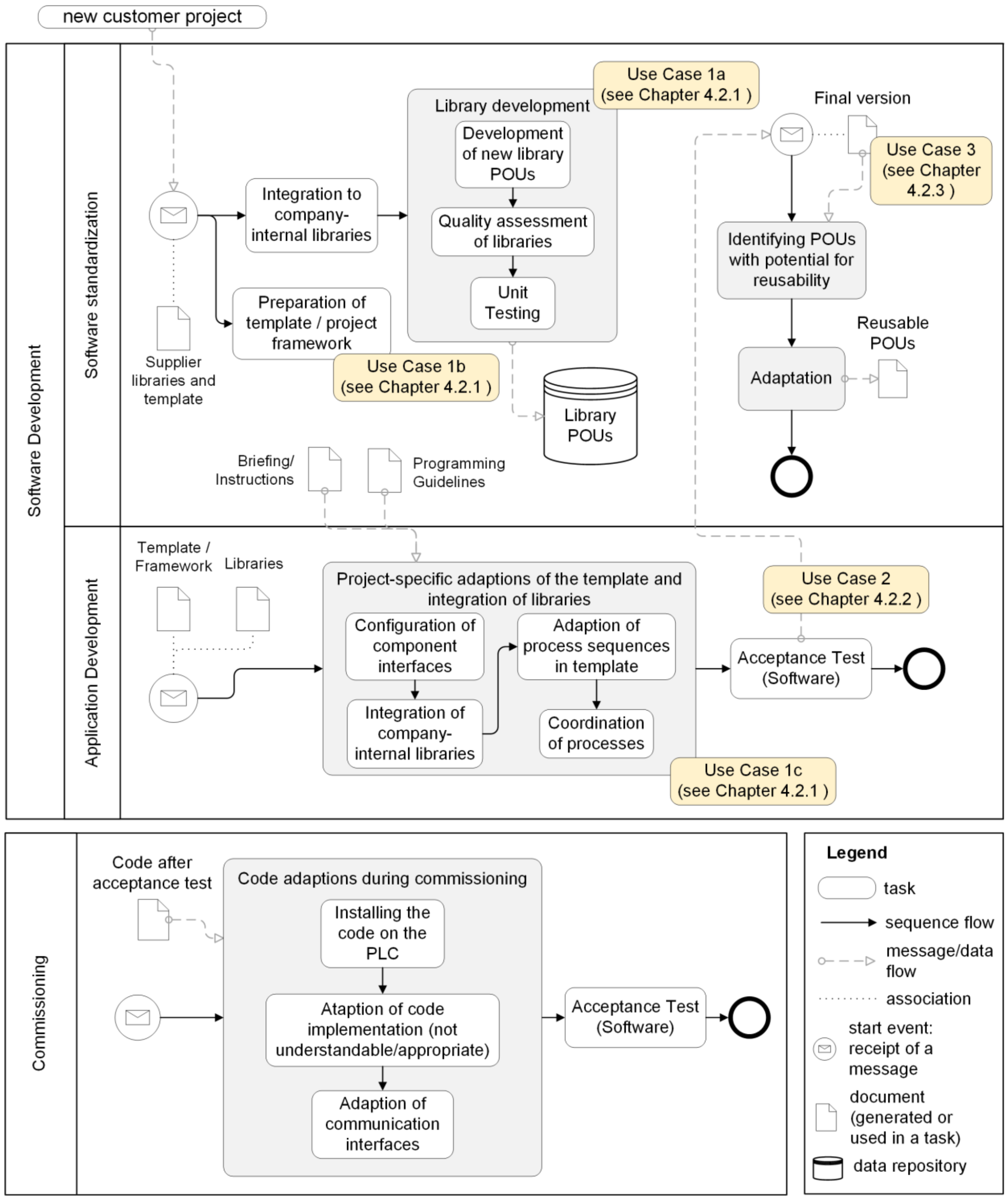

new customer project
Software Development
Software standardization
Supplier libraries and template
Integration to company-internal libraries
Preparation of template / project framework
Library development
Development of new library POUs
Quality assessment of libraries
Unit Testing
Use Case 1a (see Chapter 4.2.1)
Use Case 1b (see Chapter 4.2.1)
Library POUs
Briefing/ Instructions
Programming Guidelines
Final version
Use Case 3 (see Chapter 4.2.3)
Identifying POUs with potential for reusability
Reusable POUs
Adaptation
Application Development
Template / Framework
Libraries
Project-specific adaptions of the template and integration of libraries
Configuration of component interfaces
Adaption of process sequences in template
Integration of company-internal libraries
Coordination of processes
Use Case 2 (see Chapter 4.2.2)
Acceptance Test (Software)
Use Case 1c (see Chapter 4.2.1)
Commissioning
Code after acceptance test
Code adaptions during commissioning
Installing the code on the PLC
Ataption of code implementation (not understandable/appropriate)
Adaption of communication interfaces
Acceptance Test (Software)
Legend
task
sequence flow
message/data flow
association
start event: receipt of a message
document (generated or used in a task)
data repository

Figure 5. Overview of a typical software development workflow in industrial practice, which was observed in a similar form in various companies in the machine and plant engineering sector.

The workflow comprises of two main parts: The *software development* itself (including the software standardization and application development) and the *commissioning*. It is important to note that the introduced workflow example shall only support the exemplary illustration of the benefits when using metrics in realistic scenarios. It does not claim to cover all aspects of software development that can be observed in industrial practice. Even in the case that software development of a specific company deviates from the proposed basic workflow (e.g., because no libraries are used), metrics can be useful in scenarios that are *similar* to the ones described in this section.

The *software standardization* (upper part of *software development* in Figure 5) includes the standardization of reusable software components, which are commonly organized in the form of software libraries that can be used for different machine-specific application projects. Thus, these POUs need to be highly mature to ensure their reliable functionality for different applications and boundary conditions, which is usually ensured by comprehensive unit tests. In series machine manufacturing, companies often use standardized *templates* as part of the provided framework that may comprise interfaces to infrastructural software such as exception handling or HMI linkage. Templates can be used as an orientation to develop the machine-specific software during *application development*.

In the *application development*, the libraries are integrated into machine-specific applications. In case templates are used, they are adapted to the functional specifications of the respective machine to be developed. The software's reliable functionality is tested afterwards via acceptance tests.

In machine and particularly in plant manufacturing, software is often changed during *commissioning*, e.g., in case the system functionality needs to be fixed or adapted on short notice, and partly even during *operation.* Enabling online changes without stopping the process is a highly requested PLC feature by machine and plant manufacturers (approx. 25% according to observation of the tool provider *Schneider Electric Control Engineering – Verification*). Thus, this workflow step is included in the basic workflow to keep these potential changes in mind when applying metric-based quality assessment.

4.2. Use Cases in the Workflow to Apply Software Quality Measurement

Disclaimer on the introduced use cases
When applying this guideline, please be aware of the following points:

- *General recommendations for using software metrics: Before making any changes, please make sure to always keep in mind quality aspects.*
- *Please take the context of your use case into account while adapting them. Depending on involved stakeholders and development procedures, different points / application scenarios in the workflow are suitable for using metrics.*

In the following sections, a set of typical use cases is introduced. These use cases have been observed in different companies from machine and plant manufacturing as examples for real-world scenarios in industrial PLC software development, where a quantification of software quality attributes is considered helpful.

All use cases are described using the categories below, which contain all the relevant information for each scenario:

- ***Stakeholder:*** Roles involved for a particular scenario that can benefit from the application of metrics.

- ***Workflow step:*** Concrete points in an existing workflow where metrics can be applied.
- ***Description:*** Short outline of the use case characteristics.
- ***Possible variations and boundary conditions*****:** Examples for possible specifications / alternatives of the use case as well as assumed boundary conditions.
- ***Relevant quality attributes:*** Quality attributes that are relevant for the use cases in the given scenario and that should be measured with the metrics.
- ***Achievable benefits using metrics:*** Motivations and advantages of using the metrics.

In general, there are three scenarios (listed below) to integrate metric-based measurement of software quality into industrial development practice. Depending on the company-specific boundary conditions, it needs to be decided individually which variant fits best to the respective goals and needs:

- **Continuous quality assessment** across the whole software development cycle
- Quality assessment of existing legacy software **at defined points in the workflow** (e.g., compare the difference between metric values before and after a change of an existing project)
- Development of **"greenfield" software projects** (quality assessment of newly developed software)

4.2.1 Use Case 1: Continuous quality checks of POUs during software development

In an ideal case, quality checks are made continuously during a POU's lifecycle whenever a change is made. While this is a common practice in development using high-level programming languages such as Java or C#, continuous quality checks are not yet standardized or fully utilized in most PLC-based control software development. However, the continuous evaluation of software quality and the early identification of deviations from quality goals is assessed as highly beneficial to avoid time- and cost-intensive refactoring in the long run. In the following, this use case and potential variations are described using the above-mentioned categories.

Stakeholders: Software library developers, application software developers, project managers, clients

Description: After each significant change (e.g., Bug Fix, New Feature etc.), a quality check is performed before the new version (or variant) of the POU to enable a continuous quality monitoring.

Workflow Step: Suitable in all workflow steps in which the software is changed across its lifecycle, including, e.g.:

- **1.a)** After changes of software library POUs (e.g., fixing of bugs identified during unit testing, development of new functionalities)
- **1.b)** Preparation of templates and project frameworks (e.g., adaptions based on feedback from application developers, fixing of errors identified after unit / software acceptance tests)
- **1.c)** After project-specific adaptions of templates or project frameworks

Possible Variations and boundary Conditions:

- *CI/CD with version control system:* Automatic continuous quality checks after each change before merging into the "master" (in case of GIT-based version control); If check fails, the reported issues must be fixed or reviewed.
- *Agile software development approach:* Running metrics, observe and compare the results with the beginning of sprint. Efficiency can be increased in case of automated metric comparison. As part of an agile methodology, a quality check can be part of Definition-of-Done.
- *Waterfall software development:* Integration of regular metric reviews into the workflow before commissioning.

Relevant Quality Attributes: Maintainability, Reusability, Testability, Efficiency, Reliability

Achievable Benefits Using Metrics: Deviations from desired quality targets are detected at an early stage and can be compensated in time. This avoids long-term costs due to susceptibility to errors and costly maintenance; avoidance of costly refactoring of historically grown software at later stages and improvement of software optimization "on-the-fly" right during development.

4.2.2 Use Case 2: Comparison of code before and after commissioning

During on-site system commissioning in plant manufacturing, the software is often subject to continuous changes. Due to high time pressure, the software is changed on a short notice usually by technicians without comprehensive programming skills. Software metrics may support the identification of such changes after the system is commissioned to identify software parts that are, e.g., frequently affected by changes during start-up. This may indicate, e.g., that the changed POUs are difficult to understand under time pressure by non-software experts and, thus, should be refactored.

Stakeholders: Start-up personnel on-site; software application developers; management

Description: After commissioning, variations in quality attributes due to short-term software changes in the field are evaluated to identify source code that is frequently modified during commissioning.

Workflow Step: After acceptance test

Possible Variations / Boundary Conditions:

- Start-up personnel has access to the whole source code: Evaluation of the whole PLC software project
- Start-up personnel has only access to specific parts of the source code: Targeted evaluation of accessible software parts

Relevant Quality Attributes: Maintainability, Reuse, Efficiency, Reliability

Achievable Benefits Using Metrics: Metrics can be used to identify software parts that are particularly affected by major changes during commissioning. This may hint at software parts that are difficult to understand for start-up personnel or may cause issues during start-up, thus need to be revised.

Metrics may support the automatic identification of software parts that are usually “not touched” during commissioning and, thus, can be “protected” to avoid unintended changes.

4.2.3 Use Case 3: Plant / Machine audit after project has been finished

In ever-changing and rapidly advancing industrial automation environment, the importance of high-quality software continues to grow. This is why companies are more and more interested in leveraging optimization potential of software in a targeted and efficient manner. One approach highlighting this is plant or machine audits after completion of a project.

Metrics can be used to give an objective overview of essential quality attributes of the software, without requiring in-depth knowledge of the source code. This use case is, therefore, particularly interesting from a management perspective, where decisions have to be made based on a cost-benefit ratio.

Stakeholders: Management, quality team (if applicable).

Description: Selected metrics of all PLC programs are calculated to classify the programs by their size and complexity and to detect potential technical debt to organize the priorities of future development. This use case is not planned by the developers themselves but more by the management to help make the right decisions for their business operations.

Workflow Step: After acceptance test or when a project is finished.

Possible Variations / Boundary Conditions:

- Plant manufacturing: Comparison of metric values of different PLCs within one plant to compare different machines / plant parts within the system
- Machine manufacturing (but also applicable to plant manufacturing): Comparison of metric values across different machine types

Relevant Quality Attributes:, Maintainability, Reusability, Testability, Efficiency, Reliability

Achievable Benefits using Metrics: Decisions on the prioritization of refactoring measures can be precisely quantified and thus discussed using data. The success of quality optimizations can be quantitatively measured across projects. (Recurring) quality deficits can be automatically identified across projects resulting in the identification of "hidden" quality issues and optimization potentials.

4.2.4 Further Use Cases supported by metrics

- Quality checks of third-party code.
- For vendor selection: Application of quality guideline could be incorporated in the contract.
- Identification of software functions that would highly benefit in terms of improvement from refactoring, in an already existing code base.
- Quality checks during acceptance tests.

5. Recommendations on How to Use Metrics in Industrial PLC Software Development

In computer science, there are several tools (e.g., SonarQube by Sonar or Sigrid by Software Improvement Group), recommendations and guidelines (e.g., Common Weakness Enumeration) on how to improve software. For the improvement of industrial PLC software, tools and guidelines are available from platform providers such as Schneider Electric and the CODESYS Group but also from platform-independent organizations such as PLCopen. All these tools, guidelines and recommendations have in common that they are formulated based on certain categories to support software developers in the correct application and interpretation of analysis results.

Derived from the consensus of these well-accepted categories used in established tools and guidelines, the following categories are proposed to structure the recommendations in a uniform way that is understandable for the concerned stakeholders.

- ***Non-compliant code example***: Negative example of source code that does not fulfill the quality attribute and an illustration how this can be made explicit in selected software metrics.
- ***Compliant code example:*** Positive example of source code that does fulfill the quality attribute and an illustration how this can be made explicit in selected software metrics
    - ***Note:*** *The selected code excerpts for compliant and non-compliant code examples are extracted from an idealized, didactic example of a water heater and do not necessarily represent the extent of a complete industrial application. The analyses have been exemplarily conducted using the CODESYS Static Analysis.*
- ***Implementation specification:*** Examples for metrics that are available in industrial PLC software platforms and analysis tools that can be used to measure a specific influencing factor. In the proposed recommendations, the implementation specifications refer to the tools below. In future versions of the guideline, additional code analysis tools providing software metrics shall be included.
    - CODESYS – Static Analysis *(referred to as CODESYS Static Analysis)*
    - Schneider Electric – EcoStruxure – Machine Advisor Code Analysis *(referred to as SE – EcoStruxure MACA)*
    - Schneider Electric – EcoStruxure Control Engineering – Verification *(referred to as SE - EcoStruxure CE-V)*
    - Software Improvement Group – Sigrid *(referred to as SIG Sigrid Maintainability and Architecture Quality - AQ)*
- ***Potential mitigations:*** Actions that can be performed or aspects that should be considered to enhance the fulfillment of the respective quality attribute by adjusting the measured influencing factors.

