Journal of Universal Computer Science, vol. 30, no. 1 (2024), 25-48
submitted: 27/12/2022, accepted: 21/9/2023, appeared: 28/1/2024 CC BY-ND 4.0

Towards a set of metrics for hybrid (quantum/classical) systems maintainability

Ana Díaz Muñoz
(Institute of Technologies and Information Systems & Escuela Superior de Informática,
University of Castilla-La Mancha 13071 Ciudad Real, Spain
AQCLab Software Quality 13500 Ciudad Real, Spain
https://orcid.org/0000-0001-6515-8835, adiaz@aqclab.es)

Moisés Rodríguez Monje
(Institute of Technologies and Information Systems & Escuela Superior de Informática,
University of Castilla-La Mancha 13071 Ciudad Real, Spain
AQCLab Software Quality 13500 Ciudad Real, Spain
https://orcid.org/0000-0003-2155-7409, moises.rodriguez@uclm.es)

Mario Gerardo Piattini Velthuis
(Institute of Technologies and Information Systems & Escuela Superior de Informática,
University of Castilla-La Mancha 13071 Ciudad Real, Spain
https://orcid.org/0000-0002-7212-8279, mario.piattini@uclm.es)

Abstract: Given the rapid evolution that has taken place in recent years in the software industry, and along with it the emergence of quantum software, there is a need to design an environment for measuring quality metrics for hybrid, classic-quantum software.
In order to measure and evaluate the quality of classic software, there are models and standards, among which ISO/IEC 25000 stands out, which proposes a set of quality characteristics such as maintainability. However, there is currently no consensus for the measurement and evaluation of quantum software quality.
In this paper we propose a series of adaptations to "classic" metrics, as well as a set of new measurements for hybrid maintainability. Finally, a first prototype of a measurement tool developed as a SonarQube plugin, capable of measuring these metrics in quantum developments, is also presented.

Keywords: quantum software, software quality, ISO/IEC 25000, software assessment, maintainability
Categories: D
DOI: 10.3897/jucs.99348

1 Introduction

During the last few years there has been a great evolution in Information and Communication Technologies, with the emergence of the Internet, smartphones, wearables, IoT, Big Data, Smart cities, etc. But there is still a long way to go; since, after several years of research, quantum computing is starting to come to light.

As Chris Bernhardt says, "quantum computing is the beautiful fusion that comes from bringing together quantum physics and computer science" [Bernhardt, 20]. Since this is a totally novel subject, it is necessary to make as many valid contributions as possible, but without neglecting quality.

Since the beginning of the technological era, the importance of quality has been appreciated, and organizations are taking it into account when developing information systems. IT consumers are becoming more and more demanding, so they are no longer looking for a simple product or service, but for the one that offers the highest quality. But how is the quality of the systems evaluated? In order to evaluate the quality of a system, it is necessary to make measurements, establish thresholds and use tools that assist in the whole process. For this purpose, there are international standards, such as the ISO/IEC 25000 family [ISO, 14], which establish the basis for carrying out these quality evaluations.

On the other hand, if quantum computing is to be used in the coming years in sectors such as logistics, energy, agriculture, economics, health or chemistry, and if we have learned from the mistakes made throughout history in software engineering, then we must insist, from the beginning of the development of quantum systems, on the presence of quality. In the Talavera Manifesto for Quantum Software Engineering and Programming [Piattini, 20], we reaffirm the importance of ensuring the quality of quantum software.

It should be considered that researchers are currently trying to obtain the best possible results from quantum computing by making it hybrid, by combining it with traditional computing. The idea is to match the reliability of a classical computer with the strength of a quantum system, thus opening the doors to great discoveries. This pairing happens by accessing the high performance of quantum computers through classical computers, a fact that is expected to continue to be the case in the future [Pettersen, 21].

In the existing classical world, an infinity of tools are available that facilitate the results of measurements and evaluations on the quality of traditional software [Rodríguez, 19]; however, there are still no such tools for hybrid, classical and quantum software.