Since the focus of this PLCopen guideline is on applying metric-based quality assessment of industrial PLC software, the following categories are additionally added to specify the usage of the recommendations in the following sections:

- ***Use cases and corresponding workflow steps:*** Corresponding use case, for which the measurement of the specific quality attribute is helpful (cf. Section 4)
- ***Potential antagonist:*** Quality attribute that may be affected by a target conflict when optimizing another quality attribute.
- ***Cross-References to other PLCopen guidelines***: Hint to other PLCopen guidelines that can be referred to for deeper insights on selected aspects.

The recommendations in the following are structured according to the focused software quality characteristics, i.e., *maintainability* (cf. Section 5.1), *reusability* (cf. Section 5.2), *testability* (cf. Section 5.3), *efficiency* (cf. Section 5.4), and *reliability* (cf. Section 5.5).
To select the metrics that are suitable for quantifying the respective quality attributes, two workshops with international experts from industrial PLC software development have been conducted. The experts covered the perspectives of platform suppliers *(metric providers)* and machine manufacturers *(metric users).* In the workshop, available metrics from industrial code analysis tools and scientific literature have been mapped to the quality attributes (details can be found in Appendix 1).

The metric categories used in the recommendations are the following, sorted by the degree of required experience in metric-based quality assessment as prerequisite for correctly using and interpreting the respective metric (cf. Section 3.2):

Metrics for Step 1 (basic level in applying software metrics):

- Code Size
- Language-specific Size Metrics
- Software Complexity
- Code Documentation

Metrics for Step 2 (advanced level in applying software metrics):

- Variables and POU Interfaces
- Information Exchange
- Reuse Indicators

Metrics for Step 3 (expert level in applying software metrics):

- Metrics on the usage of the object-oriented extension of the IEC 61131-3: OO-IEC Elements

For an explanation across the different tools used, please refer to Appendix 1. In the following, recommendations for the measurement of the five focused quality attributes are formulated based on the expert workshops.

5.1. Recommendation: Metric-based Assessment of Maintainability

Use cases and corresponding workflow steps (cf. Section 4.2):

- **Use Case 1**: Continuous quality checks of POUs during software development.
  - *Motivation:* Check whether a specific change affects the software's maintainability and overview how the software's maintainability develops over time.
- **Use Case 2**: Comparison of code before and after commissioning.
  - *Motivation:* Check whether adaptions during the start-up phase had an impact on the software's maintainability and a revision of the changed parts is required.
- **Use Case 3**: Plant or machine audit after a project has been finished.
  - *Motivation:* Assess the software project's maintainability in relation to other projects to identify strengths, weaknesses, and optimization potentials in the given design decisions.

Description of the required metrics:

Metric category: Code Size
POU size
↑ ***High value:*** 👎

↓ ***Low value:*** 👍

Reason: *Outstandingly long implementations can be difficult to understand at first glance, thus hampering the implementation of maintenance tasks.*

Implementation specification using available metrics:
- *SE - EcoStruxure MACA: Lines Of Code*
- *SE - EcoStruxure CE-V: Number of instructions*
- *CODESYS Static Analysis: NOS - Number Of Statements*
- *SIG Sigrid Maintainability: Unit Size*

Number of actions

↑ ***High value:*** 👍
↓ ***Low value:*** 👎

Reason: *Encapsulation of code in smaller units supports maintenance work on the software.*

Implementation specification using available metrics:
- *SE - EcoStruxure MACA: Number Of Actions*
- *SE - EcoStruxure CE-V: plcobjecttype counter*
- *SIG Sigrid Maintainability: Unit Size*

🛇 ***Potential antagonist:***
↓ ***Efficiency*** *in case of increased number of calls.*

Potential mitigations to enhance maintainability based on code size:
- Make sure to apply single responsibility principle to reduce POU size by re-distributing any non-relevant or sub-functionality to other POUs or sub-elements such as methods or actions.
- Structure functionality by encapsulating coherent code parts into actions (or methods) that can be called by the POU.
- Avoid local code duplication and use Derived FB, Arrays, or Structures.

Metric category: Language-specific Size Metrics

Number of steps, transitions, branches in SFC

↑ ***High value:*** 👎
↓ ***Low value:*** 👍

Reason: *Creating very large SFCs should be avoided. Especially a high number of branches leading to a high width of SFCs may hamper the traceability of change effects with the POU.*

Implementation specification using available metrics:
- *SE - EcoStruxure MACA: Number Of Transitions*
- *SE - EcoStruxure CE-V: nbofbranches, g7height, g7width*
- *CODESYS Static Analysis: Number of SFC branches, Number of SFC steps*

Number of networks in FBD

↑ ***High value:*** 👎
↓ ***Low value:*** 👍

Reason: *A high number of networks in a POU programmed in FBD may hinder the developer from understanding the POU functionality at first glance, thus hamper the implementation of maintenance tasks. The effect is amplified in case individual networks are particularly complex.*

Implementation specification using available metrics:

- *SE – EcoStruxure MACA: Number Of FBD Networks, Halstead Complexity for FBD*

Potential mitigations to enhance maintainability based on language-specific parameters:

- In case a SFC/FBD is not readable with one look (requiring zooming out or scrolling), it should be considered whether its functionality should be distributed across several POUs.
- In case individual FBD networks stand out with high complexity values, try to distribute their functionality across multiple networks.
- Use FBD to develop interconnected modules rather than simple POUs which leads to multiple networks. Interconnected modules have fewer networks.

Metric category: Variables and POU Interfaces

Usage of global variables

↑ ***High value***: 👎
↓ ***Low value***: 👍

Reason: *Indirect data exchange via global variables can be difficult to trace and may cause undesired dependencies.*

Implementation specification using available metrics:

- *SE - EcoStruxure MACA: Number of GVL Usages*
- *SE - EcoStruxure CE-V: extvarref*
- *CODESYS Static Analysis: Used different global variables*

POU interfaces

↑ ***High value***: 👎
↓ ***Low value***: 👍

Reason: *A high number of dependencies to its environments may cause undesired cross-effects in case the POU is modified.*

Implementation specification using available metrics:

- *SE - EcoStruxure CE-V: inputcount, outputcount, nbofparam*
- *CODESYS Static Analysis: Number of input/output variables*
- *SIG Sigrid Maintainability: Unit Interfacing*

Potential mitigations to enhance maintainability based on variables and POU interfaces:

- Check whether the usage of global variables is inevitably necessary or whether relevant information could also be received directly by POUs via direct data exchange.
- In case global variables cannot be avoided, check whether the global variables are reasonably structured (e.g., functional coherent variables arranged together in STRUCTs).
- Check why POUs with outstandingly large interfaces need this amount of external information. Maybe the functionality can be split-up and distributed across more POUs (or methods / actions).
- Check if different parameters from input or output could be grouped together in a data structure when they are produced and consumed at similar locations and create the associated structure.
- Create function interfaces that pass data through arguments instead of global variables.

Metric category: Code Documentation

Comments in the source code

↑ ***High value***: 👍
↓ ***Low value***: 👎

Reason: *Code documentation in the form of comments supports software developers in understanding its functionality.*

Implementation specification using available metrics:

- *SE - EcoStruxure MACA: Source Code Commented Ratio, Commented Variables Ratio, Number Of Multiline Comments, Number Of Header Comment Lines*
- *SE - EcoStruxure CE-V: percentage of comment, result of verification tool*
- *CODESYS Static Analysis: Percentage of comment*

Potential mitigations to enhance maintainability based on code documentation:

- Code documentation can be enhanced by adding additional comments. The metrics can provide hints whether the implementation part or the variables would benefit the most from additional comments.

Note : A lot of comments doesn't necessarily mean good quality. Writing meaningful comments, and not paraphrasing the code are important. In an extreme case, one could create a POU with just comments/notes and exclude the POU from being built.

Metric category: OO-IEC Elements

Depth of inheritance

↑ ***High value***: 👎
↓ ***Low value***: 👍

Reason: *A high depth of inheritance (EXTENDS) may hamper maintainability and understandability due to traceability issues of dependencies. Especially long chains of inheritance (e.g., depth of inheritance > 10) can be hard to maintain in case errors occur at the bottom and are spread via inheritance.*

Implementation specification using available metrics:

- *SE - EcoStruxure MACA: Extended By, Extends*
- *CODESYS Static Analysis: DIT - Depth of Inheritance Tree, NOC - Number Of Children*

🛇 ***Potential antagonist:***

↓ ***Reusability &Testability*** *benefit from inheriting from existing POU structures and functionalities.*

Cohesion

↑ ***High value***: 👍
↓ ***Low value***: 👎

Reason: *A low cohesion points to the fact that a POU has more than one responsibility, i.e., it tries to fulfill more than one functionality. Understanding and maintaining is more complex because it is required to understand all functionalities that are provided, not just one.*

Implementation specification using available metrics:

- *CODESYS Static Analysis: LCOM - Lack of cohesion in methods (inverted logic as in a high lack of cohesion is disadvantageous)*

- *SIG Sigrid AQ: Component Cohesion*

Coupling of objects to the environment

↑ ***High value:*** 👎
↓ ***Low value:*** 👍

Reason: *High coupling to the environment contradicts modularity and, thus, hamper maintainability.*

Implementation specification using available metrics:
- *CODESYS Static Analysis: RFC - Response For Class, CBO - Coupling Between Objects*
- *SIG Sigrid Maintainability: Module coupling*

Encapsulation of functionality and information in properties and methods

↑ ***High value:*** 👍
↓ ***Low value:*** 👎

Reason: *Encapsulation of code in smaller units (i.e., properties or methods) supports maintenance work on the software.*

Implementation specification using available metrics:
- *SE - EcoStruxure MACA: Number Of Properties, Number Of Methods*

🛇 ***Potential antagonist:***
↓ ***Efficiency*** *in case of (nested) call chains.*

Potential mitigations to enhance maintainability based on OO-IEC elements:
- In case inheritance is used in the project, clarify the relations between POUs (classes) in additional comments in the code if needed.
- Too high coupling between methods or a lack of cohesion can be an indicator that the functionality distribution across methods is not ideal. It should be checked whether the concerned methods can be refactored or whether functionality can be re-distributed.
- Depending on the context, it might be beneficial to use composition instead of inheritance.
- Dependency inversion principle: Higher level modules should not import anything from low level modules. Both should depend on interfaces. This promotes reusability, maintainability, and testability.

Metric category: Software Complexity

Complexity and length of source code

↑ ***High value:*** 👎
↓ ***Low value:*** 👍

Reason: *High complexity can make it difficult to trace potential cross-effects of changes in the implementation part, e.g., in case a change is made to a highly nested loop (textual) or network (graphical). Additional note: A distinction between machine-specific software and used library code is required: For the used (but not changed) library code, high complexity can be tolerable since the user is not directly engaging those POU's internal complexity as long as the interfaces are easy to use.*

Implementation specification using available metrics:
- *SE - EcoStruxure MACA: Halstead Complexity, Cyclomatic Complexity*
- *SE - EcoStruxure CE-V: length, volume, difficulty, vg*

- *CODESYS Static Analysis: Halstead (D/HV/HL), Complexity (McCabe)*
- *SIG Sigrid Maintainability: Unit size, unit complexity (McCabe)*

Potential mitigations to enhance maintainability based on software complexity:

- Check whether the same functionality can be implemented with better understandability using less loops / nested networks.
- Check whether the number of used operators can be reduced, e.g., by refactoring too large POUs and re-distributing functionality.

Metric category: Information Exchange

Information flow from POU to its environment

↑ ***High value*:** 👎
↓ ***Low value*:** 👍

Reason: *A lot of information from the POU is required by its environment, thus increasing the risk of cross-effects in case of changes.*

Implementation specification using available metrics:
- *SE - EcoStruxure MACA: Fan Out*

Potential mitigations to enhance maintainability based on information exchange:

- Check whether some of the outgoing information flows from a POU can be reduced, e.g., by splitting-up a POUs functionality and re-distributing its functionality to further POUs, methods, or actions.
- To reduce high Fan Out, create an intermediate POU to factor out modules with low coupling and high cohesion.
- Simplify data structures.

Metric category: Reuse Indicators

Call depth

↑ ***High value*:** 👎
↓ ***Low value*:** 👍

Reason: *Changes to elements on lower parts of a long call chain may have cross-effects on POUs using their functionality. The higher the call depth, the higher the risk of such cross-effects.*

Implementation specification using available metrics:
- *SE - EcoStruxure CE-V: calldepthmin, calldepthmax*

Duplication

↑ ***High value*:** 👎
↓ ***Low value*:** 👍

Reason: *When code is copy-pasted, the maintenance effort for fixing bugs or making changes increases due to increase in the code amount and the need of managing separate copies in multiple places for needed changes.*

Implementation specification using available metrics:
- *CODESYS Static Analysis: Duplication ratio*
- *SIG Sigrid Maintainability: Duplication*

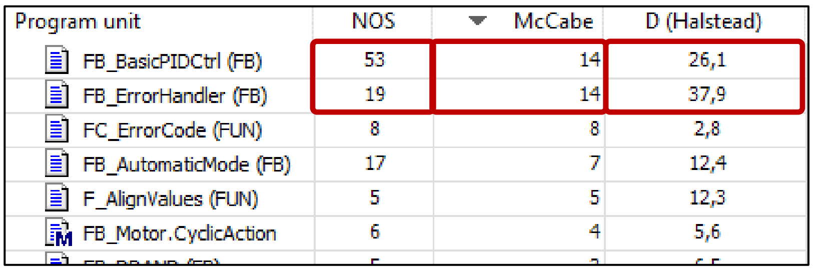
⊘ ***Potential antagonist:***
↓ *An increase in dependencies where code is reused via referring to it (such as function calls or OO mechanisms)*

Potential mitigations to enhance maintainability based on reuse indicators:

- In case a software project (or parts of it) is characterized by high call depths, it should be considered whether the call depth can be reduced by merging functionalities from two or more POUs into one.
- Use systematic reusable mechanisms such as defining functions or facilitating OO mechanisms, such as inheritance.

Disclaimer: The subsequent code excerpts are extracted from an idealized, didactic example of a water heater and do not necessarily represent the extent of a complete industrial application. The provided code snippets are independent from each other, i.e., the compliant code example is not an optimization of the non-compliant example. Due to the nature of the supplied program, the analyses have been conducted through the utilization of CODESYS Static Analysis.