Considering the importance of quality and the situation of quantum software in full birth, this paper presents a completely novel contribution, consisting in evaluating the quality of hybrid code, specifically maintainability as the capability of evolution is one of the most important characteristics in this not so matured quantum software industry. Yes, even though quantum algorithms are well-defined and currently used as they are, quantum technology is constantly evolving and in its early stages, so it is expected to continue evolving rapidly in the coming years. This means that quantum software will also continue to evolve, seeking to take advantage of the latest innovations and improvements in quantum technology. Furthermore, the current quantum market is highly competitive and the ability to develop reliable and effective quantum software can make the difference between success and failure. Therefore, the aim of this paper is to present the first maintainability metrics for quantum code and an environment for measuring them on hybrid software, also demonstrating their application by analyzing a well-known quantum algorithm, such as Shor's algorithm [Shor, 97].

The rest of the paper is structured as follows: Section 2 presents the current state of quantum computing. Section 3 summarizes the classical metrics used to evaluate the maintainability of classical software. Subsequently, section 4 describes the new metrics applicable to hybrid software, which are the basis of the automated measurement environment. Section 5 shows the measurement environment created and one of the analyses performed using this environment. Finally, section 6 presents the conclusions obtained with this work and the future lines of research.

2 Quantum computing

Quantum computing sounds like the future, but in reality, it is already with us. Today, quantum computers are already available through the cloud, thanks to large companies and research groups such as Microsoft[1], IBM [IBM, 23], Amazon[2], Google [Google, 23], Cambridge Quantum Computing[3] and D-Wave[4].

Despite the great advances that have been taking place during the last few years in the field of quantum computing, and the fact that quantum systems are already available to be programmed, the future of computing will continue to be hybrid combining quantum and classical software [TS2, 23]. In this so-called hybrid computing model, the services offered by a quantum computer are accessed from a classical computer (Figure 1).

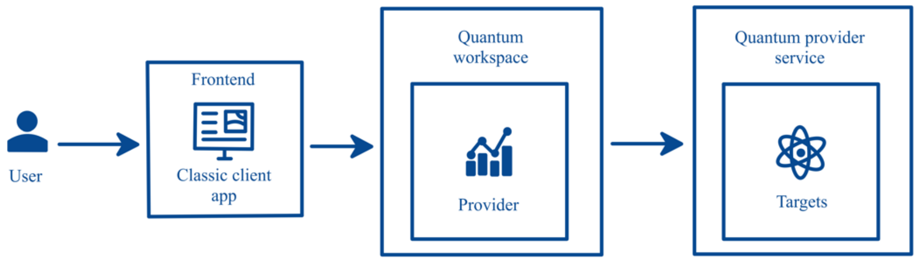

Figure 1: Architecture of a hybrid application

2.1 Python language for hybrid software

The most widely used language for working on quantum computing, at the time of the development of this article, is Python [Rossum, 09]. Table 1 shows the different advantages of this language [Rivas, 21], which allow it to be selected for the development of hybrid software.

Advantage	Description
High-level language	Python is a high-level language, which makes it easy to read, write and learn. In addition, it is an interpreted language, so it is executed directly without the need to compile it beforehand.
Multipurpose and multiparadigm	Thanks to its wide possibilities it is used in many fields (big data, AI, machine learning, etc.) and for different purposes.

[1] https://azure.microsoft.com/en-us/solutions/quantum-computing/
[2] https://aws.amazon.com/es/quantum-solutions-lab/
[3] https://cambridgequantum.com/
[4] https://www.dwavesys.com/

Useful libraries and frameworks	One of the most decisive Python skills for your choice in this area is the creation of open-source libraries. The different quantum languages correspond to different Python libraries.
High-quality syntax	Python's style rules make it much easier to read; for example, line indentations make it easier to identify the different blocks of code.
Object-oriented	Python, being an object-oriented language, allows modeling in terms of classes and objects, offering polymorphism, abstraction, cohesion, etc. It is used to reduce the size and complexity of the code, facilitating possible subsequent modifications.
Portability and cross-platform	Python is available on any operating system, be it Windows, macOS, Linux, etc. Since it is an interpreted language, it can be executed on any type of system that contains its interpreter.
Open source	It is a totally free language, so all users can use it to develop programs and distribute it freely.
Fast learning	It is a very easy language to learn and understand, since it allows you to write programs in very few lines of code.
Community	Python is supported by a large and strong community, which can contribute to its progress.

Table 1: Benefits of Python

2.2 Python extensions for hybrid software

Quantum algorithms must be implemented using programming languages. Python contains all the necessary libraries for their proper development, thanks to its ability to create open-source libraries. And although not all of them are at the same level, there are quite a few Python extensions for hybrid system programming. The most important ones are listed below:

- **Qiskit** [Qiskit, 23]. It is the Python library created by IBM for the development of quantum computing. Currently, it is the most consolidated and the one with the largest community, which is why this article, and the measurement environment are focused on it.
- **Q#**[5]. The 'qsharp' Python package, created by Microsoft, includes the Q# kernel and all the necessary functions to compile and simulate its operations from a normal Python program.
- **Cirq**[6]. is a Python software library created by Google to write, manipulate and optimize quantum circuits and then run them on quantum computers and simulators.
- **PennyLane**[7]. It is a cross-platform Python library, created by Xanadu, for differentiable programming of quantum computers. It is designed to be

[5] https://learn.microsoft.com/en-us/azure/quantum/overview-what-is-qsharp-and-qdk
[6] https://quantumai.google/cirq/
[7] https://pennylane.ai/

hardware independent, allowing quantum functions to be sent to quantum devices such as Qiskit or Braket.

- **Boto3**[8]. It is the package created by Amazon Web Service (AWS) that implements all the tools and modules needed to develop hybrid code using Python.
- **ProjectQ** [Steiger, 18]. It is an open-source software framework for quantum computing initiated by ETH Zurich. It allows users to implement their quantum programs in Python using a powerful and intuitive syntax.
- **Pytket** [Hedfords, 12]. A Python module that allows researchers, algorithm designers and software developers to build and run quantum circuits that produce the best results on the most advanced quantum devices available through the TKET toolkit.

On Table 2, a comparison of the most general characteristics of each library is presented.

Features	Qiskit	Q#	Cirq	PennyLane	Boto3	ProjectQ
Programming language	Python	C#	Python	Python	Python	Python
Compatible hardware	Yes	Yes	Yes	Yes	No	Yes
Hardware simulation	Yes	Yes	Yes	Yes	No	Yes
Circuit optimization	Yes	No	Yes	Yes	No	Yes
Circuit compilation	Yes	Yes	Yes	Yes	No	Yes
QPU compatibility	Yes	Yes	Yes	No	No	Yes
GPU compatibility	Yes	No	Yes	Yes	No	Yes
TPU compatibility	Yes	No	No	No	No	No
Number of qubits	30	60	53	No limit	N/A	32
Community	Big	Small	Small	Big	Big	Small

Table 2: Comparison of the most general characteristics of each library

2.3 Qiskit

Of the previous extensions, it can be said that Qiskit is the one with the largest community today, which has managed to make it the most widely used extension for writing quantum algorithms. For this very reason, the measurement environment developed in this work focuses on the Qiskit library.

[8] https://aws.amazon.com/es/sdk-for-python/

In March 2017, IBM, the American multinational technology and consulting company, creates Qiskit, a free software that allows collaborative work, although its first stable version is not released until December 2019. Python, together with Qiskit, aims to create and manipulate quantum programs and run them on simulators, such as the well-known IBM Quantum Experience[9].

3 Software quality assessment

In order to carry out software quality assessments, it is necessary to have an environment consisting of a quality model, an assessment process and a set of support tools. The ISO/IEC 25010 [ISO, 11] standard defines a software product quality model composed of eight quality characteristics as shown in Figure 2.