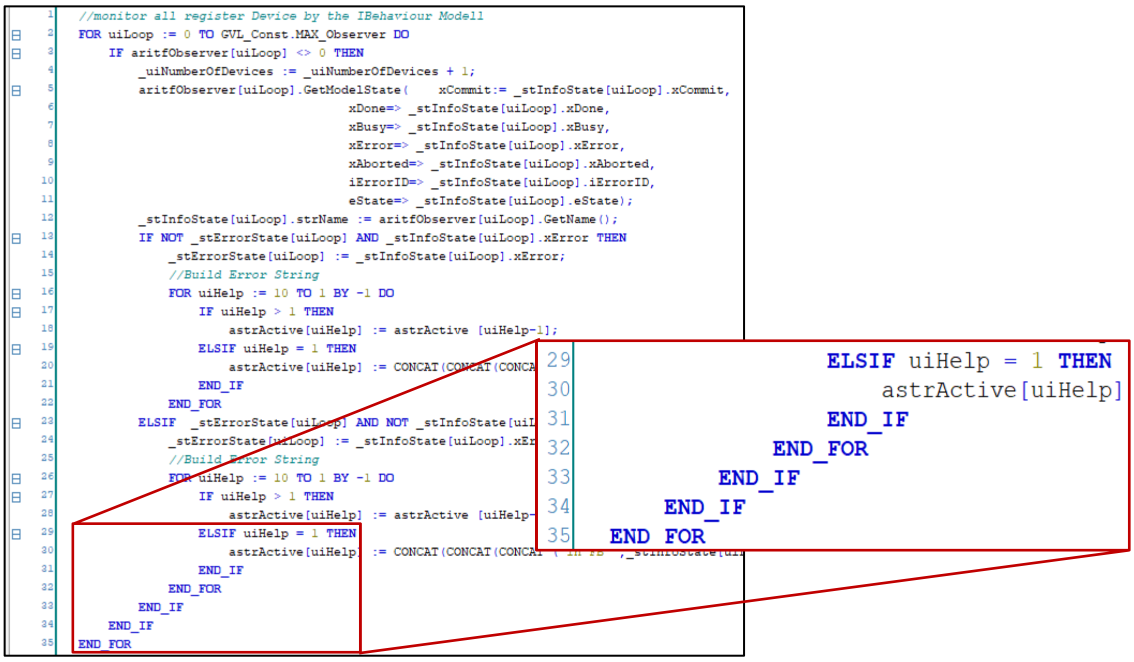
Noncompliant code example: In the metric categories of "Code Size" and "Software Complexity", two outliers can be spotted, with by far the highest Halstead and Cyclomatic Complexity (McCabe) as well as one of the highest Number-Of-Statements, encouraging inquiry into possible code optimization for enhanced maintainability (cf. Figure 6).



Program unit	NOS	McCabe	D (Halstead)
FB_BasicPIDCtrl (FB)	53	14	26,1
FB_ErrorHandler (FB)	19	14	37,9
FC_ErrorCode (FUN)	8	8	2,8
FB_AutomaticMode (FB)	17	7	12,4
F_AlignValues (FUN)	5	5	12,3
FB_Motor.CyclicAction	6	4	5,6

Figure 6. Metrics table from CODESYS sorted by highest McCabe Cyclomatic Complexity.

An analysis of the POU implementation reveals the reason for the high Cyclomatic Complexity: deeply nested FOR-loops and IF-statements which hamper maintainability as elaborated in the above "Description of the required metrics".

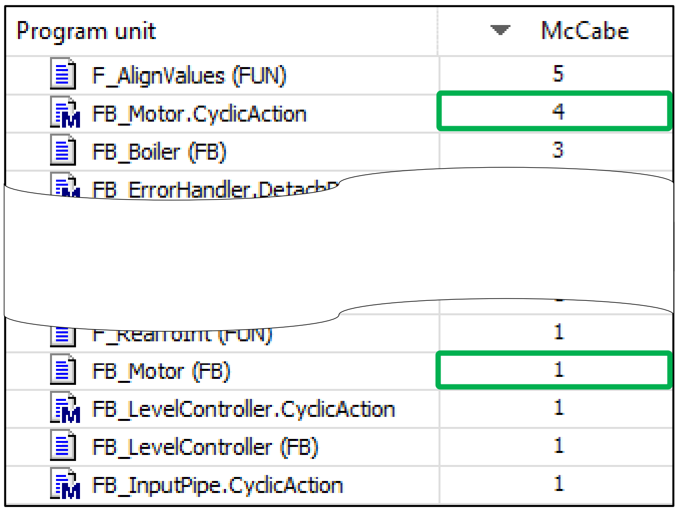


```
//monitor all register Device by the IBehaviour Modell
FOR uiLoop := 0 TO GVL_Const.MAX_Observer DO
    IF aritfObserver[uiLoop] <> 0 THEN
        _uiNumberOfDevices := _uiNumberOfDevices + 1;
        aritfObserver[uiLoop].GetModelState(    xCommit:= _stInfoState[uiLoop].xCommit,
                                        xDone=> _stInfoState[uiLoop].xDone,
                                        xBusy=> _stInfoState[uiLoop].xBusy,
                                        xError=> _stInfoState[uiLoop].xError,
                                        xAborted=> _stInfoState[uiLoop].xAborted,
                                        iErrorID=> _stInfoState[uiLoop].iErrorID,
                                        eState=> _stInfoState[uiLoop].eState);
        _stInfoState[uiLoop].strName := aritfObserver[uiLoop].GetName();
        IF NOT _stErrorState[uiLoop] AND _stInfoState[uiLoop].xError THEN
            _stErrorState[uiLoop] := _stInfoState[uiLoop].xError;
            //Build Error String
            FOR uiHelp := 10 TO 1 BY -1 DO
                IF uiHelp > 1 THEN
                    astrActive[uiHelp] := astrActive [uiHelp-1];
                ELSIF uiHelp = 1 THEN
                    astrActive[uiHelp] := CONCAT(CONCAT(CONCA
                END_IF
            END_FOR
        ELSIF  _stErrorState[uiLoop] AND NOT _stInfoState[uiL
            _stErrorState[uiLoop] := _stInfoState[uiLoop].xEr
            //Build Error String
            FOR uiHelp := 10 TO 1 BY -1 DO
                IF uiHelp > 1 THEN
                    astrActive[uiHelp] := astrActive [uiHelp-
                ELSIF uiHelp = 1 THEN
                    astrActive[uiHelp] := CONCAT(CONCAT(CONCA
                END_IF
            END_FOR
        END_IF
    END_IF
END_FOR
```

```
29                ELSIF uiHelp = 1 THEN
30                    astrActive[uiHelp]
31                END_IF
32            END_FOR
33        END_IF
34    END_IF
35 END_FOR
```

Figure 7. Code snippet highlighting the nested IF-Statements

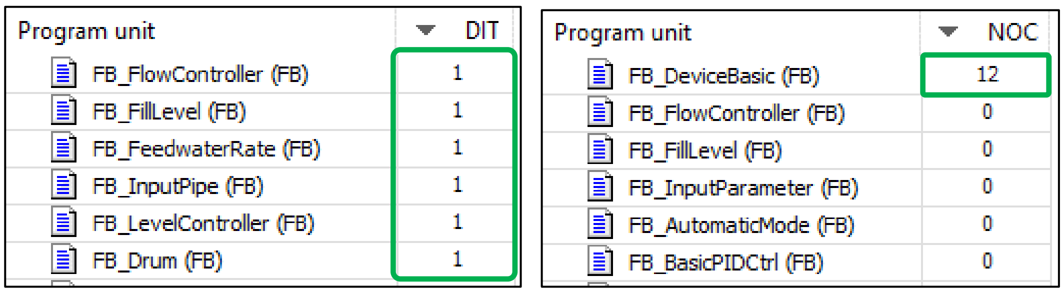
Compliant code example: By applying the suggested mitigation to enhance maintainability by re-distributing functionality, in this case by using methods, McCabes Cyclomatic Complexity can be split up into POUs with smaller values.



Program unit	McCabe
F_AlignValues (FUN)	5
FB_Motor.CyclicAction	4
FB_Boiler (FB)	3
FB_ErrorHandler.Detach	
F_RealPoint (FUN)	1
FB_Motor (FB)	1
FB_LevelController.CyclicAction	1
FB_LevelController (FB)	1
FB_InputPipe.CyclicAction	1

Figure 8. Excerpt from metrics table showing lowered cyclomatic complexity of optimized POUs

Looking at the metric category "OO-IEC Elements" shows that the enhanced maintainability is apparent in the form of a low Depth-Of-Inheritance (DIT) and a manageable amount of Number-of-Children (NOC).



Program unit	DIT
FB_FlowController (FB)	1
FB_FillLevel (FB)	1
FB_FeedwaterRate (FB)	1
FB_InputPipe (FB)	1
FB_LevelController (FB)	1
FB_Drum (FB)	1

Program unit	NOC
FB_DeviceBasic (FB)	12
FB_FlowController (FB)	0
FB_FillLevel (FB)	0
FB_InputParameter (FB)	0
FB_AutomaticMode (FB)	0
FB_BasicPIDCtrl (FB)	0

Figure 9. Metrics tables from CODESYS sorted by Depth of Inheritance Tree (DIT) and Number Of Children (NOC)

Cross-reference to other PLCopen guidelines:

PLCopen_OOP_Guidelines V10.pdf of November 18, 2021

5.2. Recommendation: Metric-based Assessment of Reusability

Use cases and corresponding workflow steps (cf. Section 4.2):

- **Use Case 1**: Continuous quality checks of POUs during software development.
  - *Motivation*: Check whether a specific change affects the software's reusability and overview how the software's reusability develops over time.
- **Use Case 2**: Comparison of code before and after commissioning.
  - *Motivation*: Check whether adaptions during the start-up phase had an impact on the software's reusability and a revision of the changed parts is required. Changes during commissioning via Copy, Paste & Modify could be compensated by standardizing reusable functionality.
- **Use Case 3**: Plant or machine audit after a project has been finished.
  - *Motivation:* Assess the amount of reuse of a project's POUs in relation to other projects to identify strengths, weaknesses and optimization potentials in the given design decisions regarding, e.g., efficiency of development processes.

Description of the required metrics:

Metric category: Code Size
POU size ↑ ***High value:*** 👎 ↓ ***Low value:*** 👍 ***Reason:*** *POUs of small size (small granularity) are usually more flexible regarding the field of application because they usually fulfill a specific, manageable functionality (e.g., reading a sensor value) and, thus, can be reused for many applications. On the other hand, very large POUs can be an indicator for very comprehensive functionality tailored to a specific machine, thus hampering its reusability.* ***Implementation specification using available metrics:*** - *SE - EcoStruxure MACA: Lines Of Code* - *SE - EcoStruxure CE-V: Number of instructions* - *CODESYS Static Analysis: NOS - Number Of Statements* - *SIG Sigrid Maintainability: Unit size*
Potential mitigations to enhance reusability based on code size:

- Make sure to apply single responsibility principle to reduce POU size by re-distributing any non-relevant or sub-functionality to other POUs or sub-elements such as methods or actions.
- Structure functionality by encapsulating coherent code parts into actions (or methods) that can be called by the POU.
- Avoid local code duplication and use Derived FB, Arrays, or Structures.

Metric category: Language-specific Size Metrics

Number of transitions in SFC

↑ ***High value***: 👎
↓ ***Low value***: 👍

Reason: *cf. Size Metrics*

Implementation specification using available metrics:
- *SE - EcoStruxure MACA: Number Of Transitions*
- *SE - EcoStruxure CE-V: nbofbranches, g7height, g7width*
- *CODESYS Static Analysis: Number of SFC branches, Number of SFC steps*

Number of networks in FBD

↑ ***High value***: 👎
↓ ***Low value***: 👍

Reason: *cf. Size Metrics*

Implementation specification using available metrics:
- *SE - EcoStruxure MACA: Number Of FBD Networks, Halstead Complexity for FBD*

Potential mitigations to enhance reusability based on language-specific parameters:
- In case an SFC/FBD is not readable with one look (requiring zooming out or scrolling), it should be considered whether its functionality should be distributed across several POUs.
- In case individual FBD networks stand out with high complexity values, try to distribute their functionality across multiple networks.
- Usage of FBD to develop interconnected modules rather than simple POUs which leads to multiple networks. Interconnected modules have fewer networks.

Metric category: Variables and POU Interfaces

Usage of global variables

↑ ***High value***: 👎
↓ ***Low value***: 👍

Reason: *Reading and writing into global variables lead to (undesired) dependencies from global data from other POUs that cannot be provided via calls, thus the creation of well-defined interfaces between POUs as a prerequisite for reuse is hampered.*

Implementation specification using available metrics:
- *SE - EcoStruxure MACA: Number of GVL Usages*
- *SE - EcoStruxure CE-V: extvarref*
- *CODESYS Static Analysis: Used different global variables*

Direct hardware accesses

↑ ***High value***: 👎
↓ ***Low value***: 👍

Reason: *Direct hardware accesses can be an indicator that a POU is tailored to a specific hardware configuration, thus reusability for other applications can be impaired.*

Implementation specification using available metrics:

- *CODESYS Static Analysis: Number of direct address accesses(I/Os)*

Potential mitigations to enhance reusability based on variables and POU interfaces:

- Check whether the usage of global variables is inevitably necessary or whether relevant information could also be received directly by POUs via direct data exchange.
- In case global variables cannot be avoided, check whether the global variables are reasonably structured (e.g., functional coherent variables arranged together in STRUCTs).
- Check why POUs with outstandingly large interfaces need this amount of external information. Maybe the functionality can be split-up and distributed across more POUs (or methods / actions).
- Check if different parameters from input or output could be grouped together in a data structure when they are produced and consumed at similar locations and create the associated structure.
- Create function interfaces that pass data through arguments instead of global variables.