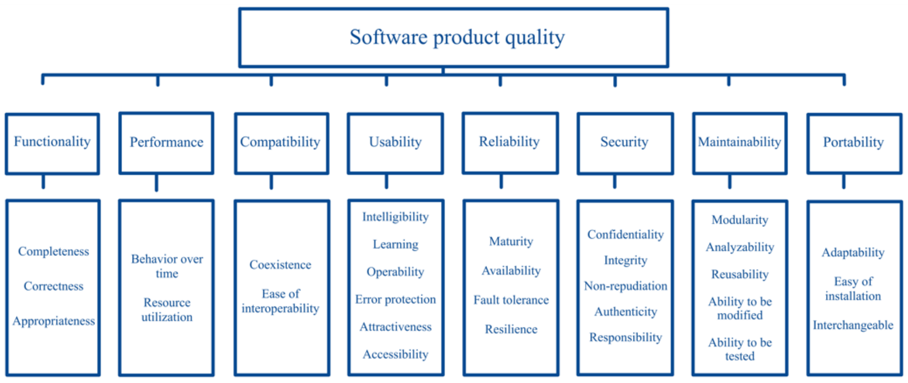

Figure 2: Software product quality model according to ISO/IEC 25010

Of the quality characteristics presented above, this study focuses on the characteristic of maintainability, considering that it is one of the most important aspects to take into account for a technology that is in its infancy [Rodríguez, 15]. In this way it will be possible to ensure that the hybrid software that is started to be developed now, can be evolved efficiently in the future, since as we know the maintenance phase is one of the most expensive in the software life cycle, if not the most, reaching in some cases 60% of the total effort [Glass, 02].

The main metrics used for the evaluation of quality according to its maintainability in classic software are the following [Rodríguez, 14]:

M1: Non-compliance with encoding standards rules.
M2: Code documentation.
M3: Complexity.
M4: Structuring of classes, packages and files.
M5: Size of methods.
M6: File size.
M7: Duplicate code.

[9] https://quantum-computing.ibm.com/

M8: Dependency cycles.

The above metrics are, in principle, independent of the programming language. In this article, the analysis of these metrics for the Python language and their application and adaptation for use with hybrid software is performed.

This work aims to extend the SonarQube environment focused on classical software, so that it can also be used for the evaluation of hybrid software.

3.1 Tools for assessing 'classic' quality metrics

In this section, we analyze the main tools that perform measurements on some of the metrics mentioned above. It should be noted that these tools are those compatible with Python, due to the benefits of this language and the fact that the extensions for quantum computing from manufacturers such as IBM or Google are based on it.

Some tools dedicated to measuring and evaluating metrics of code maintainability are the following.

- **SonarQube**[10]. It is an automatic code review tool to detect bugs, vulnerabilities and code smells in your code.
- **Pycodestyle** [Rocholl, 23]. It is an automatic analyzer of a system's Python scripts, which points out specific areas where the code can be improved.
- **Flake8** [Ziade, 23]. This is a great set of tools for checking your source code against PEP8, programming errors, and for checking cyclomatic complexity.
- **Pylint** [Python, 23]. This is a Python static code analysis tool that looks for programming errors, helps enforce a coding standard, detects code smells, and offers simple refactoring suggestions.
- **Prospector** [Crowder, 23]. Is a static analysis tool for analyzing Python code and displaying information about bugs, potential problems, convention violations and complexity.
- **Radon** [Lacchia, 23]. It is a Python tool that calculates several code metrics, such as cyclomatic complexity, comment lines or maintainability index.
- **Pygenie** [Gift, 10]. This is a tool for evaluating the cyclomatic complexity of a Python script.

Table 3 shows the results of the study of the existing tools for each of the metrics:

Metric	Tools
Non-compliance with rules	SonarQube, Pycodestyle, Flake8, Pylint and Prospector
Code documentation	SonarQube y Radon
Complexity	SonarQube, Pylint, Prospector, Radon and Pygenie
Class structuring	SonarQube
Package structuring	SonarQube
File structuring	SonarQube

[10] https://www.sonarsource.com/products/sonarqube/

Size of methods	-
File size	SonarQube
Duplicate code	SonarQube y Pylint
Dependency cycles	-

Table 3: Tools that evaluate maintainability metrics on Python

As can be seen in Table 3, the tool that performs the largest number of code maintainability metrics measurements is SonarQube. Sonar Source is a company dedicated to software development for quality and security, which is composed of three analyzers: SonarLint[11], SonarCloud[12] and SonarQube, the latter being the leading tool in the market for continuous analysis of code quality.

3.2 SonarQube for evaluating classical quality metrics

SonarQube is a free software platform used to automatically manage the quality of source code, performing a static analysis of it through the hundreds of rules it integrates, in order to warn about possible improvements in quality and security [Guaman, 17]. Thanks to tools such as Sonar, cleaner and safer code is developed.