Metric category: OO-IEC Elements

Usage of inheritance

↑ ***High value***: 👍
↓ ***Low value***: -

Reason: *Inheritance and extension of the functionality of an existing POU may indicate that an implemented functionality can be easily enlarged and reused for different applications.*

Implementation specification using available metrics:

- *SE - EcoStruxure MACA: Extended By, Extends*
- *CODESYS Static Analysis: DIT - Depth of Inheritance Tree, NOC - Number Of Children*

🛇 ***Potential antagonist:***

↓ ***Maintainability*** *in case of traceability issues of dependencies*

Cohesion

↑ ***High value***: 👍
↓ ***Low value***: 👎

Reason: *A high cohesion within objects may indicate that the object focuses on one coherent functionality, which simplifies its reusability in different applications.*

Implementation specification using available metrics:

- *CODESYS Static Analysis: LCOM - Lack of cohesion in methods (inverted logic as in a high lack of cohesion is disadvantageous)*
- *SIG Sigrid AQ: Component Cohesion*

Coupling of objects to the environment

↑ ***High value***: 👎

↓ ***Low value***: 👍

Reason: *A high amount of dependencies of a POU to its environment (may indicate dependability of a specific context, and, thus, impaired reusability).*

Implementation specification using available metrics:
- *CODESYS Static Analysis: RFC - Response For Class, CBO - Coupling Between Objects*
- *SIG Sigrid Maintainability: Module coupling*

Implementation of Interfaces

↑ ***High value***: 👍
↓ ***Low value***: -

Reason: *Well-defined interfaces are the prerequisite for reusability. The usage of interfaces as defined in the object-oriented extension of the IEC 61131-3 is considered a valuable lever for adapting functionality by exchanging POUs with the same interface, thus supporting the reuse of a POU's functionality for different applications.*

Implementation specification using available metrics:
- *SE - EcoStruxure MACA: Implemented By, Implements*

🛇 ***Potential antagonist:***
↓ ***Efficiency*** *A negative impact of calls of properties and methods via interfaces on the performance at runtime has been identified in previous investigations [15].*

Encapsulation of functionality and information in properties and methods

↑ ***High value***: 👍
↓ ***Low value***: 👎

Reason: *Encapsulation of code in smaller units such as properties and methods with defined functionality scope supports flexibility and reusability (cf. size metrics for reusability).*

Implementation specification using available metrics:
- *SE - EcoStruxure MACA: Number Of Properties, Number Of Methods*

🛇 ***Potential antagonist:***
↓ ***Efficiency*** *in case of (nested) call chains.*

Potential mitigations to enhance reusability based on OO-IEC elements:
- In case inheritance is used in the project, clarify the relations between POUs (classes) in additional comments in the code if needed.
- Too high coupling between methods or a lack of cohesion can be an indicator that the functionality distribution across methods is not ideal. It should be checked whether the concerned methods can be refactored or whether functionality can be re-distributed.
- A minimum depth of inheritance promotes reusability. Limit to a maximum to avoid getting into complex/unpredictable behavior. Here, a balance is needed.
- Dependency inversion principle: Higher level modules should not import anything from low level modules. Both should depend on interfaces. This promotes reusability, maintainability, and testability.

Metric category: Software Complexity

Complexity and length of source code

↑ ***High value***: 👎

↓ ***Low value:*** 👍

Reason: *A significantly high complexity may indicate a coarse, monolithic granularity of functionality distribution, thus, it may indicate that a POU's functionality covers a larger part of the application and is, therefore, more difficult to reuse for different applications.*

Implementation specification using available metrics:
- *SE - EcoStruxure MACA: Halstead Complexity, Cyclomatic Complexity*
- *SE - EcoStruxure CE-V: length, volume, difficulty, vg*
- *CODESYS Static Analysis: Halstead (D/HV/HL), Complexity (McCabe)*
- *SIG Sigrid Maintainability: Unit size, unit complexity (McCabe)*

Potential mitigations to enhance reusability based on software complexity:
- Check whether the same functionality can be implemented with better understandability using less loops / nested networks.
- Check whether the number of used operators can be reduced, e.g., by refactoring too large POUs and re-distributing functionality.
- Create smaller functions to lower complexity and increase reusability.
- Encapsulation and abstraction help with reusability.

Metric category: Information Exchange

Direct data exchange between POUs via calls

↑ ***High value:*** 👍
↓ ***Low value:*** 👎

Reason: *Direct data exchange via calls combined with low to zero data exchange via global variables often is an indicator for sophisticated POU interfaces supporting reusability.*

Implementation specification using available metrics:
- *SE - EcoStruxure MACA: Call In, Call Out*
- *SE - EcoStruxure CE-V: calledcount, callproc*
- *CODESYS Static Analysis: Number of calls*

🛇 ***Potential antagonist:***
↓ ***Efficiency*** *A high number of calls may impair the software's efficiency [15].*

Information flow from POU to its environment

↑ ***High value:*** 👎
↓ ***Low value:*** 👍

Reason: *A high information flow from a POU to its environment may indicate that a high number of adjacent software elements are dependent on information from it, thus hampering its reusability.*

Implementation specification using available metrics:
- *SE - EcoStruxure MACA: Fan Out*

Potential mitigations to enhance reusability based on information exchange:
- Check whether some of the outgoing information flows from a POU can be reduced, e.g., by splitting-up a POUs functionality and re-distributing its functionality to further POUs, methods, or actions.
- To reduce high Fan Out – create an intermediate POU to factor out modules with low coupling and high cohesion.

- Simplify data structures.

Metric category: Reuse Indicators

Call depth

↑ ***High value:*** 👎
↓ ***Low value:*** 👍

Reason: *High call depths may indicate that the software is not flat and monolithic but hierarchically structured, i.e., functionality is encapsulated to be reused via calls.*

Implementation specification using available metrics:

- *SE - EcoStruxure CE-V: calldepthmin, calldepthmax*

Duplication

↑ ***High value:*** 👎
↓ ***Low value:*** 👍

Reason: *When code is copy-pasted, the maintenance effort for fixing bugs or making changes increases due to increase in the code amount and the need of managing separate copies in multiple places for needed changes.*

Implementation specification using available metrics:

- *CODESYS Static Analysis: Duplication ratio*
- *SIG Sigrid Maintainability: Duplication*

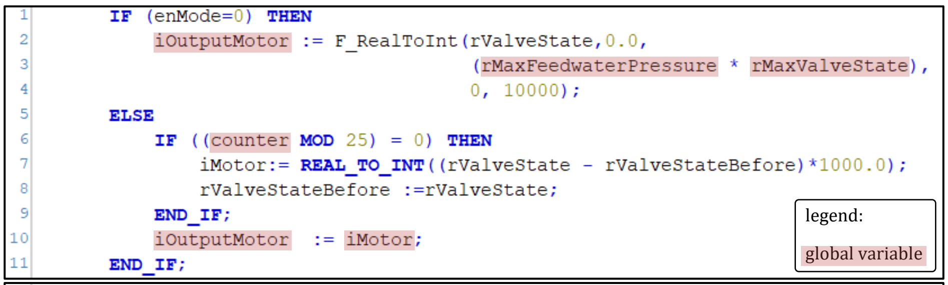
🛇 ***Potential antagonist:***
↓ *An increase in dependencies where code is reused via referring to it (such as function calls or OO mechanisms)*

Potential mitigations to enhance reusability based on reuse indicators:

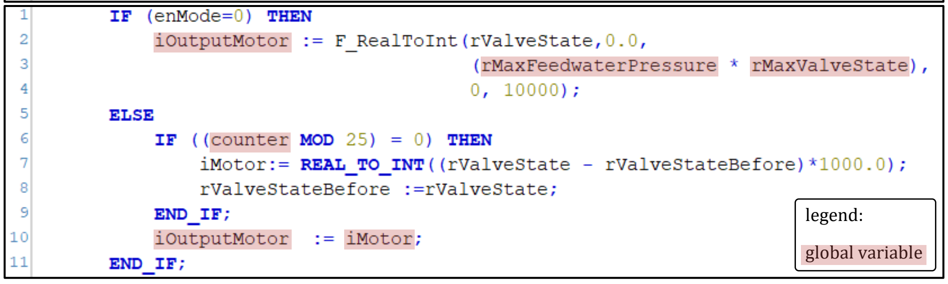
- In case a software project (or parts of it) is characterized by high call depths, it should be considered whether the call depth can be reduced by merging functionalities from two or more POUs into one.
- Use systematic reusable mechanisms such as defining functions or facilitating OO mechanisms, such as inheritance.

Disclaimer: The subsequent code excerpts are extracted from an idealized, didactic example of a water heater and do not necessarily represent the extent of a complete industrial application. The provided code snippets are independent from each other, i.e., the compliant code example is not an optimization of the non-compliant example. Due to the nature of the supplied program, the analyses have been conducted through the utilization of CODESYS Static Analysis.

Noncompliant code example: At first glance it is, neither from this code snippet nor its initialization, apparently visible that the presented method below uses five global variables. This hidden dependency to a global variable list (GVL) can cause issues with reusing "FB_Motor.CyclicAction" by hampering the creation of a well-defined interface to other POUs.

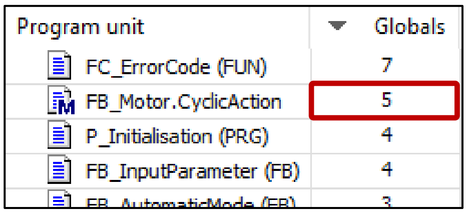


```
IF (enMode=0) THEN
    iOutputMotor := F_RealToInt(rValveState,0.0,
                                (rMaxFeedwaterPressure * rMaxValveState),
                                0, 10000);
ELSE
    IF ((counter MOD 25) = 0) THEN
        iMotor:= REAL_TO_INT((rValveState - rValveStateBefore)*1000.0);
        rValveStateBefore :=rValveState;
    END_IF;
    iOutputMotor  := iMotor;
END_IF;
```



```
IF (enMode=0) THEN
    iOutputMotor := F_RealToInt(rValveState,0.0,
                                (rMaxFeedwaterPressure * rMaxValveState),
                                0, 10000);
ELSE
    IF ((counter MOD 25) = 0) THEN
        iMotor:= REAL_TO_INT((rValveState - rValveStateBefore)*1000.0);
        rValveStateBefore :=rValveState;
    END_IF;
    iOutputMotor  := iMotor;
END_IF;
```

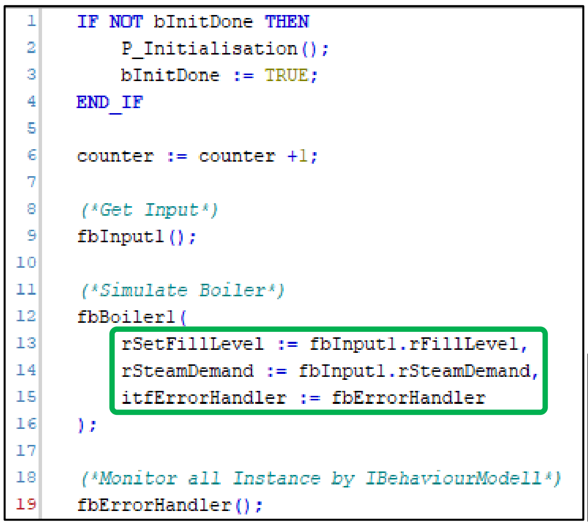
Figure 10. Code snippet with used global variables highlighted in red.



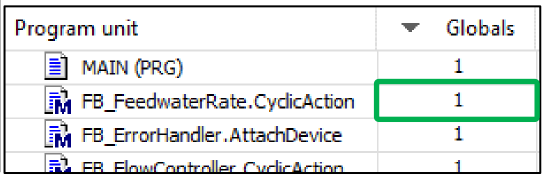
Program unit	Globals
FC_ErrorCode (FUN)	7
FB_Motor.CyclicAction	5
P_Initialisation (PRG)	4
FB_InputParameter (FB)	4
FB_AutomaticMode (FB)	3

Figure 11. Metrics table from CODESYS sorted by highest number of global variables used.

Compliant code example: By directly accessing information through POUs within the call of "fbBoiler1()", instead of via global variables, the dependencies between the POUs are clearly defined and visible. As stated in the reasoning provided in the table above, this well-defined interface between POUs, as observed here, serves as a key requirement to ensure good reusability. It has the potential to greatly reduce oversights and, consequently, errors when reusing this particular POU.



```
IF NOT bInitDone THEN
    P_Initialisation();
    bInitDone := TRUE;
END_IF

counter := counter +1;

(*Get Input*)
fbInput1();

(*Simulate Boiler*)
fbBoiler1(
    rSetFillLevel := fbInput1.rFillLevel,
    rSteamDemand := fbInput1.rSteamDemand,
    itfErrorHandler := fbErrorHandler
);

(*Monitor all Instance by IBehaviourModell*)
fbErrorHandler();
```



Program unit	Globals
MAIN (PRG)	1
FB_FeedwaterRate.CyclicAction	1
FB_ErrorHandler.AttachDevice	1
FB_FlowController.CyclicAction	1

Figure 12. Code snippet with associated metrics table to highlight the absence of global variables.

Cross-reference to other PLCopen guidelines:

PLCopen_OOP_Guidelines V10.pdf

5.3. Recommendation: Metric-based Assessment of Testability

<u>**Use cases and corresponding workflow steps (cf. Section 4.2):**</u>

- **Use Case 1**: Continuous quality checks of POUs during software development.
  - *Motivation:* Check whether a specific change affects the software's testability and overview how the software's testability develops over time.
- **Use Case 3**: Plant or machine audit after a project has been finished.
  - *Motivation:* Assess a software project's testability in relation to other projects to identify strengths, weaknesses and optimization potentials in the given design decisions regarding, e.g., efficiency of testing processes.

<u>**Description of the required metrics:**</u>

Metric category: Code Size

<u>**POU size**</u>

↑ ***High value***: 👎
↓ ***Low value***: 👍

Reason: *Significantly large POUs are often characterized by complex control flows that require expensive testing routines.*

Implementation specification using available metrics:

- *SE - EcoStruxure MACA: Lines Of Code*
- *SE - EcoStruxure CE-V: Number of instructions*
- *CODESYS Static Analysis: NOS - Number Of Statements*
- *SIG Sigrid Maintainability: Unit size*

<u>**Number of actions**</u>

↑ ***High value***: 👍
↓ ***Low value***: 👎

Reason: *A high number of actions may indicate that functionality is encapsulated in smaller parts that are easier to test.*

Implementation specification using available metrics:

- *SE - EcoStruxure MACA: Number Of Actions*
- *SE - EcoStruxure CE-V: plcobjecttype counter*
- *SIG Sigrid Maintainability: Unit size*

🛇 ***Potential antagonist:***
↓ ***Efficiency*** *in case of increased number of calls.*

<u>**Potential mitigations to enhance testability based on code size:**</u>

- Make sure to apply single responsibility principle to reduce POU size by re-distributing any non-relevant or sub-functionality to other POUs or sub-elements such as methods or actions.
- Structure functionality by encapsulating coherent code parts into actions (or methods) that can be called by the POU.
- Avoid local code duplication and use Derived FB, Arrays, or Structures.

Metric category: Language-specific Size Metrics

<u>**Number of transitions in SFC**</u>

↑ ***High value***: 👎
↓ ***Low value***: 👍

Reason: *cf. Size Metrics*

Implementation specification using available metrics:

- *SE - EcoStruxure MACA: Number Of Transitions*
- *SE - EcoStruxure CE-V: nbofbranches, g7height, g7width*
- *CODESYS Static Analysis: Number of SFC branches, Number of SFC steps*

Number of networks in FBD

↑ ***High value***: 👎
↓ ***Low value***: 👍

Reason: *cf. Size Metrics*

Implementation specification using available metrics:

- *SE - EcoStruxure MACA: Number Of FBD Networks, Halstead Complexity for FBD*

Potential mitigations to enhance testability based on language-specific parameters:

- In case a SFC/FBD is not readable with one look (requiring zooming out or scrolling), it should be considered whether its functionality should be distributed across several POUs.
- In case individual FBD networks stand out with high complexity values, try to distribute their functionality across multiple networks.
- Usage of FBD to develop interconnected modules rather than simple POUs which leads to multiple networks. Interconnected modules have fewer networks.