Static code analysis consists of evaluating the software without having the need to execute it, which provides some advantages over dynamic analysis. One of these advantages is that static analysis operates on all possible execution branches of a program; while dynamic analysis only has access to the code paths that are executing at the time of evaluation [Thomson, 21].

SonarQube supports the analysis of more than twenty programming languages, although this article focuses on the use of this tool to analyze metrics on projects developed in Python, as previously mentioned. In addition, Sonar imports several static code analyzers that help it to measure specific metrics, such as Flake8 to evaluate the rules linked to the style guide or to measure cyclomatic complexity.

In the following, each of the quality metrics mentioned above are analyzed to check their evaluation using the selected tool, SonarQube.

M1: Non-compliance with encoding standards rules

All languages have a programming standard, being PEP8 the most used style guide and the one recommended by IBM in the case of Python [Rossum, 01]. SonarQube generates Issues when the rules set by the Standard Library are violated.

M2: Code documentation

This metric refers to the existing comments in the code of a program, which are used to explain the different functionalities of the program. SonarQube calculates the

[11] https://www.sonarsource.com/products/sonarlint/
[12] https://www.sonarsource.com/products/sonarcloud/

number of comment lines in each file and directory of an analyzed Project, as well as the density of these comments as a percentage.

M3: Complexity

SonarQube raises issues of both cognitive complexity and cyclomatic complexity in code that is structured in a way that is complicated to understand. The high probability of errors appearing in complex code has been demonstrated, and confusing code could lead maintainers to add even more. The advantage of Sonar is that it calculates such values at the file level as well as at the method and function level, thus avoiding possible masking.

M4.1: Class structuring

This metric indicates how well the classes of a project are organized. To do so, it is necessary to measure the number of methods found in each of the classes. In fact, SonarQube does not indicate the number of methods or functions for each of the classes of the evaluated project, but it does indicate the number of functions found in each file. For object-oriented languages, the number of methods per file and the number of methods per class are exactly the same; therefore, SonarQube obtains the measurement of this metric for Python, since it is an object-oriented language.

M4.2: Package structuring

This property indicates the level of organization of the packages by evaluating the distribution of classes among the system packages. Sonar shows the number of classes located on each package evaluated within the Project to be analyzed.

M4.3: File structuring

File structuring indicates how well the files are organized in the system. For this purpose, the distribution of the functions or methods among the different files that make up the application is checked. SonarQube displays the results once the calculation has been performed.

M5: Size of methods

This metric is responsible for measuring the number of significant lines of code in each of the methods or functions of a program, i.e., without taking into account the comment lines or blank lines. Sonar, unfortunately, does not perform the calculation of this property.

M6: File size

In this case, the size evaluation is performed at the file level, based on the number of lines of code that are significant in each file of the Project to be analyzed. Sonar provides the result for this metric without any problem.

M7: Duplicate code

This property refers to code fragments that are duplicated. The duplicated code complicates the maintainability too much, since if you want to make improvements in this code, you must modify all the parts of the system where the repeated code appears. Sonar provides this data at the class level or at the general system level. In addition, Sonar calculates from the exact duplicated lines and blocks to the overall density of the duplicated code.

M8: Dependency cycles

It refers to the existence of dependency cycles between the system's packages. When you want to maintain the code, the cycles will affect negatively making it difficult to make improvements or modifications, since a change in a package will affect all the packages in the cycle of dependencies. SonarQube also does not measure this metric on the projects analyzed in Python language.

4 Metrics for hybrid software

This section presents, on the one hand, the adaptations and considerations to be able to use the previous classical metrics in hybrid software. On the other hand, a new set of metrics applicable only to this new quantum software is presented.

4.1 Classic metrics applicable to hybrid software

The metrics studied are those explained in the previous chapter, those applicable to classical code, trying to conclude this chapter with the most appropriate metrics to measure the quality of hybrid code. To study the applicability of each of them, the Python and Qiskit languages are taken into account, the latter being the most developed quantum language so far.

It is necessary to spend time investigating the metrics applicable to quantum code, since they are the ones that serve as the basis for the measurement environment of the developed hybrid software that we will discuss later. They are studied individually below:

M1: Non-compliance with encoding standards rules

One of the fundamental pillars of Python is its Standard Library, which contains the rules that must be followed to develop code using this language. Just as this standard is applied when writing classical code, it should also be applied to the development of quantum code to improve its productivity.

Tools such as Pylint and Pycodestyle are used in order to enforce a consistent code style in the project. Both tools are used to enforce the style rules corresponding to the PEP8 standard [Rossum, 2001], the most widely used standard for Python code development.

One of the fundamental rules of the Python standard, of which its analysis is necessary, gives rise to the following property:

- **Imports**

This property consists of analyzing the imports performed, checking that all of them are in the first cell of the Notebook, since the development of quantum algorithms is usually performed on Jupyter Notebook, where the scripts are divided into cells. If so, the percentage of imports found in the first cell would be 100%. This eases the complexity of code structuring, related to code maintainability.

The base metrics to be calculated are shown in Table 4:

Metric	Description
N_IFC	Number of imports made in the first cell
N_TI	Number of total imports made throughout the code

Table 4: Base metrics for 'Imports'

The metric derived from the division of the two previous metrics is shown in Table 5:

Metric	Description
% IFC	Percentage of imports made in the first cell % IFC = (N_IFC / N_TI) · 100

Table 5: Derived metric for 'Imports'

M2: Code documentation

This property refers to the number of comments defined throughout the code, which are necessary to explain the functionality of the code.

This property is measured based on the four base metrics shown in Table 6:

Metric	Description
N_Com	Number of comment lines
N_TL	Total number of lines, taking into account both comment lines and source code lines

Table 6: Base metrics for 'Documentation'

Starting from the base metrics, the derived metrics shown in Table 7 can be calculated:

Metric	Description
% Com	Percentage of comments % Com = (N_Com / N_TL) · 100

Table 7: Derived metrics for 'Documentation'

M3: Complexity

This property is related to the code's ability to be modified, its testability and its analyzability. Although it can be measured in many ways, the cyclomatic complexity metric is the most widespread.

Analyzing Python and Qiskit, three different ways of adding forks throughout the code are located, thus generating changes in the result and cyclomatic complexity.

- ***If* instructions**

Qiskit has a conditional if instruction, which allows to apply operations on qubits depending on the state in which a classical bit is at a given time.

To facilitate the understanding of its use in the source code, the example of Algorithm 1 is shown.

```
qc = QuantumCircuit(q,c)
qc.x(q[0].c_if(c,0))
qc.measure(q,c)
```

Algorithm 1: If instruction in Qiskit code

In these lines of code, the X operation will be applied on the 0 qubit, depending on the value taken by the classical bit c. If the value of the classical register, interpreted as a binary number, corresponds to 0, then the state of the 0 qubit will be inverted.

- **Controlled operations**

Qiskit has controlled operations to apply a gate to a qubit depending on the state of another qubit [Kumar, 23]. For example, one may want to change the state of a second qubit when the state of the first qubit, called the control qubit, is at $|0\rangle$.

- **Conditional *if* statement of the Python language**

Given the use of Python in hybrid code, the conditional statements of this language can be used. They can also be used in nested form.

Having seen the different ways to increase the cyclomatic complexity of the Qiskit code, the new applicable base metrics are shown in Table 8:

Metric	Description
N_ifI	Number of *if* instructions
N_CO	Number of controlled operations
N_ifS	Number of conditional *if* statements

Table 8: Base metrics for 'Cyclomatic complexity'

Starting from three of the base metrics, the derived metrics shown in Table 9 can be calculated:

Metric	Description
N_CCO	Number of operations that increase cyclomatic complexity N_CCO = N_Oif + N_CO + N_ifS

Table 9: Derived metric for 'Cyclomatic complexity'

M4.1: Class structuring

This property is in charge of measuring the number of functions defined in each of the classes, which is equivalent to the number of functions defined in each file.

Therefore, the metric shown in Table 10 is defined:

Metric	Description
N_FC	Number of functions defined in a class or file

Table 10: Metrics for 'Class Structuring'

M4.2: Package structuring

This quality property is in charge of analyzing the number of classes in each of the packages, but in Qiskit it is not necessary to calculate it because the algorithms are developed within the same package.