Metric category: Variables and POU Interfaces

Direct hardware accesses

↑ ***High value***: 👎
↓ ***Low value***: 👍

Reason: *In case a POU requires direct hardware access to perform its functionality, i.e., reading from sensors or writing to actuators, this leads to additional dependencies that need to be considered during testing and increase the effort, e.g., to simulate the hardware behavior or in case the POUs are tested with real hardware.*

Implementation specification using available metrics:

- *CODESYS Static Analysis: Number of direct object accesses (I/Os)*

POU interfaces

↑ ***High value***: 👎
↓ ***Low value***: 👍

Reason: *A high number of POU interfaces, in particular a high number of inputs, leads to a high amount of dependencies to a POU's environment, thus testing the correct functionality requires the consideration of a high number of different parameter variations.*

Implementation specification using available metrics:

- *SE - EcoStruxure CE-V: inputcount, outputcount, nbofparam*
- *CODESYS Static Analysis: Number of input/output variables*
- *SIG Sigrid Maintainability: Unit interfacing*

Potential mitigations to enhance testability based on variables and POU interfaces:

- Check why POUs with outstanding large interfaces need this amount of external information. Maybe the functionality can be split-up and distributed across more POUs (or methods / actions).
- Check if different parameters from input or output could be grouped together in a data structure when there are produced and consumed at similar places and create the associated structure.

Metric category: OO-IEC Elements

Usage of inheritance

↑ ***High value***: 👍
↓ ***Low value***: -

Reason: *Using inheritance, functionality of a base class is reused for different applications, thus, testing effort can be reduced since testing the base class behavior also covers the FBs derived from it.*

Implementation specification using available metrics:

- *SE - EcoStruxure MACA: Extended By, Extends*
- *CODESYS Static Analysis: DIT - Depth of Inheritance Tree, NOC - Number Of Children*

🛇 ***Potential antagonist:***
↓ ***Maintainability*** *A high depth of inheritance may hamper maintainability due to traceability issues of dependencies.*

Cohesion

↑ ***High value***: 👍
↓ ***Low value***: 👎

Reason: *A high cohesion may improve testability since it is usually an indicator that an object (e.g., a POU or method) comprises one specific functionality rather than a combination or concatenation of several functionalities. This supports testability since an element with high cohesion often can be tested with a single test case.*

Implementation specification using available metrics:

- *CODESYS Static Analysis: LCOM - Lack of cohesion in methods (inverted logic as in a high lack of cohesion is disadvantageous)*
- *SIG Sigrid AQ: Component Cohesion*

Coupling of objects to the environment

↑ ***High value***: 👎
↓ ***Low value***: 👍

Reason: *In contrast to a high cohesion, a high coupling of an object to its environment leads to a high number of dependencies that need to be considered in the development of test cases (cf. POU interfaces).*

Implementation specification using available metrics:

- *CODESYS Static Analysis: RFC - Response For Class, CBO - Coupling Between Objects*
- *SIG Sigrid Maintainability: Module coupling*

Implementation of Interfaces

↑ ***High value***: 👍
↓ ***Low value***: -

Reason: *If used appropriately, interfaces support the decoupling of functionalities in the software and thus support their testability.*

Implementation specification using available metrics:

- *SE - EcoStruxure MACA: Implemented By, Implements*

⦸ ***Potential antagonist:***

↓ ***Efficiency:*** *A negative impact of calls of properties and methods via interfaces on the performance at runtime has been identified in previous investigations [15].*

Encapsulation of functionality and information in properties and methods

↑ ***High value***: 👍
↓ ***Low value***: 👎

Reason: *Encapsulation of code in smaller units reduces the number of test cases required for an individual object.*

Implementation specification using available metrics:

- *SE - EcoStruxure MACA: Number Of Properties, Number Of Methods*

⦸ ***Potential antagonist:***

↓ ***Efficiency*** *in case of (nested) call chains*

Potential mitigations to enhance testability based on OO-IEC elements:

- In case inheritance is used in the project, clarify the relations between POUs (classes) in additional comments in the code if needed.
- Too high coupling between methods or a lack of cohesion can be an indicator that the functionality distribution across methods is not ideal. It should be checked whether the concerned methods can be refactored or whether functionality can be re-distributed.
- Dependency inversion principle: Higher level modules should not import anything from low level modules. Both should depend on interfaces. This promotes reusability, maintainability, and testability.

Metric category: Software Complexity

Complexity and length of source code

↑ ***High value***: 👎
↓ ***Low value***: 👍

Reason: *High complexity may indicate a high number of (deeply nested) control flows in an object, thus requiring comprehensive test cases to achieve path coverage.*

Implementation specification using available metrics:

- *SE - EcoStruxure MACA: Halstead Complexity, Cyclomatic Complexity*
- *SE - EcoStruxure CE-V: length, volume, difficulty, vg*
- *CODESYS Static Analysis: Halstead (D/HV/HL), Complexity (McCabe)*
- *SIG Sigrid Maintainability: Unit size, unit complexity (McCabe)*

Potential mitigations to enhance testability based on software complexity:

- Check whether the same functionality can be implemented with better understandability using less loops / nested networks.

- Check whether the number of used operators can be reduced, e.g., by refactoring too large POUs and re-distributing functionality.

Metric category: Information Exchange

Number of calls to other POUs

↑ ***High value***: 👍
↓ ***Low value***: 👎

Reason: *In case data is primarily exchanged via direct data exchange (calls) with low to zero data exchange via global variables, this supports testability since there is a lower risk of implicit or hidden dependencies that might be missed during testing.*

Implementation specification using available metrics:
- *SE - EcoStruxure MACA: Call Out*
- *SE - EcoStruxure CE-V: callproc*

⦸ ***Potential antagonist:***
↓ ***Efficiency*** *A high number of calls might impair the software's efficiency during runtime.*

Number of read / write accesses on variables

↑ ***High value***: 👎
↓ ***Low value***: 👍

Reason: *A high number of read and write accesses on variables indicate a high number of dependencies that need to be considered during testing.*

Implementation specification using available metrics:
- *SE - EcoStruxure MACA: Number Of Writes, Number Of Reads*
- *SE - EcoStruxure CE-V: maxmemwrite, maxmemread*

Informationflow from POU to its environment

↑ ***High value***: 👎
↓ ***Low value***: 👍

Reason: *A high number of information flow from a POU to its environment may indicate a high number of dependencies that need to be considered during testing.*

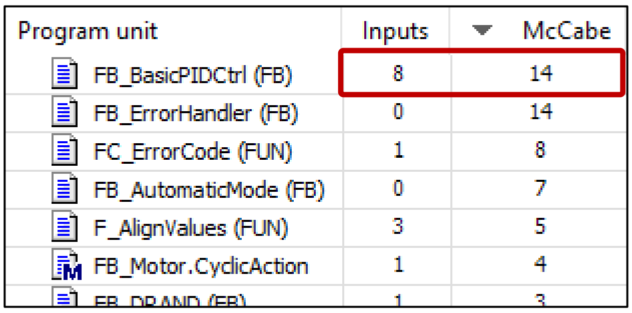
Implementation specification using available metrics:
- *SE - EcoStruxure MACA: Fan Out*

Potential mitigations to enhance testability based on information exchange:
- Check whether some of the outgoing information flows from a POU can be reduced, e.g., by splitting-up a POUs functionality and re-distributing its functionality to further POUs, methods, or actions.
- To reduce high Fan Out – create an intermediate POU to factor out modules with low coupling and high cohesion
- Simplify data structures.

Metric category: Reuse Indicators
Call depth ↑ ***High value***: 👎 ↓ ***Low value***: 👍 ***Reason:*** *A high call depth indicates that the POU under test is part of a long preceding call chain, i.e., changes of the POU may lead to cross effects.* ***Implementation specification using available metrics:*** - *SE - EcoStruxure CE-V: calldepthmin, calldepthmax*
Duplication ↑ ***High value***: 👎 ↓ ***Low value***: 👍 **Reason:** *When code is copy-pasted, the testing effort or the effort for making changes increases due to increase in the code amount and the need of managing separate copies in multiple places for needed changes.* ***Implementation specification using available metrics:*** - *CODESYS Static Analysis: Duplication ratio* - *SIG Sigrid Maintainability: Duplication* 🛇 ***Potential antagonist:*** ↓ *An increase in dependencies where code is reused via referring to it (such as function calls or OO mechanisms)*
Potential mitigations to enhance testability based on reuse indicators: • In case a software project (or parts of it) is characterized by high call depths, it should be considered whether the call depth can be reduced by merging functionalities from two or more POUs into one. • Use systematic reusable mechanisms such as defining functions or facilitating OO mechanisms, such as inheritance.

Disclaimer: *The subsequent code excerpts are extracted from an idealized, didactic example of a water heater and do not necessarily represent the extent of a complete industrial application. The provided code snippets are independent from each other, i.e., the compliant code example is not an optimization of the non-compliant example. Due to the nature of the supplied program, the analyses have been conducted through the utilization of CODESYS Static Analysis.*

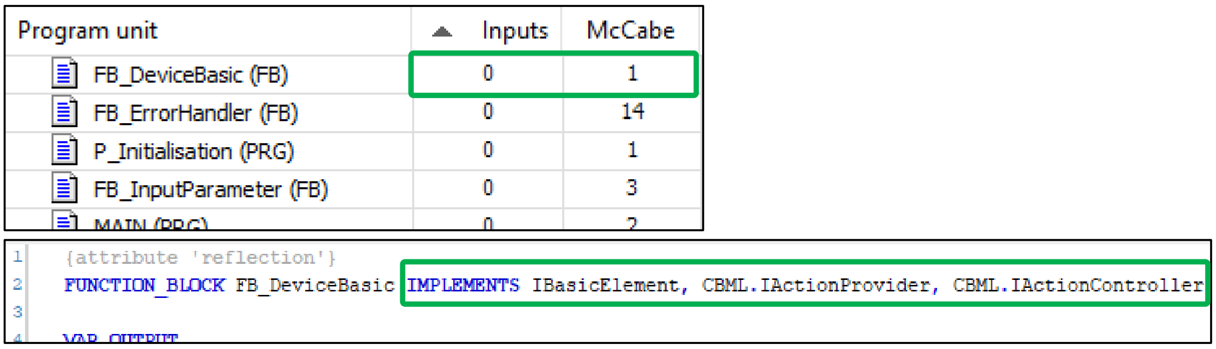
Noncompliant code example: The high number of input variables of the PID controller "FB_BasicPIDCtrl" requires a considerable effort for comprehensive tests covering all possible combinations of parameters. In combination with the complex nesting indicated by the outstanding McCabe Cyclomatic Complexity, this effect may be further amplified.



Program unit	Inputs	McCabe
FB_BasicPIDCtrl (FB)	8	14
FB_ErrorHandler (FB)	0	14
FC_ErrorCode (FUN)	1	8
FB_AutomaticMode (FB)	0	7
F_AlignValues (FUN)	3	5
FB_Motor.CyclicAction	1	4
FB_DRAND (FB)	1	3

Figure 13. Metrics table from CODESYS sorted by McCabe Complexity

<u>Compliant code example:</u> In contrast, a POU with minimal to no dependencies to the environment indicates a reduced effort and time in testing. In this specific instance of "FB_DeviceBasic", this has been achieved through the utilization of interfaces, enabling separate testing procedures that need only be conducted once for all POUs they are implemented in.



Program unit	Inputs	McCabe
FB_DeviceBasic (FB)	0	1
FB_ErrorHandler (FB)	0	14
P_Initialisation (PRG)	0	1
FB_InputParameter (FB)	0	3
MAIN (PRG)	0	2

```
{attribute 'reflection'}
FUNCTION_BLOCK FB_DeviceBasic IMPLEMENTS IBasicElement, CBML.IActionProvider, CBML.IActionController

VAR_OUTPUT
```

Figure 14. Metrics table from CODESYS with associated code snippet of "FB_DeviceBasic" emphasizing the use of interfaces

5.4. Recommendation: Metric-based Assessment of Efficiency

<u>Use cases and corresponding workflow steps (cf. Section 4.2):</u>

- **Use Case 1**: Continuous quality checks of POUs during software development.
  - *Motivation*: Check whether a specific change affects the software's efficiency and overview how the software's reusability develops over time.
- **Use Case 2**: Comparison of code before and after commissioning.
  - *Motivation*: Check whether adaptions during the start-up phase had an impact on the software's efficiency and a revision of the changed parts is required.
- **Use Case 3**: Plant or machine audit after a project has been finished.
  - *Motivation*: Assess the efficiency of a project's POUs in relation to other projects to identify strengths, weaknesses and optimization potentials in the given design decisions regarding, e.g., in relation to the performance of the used automation hardware.

Description of the required metrics:

Metric category: Code Size

Memory allocation

↑ ***High value***: 👎
↓ ***Low value***: 👍

Reason: *Previous investigations showed that the assignments of variables to an FB instance when calling it may negatively affect the software's performance in case the respective FB has a high memory size.*

Implementation specification using available metrics:
- *CODESYS Static Analysis: Code size, Stack size*
- *SE - EcoStruxure MACA: Memory Size (Data)*
- *SE - EcoStruxure CE-V: Memory size*

Number of actions

↑ ***High value***: 👎
↓ ***Low value***: 👍

Reason: *A high number of actions may cause a high number of calls, which may have a negative impact on the performance of the calling software element.*

Implementation specification using available metrics:
- *SE – EcoStruxure MACA: Number Of Actions*
- *SE – EcoStruxure CE-V : plcobjecttype counter*
- *SIG Sigrid Maintainability: Unit size*

🛇 ***Potential antagonist:***
↓ ***Maintainabiliy & Testability*** *in case of decreased encapsulation of functionality*

Potential mitigations to enhance efficiency based on code size:
- Structure functionality by encapsulating coherent code parts into actions (or methods) that can be called by the POU. However, for highly time-critical software parts, avoid unnecessarily long call chains.

Metric category: Variables and POU Interfaces

Usage of global variables

↑ ***High value:*** 👎
↓ ***Low value:*** 👍

Reason: *GVL usage may impair efficiency in case there is a lot of global data traffic.*

Implementation specification using available metrics:

- *SE - EcoStruxure MACA: Number of GVL Usages*
- *SE - EcoStruxure CE-V: extvarref*
- *CODESYS Static Analysis: Used different global variables*

Usage of local variables

↑ ***High value:*** 👎
↓ ***Low value:*** 👍

Reason: *A high number of local variables may indicate a high amount of POU-internal data traffic that might impair its performance efficiency.*

Implementation specification using available metrics:

- *SE - EcoStruxure MACA: Number Of Variables*
- *CODESYS Static Analysis: Number of local variables*

Potential mitigations to enhance efficiency based on variables and POU interfaces:

- Check whether the usage of global variables is inevitably necessary or whether relevant information could also be received directly by POUs via direct data exchange.
- In case global variables cannot be avoided, check whether the global variables are reasonably structured (e.g., functional coherent variables arranged together in STRUCTs)
- Check why POUs with outstanding large interfaces need this amount of external information. Maybe the functionality can be split-up and distributed across more POUs (or methods / actions).
- Check if different parameters from input or output could be grouped together in a data structure when there are produced and consumed at similar places and create the associated structure.
- Create function interfaces that pass data through arguments instead of global variables.