M4.3: File structuring

Since Qiskit continues to be developed together with the object-oriented Python language, the metric defined in the quality property called 'class structuring' (FC No.) is also used to analyze the structuring of the different files.

M5: Method size

This property measures the number of lines that are not comments in the methods, so it is applicable to Qiskit facilitating the analysis of the number of lines of code located in the methods.

The metric that this property analyzes is shown in Table 11:

Metric	Description
N_ML	Number of lines of code contained in one method

Table 11: Metric for 'Method size'

M6: File size

This property performs size evaluation at the file level, based on the measurement of the number of lines of code that are not comments. It is applicable to hybrid code developed using Qiskit in the same way as it is applicable to classical code.

The metric that this property analyzes is shown in Table 12:

Metric	Description
N_LF	Number of lines of code contained in a file

Table 12: Metric for 'File size'

M7: Duplicate code

Duplicate code fragments make it difficult to maintain the Qiskit library code as in Python. The code with duplicated fragments ends up being much longer, without the appropriate abstraction levels and with a higher number of errors, so it is important to measure it also in the hybrid code.

The base metrics that help to calculate this quality property are shown in Table 13:

Metric	Description
N_DL	Number of duplicated lines of code
N_TL	Total number of lines, taking into account both comment lines and source code lines

Table 13: Base metrics for 'Duplicate code'

Starting from the base metrics, the derived metrics located in Table 14 can be calculated:

Metric	Description
% DC	Percentage of duplicated lines of code % DC = (N_DL / N_TL) · 100

Table 14: Derived metric for 'Duplicate code'

M8: Dependency cycles

Hybrid code using Qiskit is always developed within the same package, so no dependency cycles can arise. This metric does not apply to hybrid code developed using the Qiskit library.

4.2 New metrics applicable to hybrid software

Once the classical maintainability metrics and their support in hybrid code using Python and Qiskit have been analyzed, it is necessary to define new metrics to produce software with adequate quality and productivity. The code must be sufficiently flexible and understandable, making it easy to make modifications to reflect changes. It is clear that the easier the code is to understand, the easier it will be to maintain. In addition, it is necessary for quantum designers and programmers to properly understand the quantum foundations to design high quality code.

For the analysis of these metrics at the code level, the metrics already defined for circuits, and represented in [Cruz-Lemus, 21], are taken into account.

M1: Circuit size

The larger the code, the more complex it will be to understand and maintain; therefore, for the study of this quality property it is necessary to analyze the metrics shown in Table 15.

Metric	Description
Width	Total number of qubits created
Depth	Maximum number of doors applied on a qubit

Table 15. Metrics for 'Code size'

M2: Complexity of gates operations

The greater the number of gates applied to the qubits, the more complex the code will end up being to understand.

For the study of this quality property, it is convenient to measure the base metrics shown in Table 16.

Metric	Description
N_NOT	Number of gates X
N_Y	Number of gates Y
N_Z	Number of gates Z
N_I	Number of gates Identity
N_U	Number of gates U
N_P	Number of gates P
N_H	Number of gates Hadamard
N_S	Number of gates S
N_IS	Number of gate inverses S
N_T	Number of gates T
N_IT	Number of gate inverses T
N_RX	Number of gates RX
N_RY	Number of gates RY
N_RZ	Number of gates RZ
N_CNOT	Number of gates CX
N_CY	Number of gates CY
N_CZ	Number of gates CZ
N_CU	Number of gates CU
N_CH	Number of gates CH
N_CRZ	Number of gates CRZ
N_CP	Number of gates CP

N_CU3	Number of gates CU3
N_SWAP	Number of gates SWAP
N_Toff	Number of gates Toffoli
N_Fred	Number of gates Fredkin

Table 16: Base metrics for 'Complexity of gate operations'

Once the base metrics have been analyzed, the following derived metrics can be calculated, as shown in Table 17.