Metric category: OO-IEC Elements

Implementation of Interfaces

↑ ***High value:*** 👎
↓ ***Low value:*** 👍

Reason: *Performance analyses in industrial PLC software have confirmed that a high number of method and property calls via interfaces may lead to performance issues at runtime [15].*

Implementation specification using available metrics:

- *SE - EcoStruxure MACA: Implements*

⦸ ***Potential antagonist:***

↓ ***Reusability & Testability*** *Using inheritance, reusability of the functionality of a base class is enhanced and testing effort can be reduced since testing the base class behavior also covers the FBs derived from it.*

<u>Encapsulation of functionality and information in properties and methods</u>

↑ ***High value***: 👎
↓ ***Low value***: 👍

Reason: *Encapsulation of code in smaller units is beneficial regarding modularity, but may lead to impaired performance in case of an increased number of calls.*

Implementation specification using available metrics:

- *SE - EcoStruxure MACA: Number Of Properties, Number Of Methods*

🛇 ***Potential antagonist:***
↓ ***Maintainability & Reusability & Testability*** *in case of lack of encapsulation of functionality*

<u>Potential mitigations to enhance efficiency based on OO-IEC elements:</u>

- Too high coupling between methods or a lack of cohesion can be an indicator that the functionality distribution across methods is not ideal. It should be checked whether the concerned methods can be refactored or whether functionality can be re-distributed.

Metric category: Information Exchange

<u>Incoming direct data exchange between POUs via calls</u>

↑ ***High value***: 👎
↓ ***Low value***: 👍

Reason: *A high number of calls may lead to impaired performance at runtime.*

Implementation specification using available metrics:

- *SE - EcoStruxure MACA: Call Out*
- *SE - EcoStruxure CE-V: callproc*

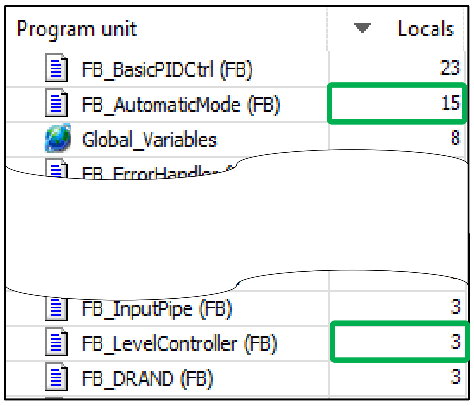
🛇 ***Potential antagonist:***
↓ ***Reusability & Testability:*** *Monolithic POUs instead of encapsulating and distributing functionality and using it by calling the respective POU may impair reusability and testability.*

<u>Potential mitigations to enhance efficiency based on information exchange:</u>

- Check whether some of the outgoing information flows from a POU can be reduced, e.g., by splitting-up a POUs functionality and re-distributing its functionality to further POUs, methods, or actions.
- Simplify data structures.

<u>Disclaimer:</u> The subsequent code excerpts are extracted from an idealized, didactic example of a water heater and do not necessarily represent the extent of a complete industrial application. The provided code snippets are independent from each other, i.e., the compliant code example is <u>not</u> an optimization of the non-compliant example. Due to the nature of the supplied program, the analyses have been conducted through the utilization of CODESYS Static Analysis.

<u>Compliant/Noncompliant code examples:</u> It is not feasible to universally state the exact threshold at which a POU may encounter runtime issues and leads to a substantial decline in efficiency. However, an increased level of internal POU data traffic, induced by the higher quantity of local variables, may have a negative impact on the runtime of e.g., “FB_AutomaticMode” compared to an FB with less internal data traffic, e.g., “FB_LevelControl”.



Program unit	Locals
FB_BasicPIDCtrl (FB)	23
FB_AutomaticMode (FB)	15
Global_Variables	8
FB_ErrorHandler	
FB_InputPipe (FB)	3
FB_LevelController (FB)	3
FB_DRAND (FB)	3

Figure 15. Excerpt from metrics table showing the range in the number of local variables used

5.5. Recommendation: Metric-based Assessment of Reliability

__Use cases and corresponding workflow steps (cf. Section 4.2):__

- **Use Case 1**: Continuous quality checks of POUs during software development
  - *Motivation*: Check whether a specific change affects the software's efficiency and overview how the software's reliability develops over time
- **Use Case 2**: Comparison of code before and after commissioning
  - *Motivation*: Check whether adaptions during the start-up phase had an impact on the software's reliability and a revision of the changed parts is required.
- **Use Case 3**: Plant or machine audit after a project has been finished
  - *Motivation*: Assess the reliability of a project's POUs in relation to other projects to identify strengths, weaknesses and optimization potentials in the given design decisions regarding, e.g., to plan and prioritize test cases in the case of comprehensive / critical changes that might affect the software's reliable functionality.

__Description of the required metrics:__

Metric category: Language-specific Metrics
__Number of transitions in SFC__ ↑ ***High value***: 👎 ↓ ***Low value***: 👍 ***Reason:*** *A high level of branching in SFC code can make the code confusing and make it difficult to perform changes. Possible cross-relationships can be missed, running the risk of compromising reliable functionality.* ***Implementation specification using available metrics:*** - *SE - EcoStruxure MACA: Number Of Transitions* - *SE - EcoStruxure CE-V: nbofbranches, g7height, g7width* - *CODESYS Static Analysis: Number of SFC branches, Number of SFC steps*
__Number of networks in FBD__ ↑ ***High value***: 👎 ↓ ***Low value***: 👍 ***Reason:*** *A high number of networks in an FBD implementation may indicate a high scope of functionality assigned to an individual POU, which may hamper to keep track of its reliable functionality.*

Implementation specification using available metrics:

- *SE - EcoStruxure MACA: Number Of FBD Networks, Halstead Complexity for FBD*

Potential mitigations to enhance reliability based on language-specific parameters:

- In case a SFC/FBD is not readable with one look (requiring zooming out or scrolling), it should be considered whether its functionality should be distributed across several POUs.
- In case individual FBD networks stand out with high complexity values, try to distribute their functionality across multiple networks.
- Use FBD to develop interconnected modules rather than simple POUs which leads to multiple networks. Interconnected modules have fewer networks.

Metric category: Variables and POU Interfaces

Usage of global variables

↑ ***High value***: 👎
↓ ***Low value***: 👍

Reason: *Changes to POUs that initially act independently have a major impact through indirect use of POU data via global variables. Further, finding errors is more difficult in case cross references via global variables need to be traced.*

Implementation specification using available metrics:

- *SE - EcoStruxure MACA: Number of GVL Usages*
- *SE - EcoStruxure CE-V: extvarref*
- *CODESYS Static Analysis: Used different global variables*

Potential mitigations to enhance reliability based on variables and POU interfaces:

- Check whether the usage of global variables is inevitably necessary or whether relevant information could also be received directly by POUs via direct data exchange.
- In case global variables cannot be avoided, check whether the global variables are reasonably structured (e.g., functional coherent variables arranged together in STRUCTs).
- Check why POUs with outstanding large interfaces need this amount of external information. Maybe the functionality can be split-up and distributed across more POUs (or methods / actions).
- Check if different parameters from input or output could be grouped together in a data structure when there are produced and consumed at similar places and create the associated structure.
- Create function interfaces that pass data through arguments instead of global variables.

Metric category: OO-IEC Elements
Coupling of objects to the environment ↑ ***High value:*** 👎 ↓ ***Low value:*** 👍 ***Reason:*** *A high number of dependencies of a software unit to its environment may lead to cross-effects in case of changes that might impair the software's reliable functionality.* ***Implementation specification using available metrics:*** - *CODESYS Static Analysis: CBO - Coupling Between Objects* - *SIG Sigrid Maintainability: Module coupling*
Potential mitigations to enhance reliability based on OO-IEC elements: • Too high coupling between methods or a lack of cohesion can be an indicator that the functionality distribution across methods is not ideal. It should be checked whether the concerned methods can be refactored or whether functionality can be re-distributed.

Metric category: Software Complexity
Complexity and length of source code ↑ ***High value:*** 👎 ↓ ***Low value:*** 👍 ***Reason:*** *The higher the software complexity, the greater the risk of introducing errors in case of a change and thus compromising the reliable functionality of the system.* ***Implementation specification using available metrics:*** - *SE - EcoStruxure MACA: Halstead Complexity, Cyclomatic Complexity* - *SE - EcoStruxure CE-V: length, volume, difficulty, vg* - *CODESYS Static Analysis: Halstead (D/HV/HL), Complexity (McCabe)* - *SIG Sigrid Maintainability: Unit size, unit complexity (McCabe)*
Potential mitigations to enhance reliability based on software complexity: • Check whether the same functionality can be implemented with better understandability using less loops / nested networks. • Check whether the number of used operators can be reduced, e.g., by refactoring too large POUs and re-distributing functionality.

Metric category: Information Exchange

Ingoing direct data exchange between POUs via calls

↑ ***High value:*** 👍
↓ ***Low value:*** -

Reason: *A high number of calls to a POU indicates that its functionality is reused by many other software units. This increases the probability of detecting and correcting errors and thus improving reliability.*

Implementation specification using available metrics:

- *SE - EcoStruxure MACA: Call In*
- *SE - EcoStruxure CE-V: calledcouns*
- *CODESYS Static Analysis: Number of calls*

Metric category: Reuse Indicators

Number of Library References

↑ ***High value:*** 👍
↓ ***Low value:*** 👎

Reason: *The use of library components, which have usually been extensively tested for various applications, usually increases reliability.*

Implementation specification using available metrics:

- *SE - EcoStruxure MACA: Number Of Library References*

Call depth

↑ ***High value:*** 👎
↓ ***Low value:*** 👍

Reason: *If changes are made to POUs that are located at the bottom of long call chains, there is a risk that errors that may have been introduced will affect the calling POUs and thus worsen reliability.*

Implementation specification using available metrics:

- *SE - EcoStruxure CE-V: calldepthmin, calldepthmax*

Duplication

↑ ***High value:*** 👎
↓ ***Low value:*** 👍

Reason: *When code is copied, potential bugs in the concerned software parts are also copied. Fixing these faults is highly time-consuming, as code must be adapted at distributed locations to maintain reliability.*

Implementation specification using available metrics:

- *CODESYS Static Analysis: Duplication ratio*
- *SIG Sigrid Maintainability: Duplication*

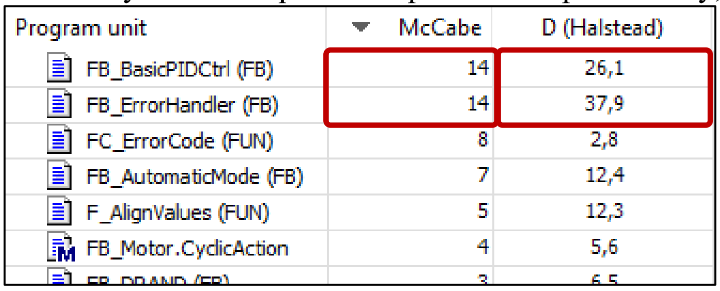
🛇 ***Potential antagonist:***
↓ *An increase in dependencies where code is reused via referring to it (such as function calls or OO mechanisms)*

Potential mitigations to enhance reliability based on reuse indicators:

- In case a software project (or parts of it) is characterized by high call depths, it should be considered whether the call depth can be reduced by merging functionalities from two or more POUs into one.

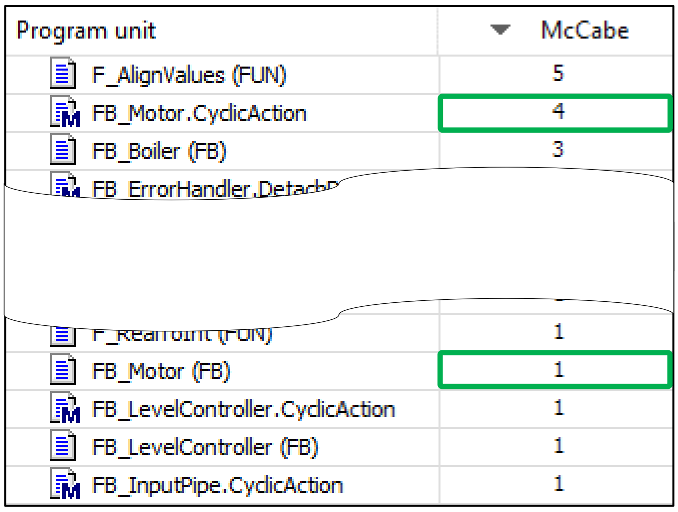
Disclaimer: The subsequent code excerpts are extracted from an idealized, didactic example of a water heater and do not necessarily represent the extent of a complete industrial application. The provided code snippets are independent from each other, i.e., the compliant code example is not an optimization of the non-compliant example. Due to the nature of the supplied program, the analyses have been conducted through the utilization of CODESYS Static Analysis.

Noncompliant code example: Similar to the previously introduced recommendation for maintainability in Section 5.1, it can be reasoned that the two outliers with significant Halstead Difficulty (D) and Cyclomatic Complexity (McCabe) can also lead to a reduction of reliability and hence may indicate a possible optimization potential by, e.g., encapsulation.



Program unit	McCabe	D (Halstead)
FB_BasicPIDCtrl (FB)	14	26,1
FB_ErrorHandler (FB)	14	37,9
FC_ErrorCode (FUN)	8	2,8
FB_AutomaticMode (FB)	7	12,4
F_AlignValues (FUN)	5	12,3
FB_Motor.CyclicAction	4	5,6
FB_DRAND (FB)	3	6,5

Compliant code example: The same argumentation is valid for the other compliant code example presented in Section 5.2, wherein the suggested mitigation of re-distributing functionality into smaller, less complex POUs may lead to improved reliability.



Program unit	McCabe
F_AlignValues (FUN)	5
FB_Motor.CyclicAction	4
FB_Boiler (FB)	3
FB_ErrorHandler.Detach	
F_RealPoint (FUN)	1
FB_Motor (FB)	1
FB_LevelController.CyclicAction	1
FB_LevelController (FB)	1
FB_InputPipe.CyclicAction	1

6. Threshold Values and Metric Application to Large-scale Industrial PLC Software Projects

In computer science, certain threshold values have been established to provide orientation for interpreting calculated metric values (cf. Table 2). The analysis and comparison of existing threshold values in the literature and available tools shows that even in high-level language development there is no general consensus on the range of these values. Even for the same metric, different authors and tool providers give diverging limits, e.g., for Cyclomatic Complexity (limit is 10 in the original source in McCabe [10], 15 in Squore Vector, only 2 according to Tarcísio et al. [16]). Nevertheless, the values can serve as an approximate guide also for IEC 61131-3-based PLC software to classify noticeable metric values – especially for Structured Text, which is similar to high-level languages in its syntax.

With the possibilities of modern programming platforms, e.g., in many CODESYS-based systems, the object-oriented extension of IEC 61131-3 can be conceptually used in such a way that PLC software is in no way inferior to high-level language software. For PLC software projects following current-day best practices such as object-orientation, it is therefore expected that the values in the table below could be used as sound reference values. Nevertheless, it is pointed out that there are often significant differences in the common practice of PLC software development in machine and plant manufacturing compared to high-level language software, which can limit the direct transfer of threshold values from computer science, especially, e.g., for graphical and/or procedural PLC software. The derivation of appropriate thresholds is a heuristic approach and should be based on logically arguable conclusions, e.g., based on cause-effect analyses (e.g., complexity value above which POU is considered unmaintainable).

One such approach connecting the measurement to the impact is presented in [17] by Alves et al. The approach presents a technology agnostic benchmark-based metric evaluation approach. In [1], Wijnmaalen et al. have performed a replication study showing that there is a correlation between issue (defect fixing, enhancement, patching, etc.) resolution times and measured maintainability metrics. The higher maintainable a system is, the faster changes can be made.

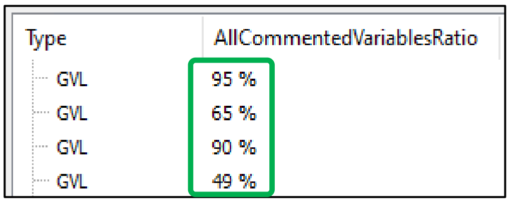
Table 2. Collection of suggested threshold values for high-level programming languages, from applications as well as research (LOC = Lines of Code, DIT=Depth of Inheritance Tree, CBO=Coupling Between Objects, NOC=Number Of Children, LCOM= Lack of Cohesion Of Methods, RFC=Response For Class)

Source	**Metric (coherent with presented metrics for IEC 61131-3)**	**Suggested threshold values in high level language tools and literature**
Squore Vector (from an example back-end project in php) [18]	Cyclomatic Complexity	≤ 15
	Number of Parameters of Method	≤ 5
	Number of Header Comments	> 0
	Number of Executable Statements	≤ 50
McCabe (intro of complexity metric) [10, 19]	Cyclomatic Complexity	≤ 10
embold Metric Thresholds (Component Level) [20]	Lines of Code	≤ 1000
	Comment Ratio	> 30
	LCOM	≤ 0.77
	Cyclomatic Complexity	50
PHP Mess Detector [21]	Cyclomatic Complexity	≤ 10
	Lines of Code (Classes)	≤ 1000
	Number of methods	≤ 25
Holzmann (NASAs 10 rules) [22]	LOC	≤ 60
Shatnawi Raed (analysis of 11 open-source java projects) [23]	DIT	≤ 3
	CBO	≤ 17
	DIT	≤ 2
	NOC	≤ 1

Source	**Metric (coherent with presented metrics for IEC 61131-3)**	**Suggested threshold values in high level language tools and literature**
Tarcísio et al. (analysis of Qualitas Corpus: a curated collection of software systems in java)[16]	LCOM	≤ 0.167
	Cyclomatic Complexity	≤ 2
	Number of Parameters	≤ 2
Rosenberg et al. (expert audit by NASAs Software Assurance Technology Center in C++/Java) [24]	DIT	2 – 5
	CBO	≤ 5
	RFC	≤ 50
Herbold, Grabowski, Waack (collection of research results) [25]	Cyclomatic Complexity for C	≤ 24
	Cyclomatic Complexity for C++	≤ 10
	Cyclomatic Complexity for C#	≤ 10
	CBO for Java	≤ 5
	RFC for Java	≤ 100
Alves et al. (technology-agnostic benchmark-based metric calculation) [17]. New thresholds cf. [27].	Unit size (Lines of Code)	Low risk: 15 , Moderate Risk: 16-30, High Risk: 31 - 60, Very-High Risk: 61+
Please note that the developers do not have to implement every POU strictly using the thresholds of low risk categories. The approach uses a risk-based measurement to evaluate the codebase as a whole, using a benchmark to determine how metric values compare to the state of the practice.	Unit complexity (Cyclomatic Complexity)	Low risk: 1-5, Moderate Risk: 6-10, High Risk: 11-25, Very-High Risk: 26+
	Module coupling (Incoming dependencies per POU)	Low risk: 0-10, Moderate Risk: 11-20, High Risk: 21-50, Very-High Risk: 51+

In PLC software, the usage of different programming languages and diverse implemented functionalities in POUs (ranging from performing simple calculations to controlling complex technological modules) make the definition of rigid threshold values quite difficult. This guideline therefore does not specify fixed threshold values to avoid misinterpretation of the metrics. However, to provide an orientation for more realistic metric values in industrial application, the following part of the section applies examples of the introduced metrics in Section 5 to realistic industrial code examples.
The subsequent code excerpts are extracted from two real-world industrial projects that have been recognized as representative of a diverse range of industrial applications. The analyses have been conducted with the *SE - EcoStruxure Machine Advisor Code Analysis* integrated in the programming environment *SE – EcoStruxure Machine Expert*.

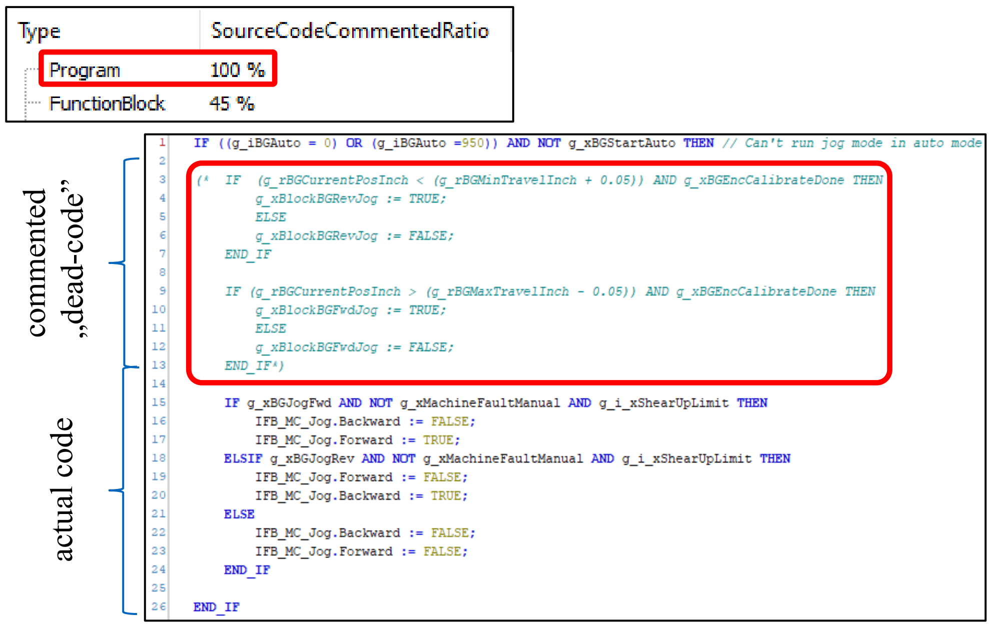
Code Documentation: Well-structured industrial example: Upon performing a code analysis, the software developer is presented with a comprehensive overview of the meticulous documentation of the GVLs. This valuable information could serve as a means of reassurance prior to committing the variable lists, or as confirmation of adherence to the internal coding guidelines of the company.



Type	AllCommentedVariablesRatio
GVL	95 %
GVL	65 %
GVL	90 %
GVL	49 %

Figure 16. Metrics table from Schneider Electric showing the good documentation of GVLs.

Code Documentation – Thread to validity: Although metrics may indicate a comprehensive documentation of this POU, it is apparent that this metric has been subject to distortion through leaving in commented "dead-code" which brings none of the positive effects of actual code documentation. As such, one must always be mindful of the potential threads on the validity of the code analysis. Metrics related to code documentation are especially prone to intentional or unintentional manipulation, but with proper attention and diligence, such concerns can be addressed and resolved.

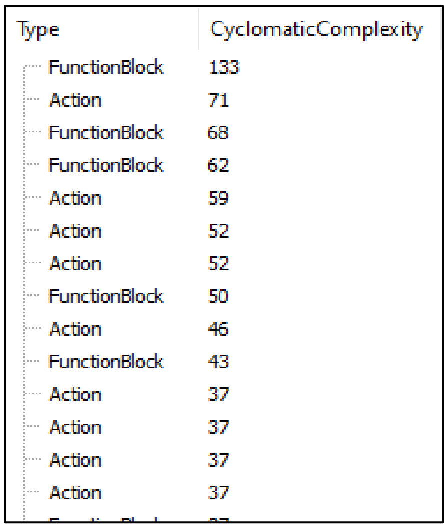


Type	SourceCodeCommentedRatio
Program	100 %
FunctionBlock	45 %

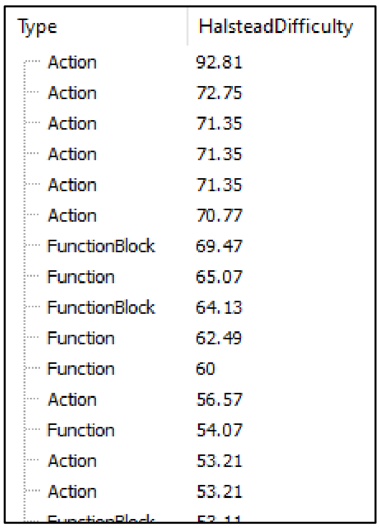
```
IF ((g_iBGAuto = 0) OR (g_iBGAuto =950)) AND NOT g_xBGStartAuto THEN // Can't run jog mode in auto mode

(*  IF  (g_rBGCurrentPosInch < (g_rBGMinTravelInch + 0.05)) AND g_xBGEncCalibrateDone THEN
        g_xBlockBGRevJog := TRUE;
        ELSE
        g_xBlockBGRevJog := FALSE;
    END_IF

    IF (g_rBGCurrentPosInch > (g_rBGMaxTravelInch - 0.05)) AND g_xBGEncCalibrateDone THEN
        g_xBlockBGFwdJog := TRUE;
        ELSE
        g_xBlockBGFwdJog := FALSE;
    END_IF*)

    IF g_xBGJogFwd AND NOT g_xMachineFaultManual AND g_i_xShearUpLimit THEN
        IFB_MC_Jog.Backward := FALSE;
        IFB_MC_Jog.Forward := TRUE;
    ELSIF g_xBGJogRev AND NOT g_xMachineFaultManual AND g_i_xShearUpLimit THEN
        IFB_MC_Jog.Forward := FALSE;
        IFB_MC_Jog.Backward := TRUE;
    ELSE
        IFB_MC_Jog.Backward := FALSE;
        IFB_MC_Jog.Forward := FALSE;
    END_IF

END_IF
```

Figure 17. Code snippet demonstrating the common malpractice of commenting out "dead-code" with its associated distorted metrics table.

Software Complexity – Remarkably complex industrial example: The metrics table presented herein originates from a real-world industrial application, that has been identified to be a negative example with ample potential for improvements, especially regarding maintainability. Nevertheless, this example serves to emphasize the variability in numerical values of metrics, particularly in comparison with the idealized, didactic non-/compliant examples provided as an initial introduction to metrics in Section 5. Furthermore, it demonstrates the inherent impossibility of a universal recommendation for upper and lower bounds of metric values, as these must always be evaluated on a case-by-case basis by domain experts.



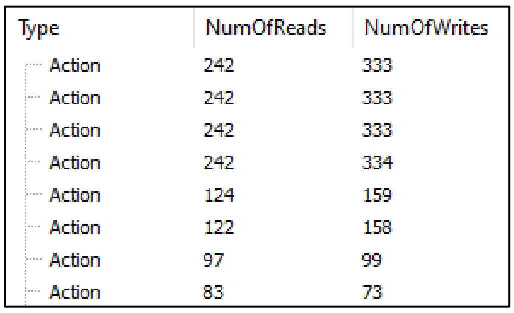
Type	CyclomaticComplexity
FunctionBlock	133
Action	71
FunctionBlock	68
FunctionBlock	62
Action	59
Action	52
Action	52
FunctionBlock	50
Action	46
FunctionBlock	43
Action	37
Action	37
Action	37
Action	37



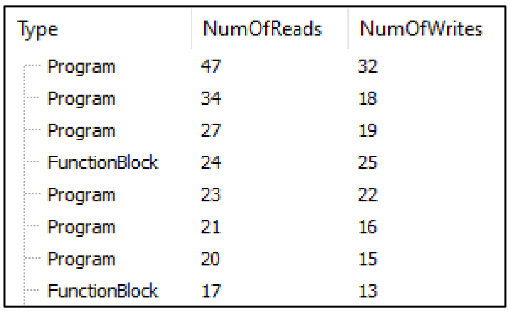
Type	HalsteadDifficulty
Action	92.81
Action	72.75
Action	71.35
Action	71.35
Action	71.35
Action	70.77
FunctionBlock	69.47
Function	65.07
FunctionBlock	64.13
Function	62.49
Function	60
Action	56.57
Function	54.07
Action	53.21
Action	53.21

Figure 18. Metrics table from Schneider Electric showing the possibly high numerical values.

Information Exchange – Industrial example comparison: This example aims to further substantiate the range of possible maximum metric values by comparing two distinct applications. Consequently, it is generally recommended to refrain from solely focusing on remaining within arbitrary limits of metric values. Instead, it is advisable to initiate an examination of outliers and overarching patterns, as well as to compare software metrics within the same domain or with projects that are known to be similar.



Type	NumOfReads	NumOfWrites
Action	242	333
Action	242	333
Action	242	333
Action	242	334
Action	124	159
Action	122	158
Action	97	99
Action	83	73



Type	NumOfReads	NumOfWrites
Program	47	32
Program	34	18
Program	27	19
FunctionBlock	24	25
Program	23	22
Program	21	16
Program	20	15
FunctionBlock	17	13

Figure 19. Two metrics tables of two different industrial applications, emphasizing the variability of the numerical values between different projects.

References

[1] J. Wijnmaalen, C. Chen, D. Bijlsma, and A. M. Oprescu, “The Relation between Software Maintainability and Issue Resolution Time: A Replication Study,” in *SaTToSE*, 2019.
[2] H. Zhu, Q. Zhang, and Y. Zhang, “HASARD: A Model-Based Method for Quality Analysis of Software Architecture,” in *Relating System Quality and Software Architecture*: Elsevier, 2014, pp. 123–156.
[3] B. W. Boehm, J. R. Brown, and M. Lipow, “Quantitative evaluation of software quality,” in *International Conference on Software Engineering*, 1976.
[4] R. G. Dromey, “A model for software product quality,” *IEEE Transactions on Software Engineering*, vol. 21, no. 2, pp. 146–162, 1995.
[5] J. A. McCall, P. K. Richards, and G. F. Walters, “Factors in Software Quality. Volume I. Concepts and Definitions of Software Quality,” 1977.
[6] *Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models*, 25010, ISO/IEC, 2011.
[7] B. Vogel-Heuser, A. Fay, I. Schaefer, and M. Tichy, “Evolution of software in automated production systems: Challenges and research directions,” *Journal of Systems and Software*, vol. 110, pp. 54–84, 2015, doi: 10.1016/j.jss.2015.08.026.
[8] *IEEE Standard for a Software Quality Metrics Methodology*, 1061, IEEE, Piscataway, NJ, USA, 1998.
[9] S. R. Chidamber and C. F. Kemerer, “A metrics suite for object oriented design,” *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994.
[10] T. J. McCabe, “A Complexity Measure,” *IEEE Transactions on Software Engineering*, SE-2, no. 4, pp. 308–320, 1976, doi: 10.1109/tse.1976.233837.
[11] M. H. Halstead, *Elements of software science*. New York, NY: North-Holland, 1977.
[12] E. M. Neumann *et al.,* “Metric-based Identification of Target Conflicts in the Development of Industrial Automation Software Libraries,” in *IEEE International Conference on Industrial Engineering and Engineering Management (IEEM)*, Kuala Lumpur, Malaysia, 2022, pp. 1493–1499.
[13] *Programmable controllers - Part 3: Programming languages (IEC 61131-3:2013); German version EN 61131-3:2013*, 61131-3, International Electrotechnical Commission (IEC), 2014.
[14] *Systems and software engineering - Vocabulary*, 24765, ISO/IEC/IEEE, Piscataway, NJ, USA, 2010.
[15] E.-M. Neumann *et al.,* “Identifying Runtime Issues in Object-Oriented IEC 61131-3-Compliant Control Software using Metrics,” in *Annual Conference of the IEEE Industrial Electronics Society (IECON)*, Singapore, Singapore, 2020, pp. 259–266.
[16] Tarcísio G. S. Filó, “A Catalogue of Thresholds for Object-Oriented Software Metrics,” pp. 48–55, 2015. [Online]. Available: http://personales.upv.es/thinkmind/dl/conferences/softeng/softeng_2015/softeng_2015_3_10_55070.pdf
[17] T. L. Alves, J. P. Correia, and J. Visser, “Benchmark-Based Aggregation of Metrics to Ratings,” in *Joint Conf. of the 21st Int. Workshop on Software Measurement and the 6th Int. Conf. on Software Process and Product Measurement*, Nara, Japan, 2011, pp. 20–29.
[18] Vector Informatik GmbH, *Squore: Analytics for Projects Monitoring.* [Online]. Available: https://www.vector.com /int/en/products/products-a-z/software/squore/ (accessed: Jul. 18 2023).
[19] A. H. Watson and T. J. McCabe, “Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric,” 1996.
[20] embold, *Metric Thresholds.* [Online]. Available: https://docs.embold.io/de/metric-thresholds/ (accessed: Jul. 18 2023).
[21] PHP, *Mess Detector - Code Size Rules.* [Online]. Available: https://phpmd.org/rules/codesize.html
[22] G. J. Holzmann, “The Power of Ten - Rules for Developing Safety Critical Code,” 2006.
[23] R. Shatnawi, “Deriving metrics thresholds using log transformation,” *Journal of Software: Evolution and Process*, vol. 27, no. 2, pp. 95–113, 2015.
[24] L. Roseberg, R. Stapko, and A. Gallo, “Risk-based Object Oriented Testing,” 1999. [Online]. Available: https:// citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=0cadbbdc244f38e4dfbae022b5e5e3fdc8249dd0
[25] S. Herbold, J. Grabowski, and S. Waack, “Calculation and optimization of thresholds for sets of software metrics,” *Empirical Software Engineering*, vol. 16, no. 6, pp. 812–841, 2011.
[26] T. L. Alves, C. Ypma, and J. Visser, “Deriving metric thresholds from benchmark data,” in *IEEE International Conference on Software Maintenance (ICSM)*, Timișoara, Romania, 2010.
[27] SIG/TÜViT, *Evaluation Criteria Trusted Product Maintainability: Guidance for producers (V15.0).* [Online]. Available: https://www.softwareimprovementgroup.com/software-analysis/

Appendix 1 Table to map available metrics and software quality attributes

LEGEND

++	strong positive correlation between metric value and quality attribute
+	rather positive correlation between metric value and quality attribute
o	neutral - no correlation between metric and quality attribute
-	rather negative correlation between metric value and quality attribute (higher metric values indicate that quality attribute is less fulfilled)
--	strong negative correlation between metric value and quality attribute (higher metric values indicate that quality attribute is less fulfilled)

Size Metrics

CODESYS Group		***Schneider Electric – EcoStruxure Machine Advisor Code Analysis***		***Schneider Electric – Control Engineering – Verification (formerly Itris PLC Checker)***		***Software Improvement Group Sigrid***		***Reliability***	***Efficiency***	***Maintainability***	***Reusability***	***Testability***
Metric	Description	*Metric*	Description	*Metric*	Description	*Metric*	Description					
NOS	Number of statements.			*Number of instructions*	Number of instructions (Affect, Conditional instruction, loop instructions, break instructions) in a given POU.			*o*	*o*	-	-	-
		Lines Of Code (LOC)	Counts the number of source code lines of a program.			*Unit size*	Lines of Code (LoC) per unit (function, function block, network, etc. – based on programming language). For visual languages each element (in the diagram) is counted as one LoC.	*o*	*o*	-	-	-
Code Size, Variable Size	Number of bytes.	*Memory Size (Data)*	Measures amount of memory allocation and processing for each instantiation of a complex type (type information and variables).	*Memory size*	Number of bits required.			*o*	-	*o*	*o*	*o*
		Number Of Actions	Information about how many actions are attached to a program or a function block.	*plcobjecttype counter*	For every kind of PLC supported by the technology, the tooling counts the number of elements found in the			*o*	-	+	*o*	+

CODESYS Group		***Schneider Electric – EcoStruxure Machine Advisor Code Analysis***		***Schneider Electric – Control Engineering – Verification (formerly Itris PLC Checker)***		***Software Improvement Group Sigrid***		***Reliability***	***Efficiency***	***Maintainability***	***Reusability***	***Testability***
Metric	Description	*Metric*	Description	*Metric*	Description	*Metric*	Description					
					application (function block, function, program, section SR, routine, etc.).							

Language-specific Size Metrics

CODESYS Group		***Schneider Electric – EcoStruxure Machine Advisor Code Analysis***		***Schneider Electric – Control Engineering – Verification (formerly Itris PLC Checker)***		***Software Improvement Group Sigrid***		***Reliability***	***Efficiency***	***Maintainability***	***Reusability***	***Testability***
Metric	Description	*Metric*	Description	*Metric*	Description	*Metric*	Description					
SFC branches	Number of SFC branches.			*nbofbranches*	Number of Branches in a given SFC routine.			--	*o*	--	-	--
SFC steps	Number of SFC steps.							-	*o*	-	-	-
		Number Of FBD Networks	Information about how many networks are available in an FBD implemented program, function block, function, method, or property.					-	*o*	-	-	-
		Halstead Complexity for FBD	Static testing method that analyzes the source code by breaking it down into tokens, classifying them, and counting them as operators or operands.					-	*o*	-	-	-
		Number Of Transitions	Information about how many transitions are attached to a program or a function block.					-	*o*	-	-	-
				g7height	Height of a given SFC routine counting each state and transition as one and divergence as no height.			-	*o*	-	-	-
				g7width	Maximum width of a given SFC routine in term of number of branches at the same level.			-	*o*	-	-	-

Variables and POU Interfaces

CODESYS Group		***Schneider Electric – EcoStruxure Machine Advisor Code Analysis***		***Schneider Electric – Control Engineering – Verification (formerly Itris PLC Checker)***		***Software Improvement Group Sigrid***		***Reliability***	***Efficiency***	***Maintainability***	***Reusability***	***Testability***
Metric	Description	*Metric*	Description	*Metric*	Description	*Metric*	Description					
Global	Number of different global variables.	*Number Of GVL Usages*	Information about how many global variables a programming object (programs, function blocks, functions, methods, etc.) uses (reading or writing).	*extvarref*	Number of external references in a POU (not a param, nor a local var).			-	-	--	--	*o*
I/Os	Number of direct object accesses.							*o*	*o*	*o*	-	-
Local	Number of local variables.	*Number Of Variables*	Information about how many variables are defined in the declaration part of programs, function blocks, functions, methods, property Get or Set, transitions, global variable lists, etc.					*o*	-	*o*	*o*	*o*
Inputs	Number of input variables.			*inputcount*	Number of input parameters of a given POU.			*o*	*o*	-	*o*	-
Outputs	Number of output variables.			*outputcount*	Number of output parameters of a given POU.			*o*	*o*	-	*o*	-
				nbofparam	Number of parameters (In, Out and InOut) of a given POU.	*Unit interfacing*	Number of parameters (In, Out and InOut) of a given POU.	*o*	*o*	-	*o*	-

Code Documentation

CODESYS Group		*Schneider Electric – EcoStruxure Machine Advisor Code Analysis*		*Schneider Electric – Control Engineering – Verification (formerly Itris PLC Checker)*		*Software Improvement Group Sigrid*		*Reliability*	*Efficiency*	*Maintainability*	*Reusability*	*Testability*
Metric	Description	*Metric*	Description	*Metric*	Description	*Metric*	Description					
Comments	Percentage of comments.	*Source Code Comment Ratio*	Calculates the %-ratio between CLOC (Comment Lines Of Code) and SLOC (Source Lines Of Code) of the implementation part of an object.	*percentage of comment*	Global ratio for the whole application.			*o*	*o*	++	*o*	*o*
		Commented Variables (All) Ratio	This metric calculates the %-ratio between commented and not commented variables in an object.	*result of Verification tool*				*o*	*o*	++	*o*	*o*
		Commented Variables (In + Out + Global) Ratio	Calculates the %-ratio between commented and not commented variables that are defined in VAR_GLOBAL, VAR_INPUT, VAR_OUTPUT, or VAR_IN_OUT.					*o*	*o*	++	*o*	*o*
		Number Of Multiline Comments	Counts the multiline comments in an object.					*o*	*o*	++	*o*	*o*
		Number Of Header Comment Lines	Counts the number of comments in the header of the declaration part.					*o*	*o*	++	*o*	*o*

OO-IEC Elements

CODESYS Group		***Schneider Electric – EcoStruxure Machine Advisor Code Analysis***		***Schneider Electric – Control Engineering – Verification (formerly Itris PLC Checker)***		***Software Improvement Group Sigrid***		***Reliability***	***Efficiency***	***Maintainability***	***Reusability***	***Testability***
Metric	Description	*Metric*	Description	*Metric*	Description	*Metric*	Description					
DIT	Depth of inheritance tree.							*o*	*o*	-	++	+
NOC	Number of children.	*Extended By*	Information about how often a function block or an interface is extended by another function block or interface.					*o*	*o*	-	++	+
		Extends	Information about how many interfaces are extended by a function block or an interface.					*o*	*o*	-	++	+
RFC	Response for class.							*o*	*o*	--	-	--
CBO	Coupling between objects.					*Component coupling*	Degree to which architectural components are depended on and depend on other components that make up a system.	-	*o*	-	--	--
LCOM	Lack of cohesion in methods.					*Component cohesion (rating refers to inverted value, cf. CODESYS metric)*	degree to which architectural components encapsulate specific business responsibilities / functionality within the system.	*o*	*o*	-	-	-
		Implemented By	Information about how often an interface is implemented by a function block.					*o*	*o*	*o*	+	+
		Implements	Information about how many interfaces are implemented by a function block.					*o*	-	*o*	+	+
		Number Of Methods	Information about how many methods are attached to a program or a function block.					*o*	-	+	+	+
		Number Of Properties	Information about how many properties are attached to a program or a function block.					*o*	-	+	+	+

Software Complexity

CODESYS Group		***Schneider Electric – EcoStruxure Machine Advisor Code Analysis***		***Schneider Electric – Control Engineering – Verification (formerly Itris PLC Checker)***		***Software Improvement Group Sigrid***		***Reliability***	***Efficiency***	***Maintainability***	***Reusability***	***Testability***
Metric	Description	*Metric*	Description	*Metric*	Description	*Metric*	Description					
HL (Halstead)	Halstead length (HL).			*length*				-	o	-	-	-
HV (Halstead)	Halstead volume (HV).			*volume*				-	o	-	-	-
D (Halstead)	Halstead difficulty (D).	*Halstead Complexity (for ST and FBD)*	Static testing method that analyzes the source code by breaking it down into tokens, classifying them, and counting them as operators or operands.	*difficulty*				-	o	--	-	--
McCabe	McCabe complexity.	*Cyclomatic Complexity*	Measure the complexity of a program by counting the number of linearly independent paths in the source code.	*vg*		*Unit complexity*	McCabe Cyclomatic Complexity per unit (function, function block, network, etc. – based on programming language). For visual languages, an element with multiple outgoing edges counts as branching points in the flow.	-	o	--	-	--

Information Exchange

CODESYS Group		*Schneider Electric – EcoStruxure Machine Advisor Code Analysis*		*Schneider Electric – Control Engineering – Verification (formerly Itris PLC Checker)*		*Software Improvement Group Sigrid*		*Reliability*	*Efficiency*	*Maintainability*	*Reusability*	*Testability*
Metric	Description	*Metric*	Description	*Metric*	Description	*Metric*	Description					
Calls	Number of calls	*Call In*	Information about who is calling a method, function, function block, etc.	*calledcount*	Number of reference calls to a given POU.			+	*o*	*o*	++	*o*
		Call Out	Information about which other objects (method, function, function block, etc.) are called by the POU.	*callproc*	Number of POU calls done in a given POU. No distinction is done between user code POUs and system POUs.			*o*	-	*o*	+	+
		Number Of Writes	Information about which variables are written.	*maxmemwrite*	Maximum number (in case of a structure with different members) of write done to the corresponding memory cell.			*o*	*o*	*o*	*o*	-
		Number Of Reads	Information about which variables are read.	*minmemwrite*	Minimum number (in case of a structure with different members) of write done to the corresponding memory cell.			*o*	*o*	*o*	*o*	-
		Fan Out	Information about how many outgoing dependencies (reads, writes, calls, etc.) a node in the analysis data model (Dependency Model) has.					*o*	*o*	--	-	--

Reuse Indicators

CODESYS Group		***Schneider Electric – EcoStruxure Machine Advisor Code Analysis***		***Schneider Electric – Control Engineering – Verification (formerly Itris PLC Checker)***		***Software Improvement Group Sigrid***		***Reliability***	***Efficiency***	***Maintainability***	***Reusability***	***Testability***
Metric	Description	*Metric*	Description	*Metric*	Description	*Metric*	Description					
		Number Of Library References	Information about how many libraries are directly referenced by an application or POU space.					+	*o*	*o*	*o*	*o*
				calldepthmin	Minimum depth level of a calling stack.			-	*o*	-	-	-
				calldepthmax	Maximum depth of calling stack (disclaimer: not calculable for recursion).			-	*o*	-	-	-
Clone ratio	Ratio of cloned code given in percent.					*Duplication*	The ratio of duplication (and/or redundancy) in a codebase caused by exact copy-paste patterns.	-	*o*	--	-	-