Metric	**Description**
N_TG	Total number of gates N_TG = N_NOT + N_Y + N_Z + N_I + N_U + N_P + N_H + N_S + N_IS + N_T + N_IT + N_RX + N_RY + N_RZ + N_CNOT + N_CY + N_CZ + N_CU + N_CH + N_CRZ + N_CP + N_CU3 + N_SWAP + N_Toff + N_Fred
N_GP	Number of gates Pauli N_GP = N_NOT + N_Y + N_Z
N_GCliff	Number of gates Clifford N_GCliff = N_H + N_S + N_IS
N_GC3	Number of gates C3 N_GC3 = N_T + N_IT
N_SRG	Number of standard rotating gates N_SRG = N_RX + N_RY + N_RZ
N_GSQ	Number of gates of a single qubit N_GSQ = N_GP + N_GCliff + N_GC3 + N_SRG + N_U + N_P + N_I
% GSQ	Percentage of gates with a single qubit %SGQ = (N_GSQ / N_TG) · 100
N_CPG	Number of controlled Pauli gates N_CPG = N_CNOT + N_CY + N_CZ
N_G2Q	Number of gates of two qubits N_G2Q = N_CPG + N_CU + N_CH + N_CRZ + N_CP + N_CU3 + N_SWAP
N_G3Q	Number of gates of three qubits N_G3Q = N_Toff + N_Fred
N_MQG	Number of multi qubit gates N_MQG = N_G2Q + N_G3Q
% MQG	Percentage of multi qubit gates % MQG = (N_MQG / N_TG) · 100

Table 17: Derived metrics for 'Complexity of gate operations'

M3: Number of measurement operations

The greater the number of measurements performed along the quantum code, the easier it will be to understand it, as well as to detect errors. Table 18 shows the quality metrics required for the calculation of this property.

Metric	Description
N_QM	Number of qubits with any Measure operation
% QM	Percentage of qubits measured % QM = (N_QM / Width) · 100

Table 18: Metrics for 'Number of measurements'

M4: Number of initialization and restart operations

This property analyzes the number of initialization operations, which initialize a flexible number of qubits to an arbitrary state, and the number of reset operations, which are responsible for sending the qubits to the $|0\rangle$ state in the middle of a given computation. Both operations are not gates, since they are not unit operations and, therefore, are not reversible.

The metrics defined for the calculation of this property are shown in Table 19:

Metric	Description
N_QReset	Number of qubits with any Reset operation
N_QInit	Number of qubits with any Initialize operation
% QReset	Percentage of qubits with any Reset operation % QReset = (N_QReset / Width) · 100
% QInit	Percentage of qubits with any Initialize operation % QInit = (N_QInit / Width) · 100

Table 19: Metrics for 'Number of initialization and restart operations'

M5: Number of auxiliary qubits

There are auxiliary bits, called Ancilla, which are used to achieve some specific goals in computation, especially in cases of reversibility. The declaration of an excessive number of such bits may end up resulting in a more complex code.

The metric defined for this quality property is shown in Table 20:

Metric	Description
N_Anc	Number of auxiliary qubits
% Anc	Percentage of auxiliary qubits % Anc = (N_Anc / Width) · 100

Table 20: Metrics for 'Number of auxiliary qubits'

5 Measurement environment for hybrid software

As seen before, SonarQube already offers different measurements for classic software about quality metrics focused on code maintainability. Another advantage of this tool is the possibility of implementing a plugin to adapt the measurements and customize the results. For this reason, the environment that has been developed to perform the classical-quantum code analysis is a plugin for SonarQube, which is compatible with Python and Qiskit.

The developed plugin becomes part of the Sonar environment itself, thus taking advantage of its scanner when performing project evaluations. Therefore, the input files to the tool are Python scripts, while the output of the evaluation by the tool will be the measurement results for both classical and quantum metrics (Figure 3).

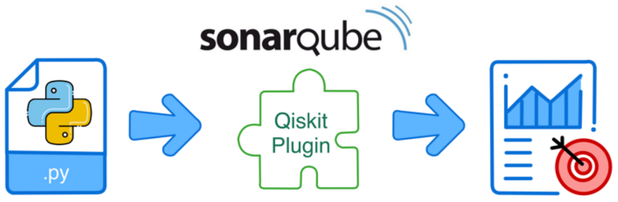

Figure 3: Hybrid code metrics analysis process

To demonstrate the usefulness of the measurement plugin of the developed hybrid software, one of the analyses performed using the new measurement environment is shown below. In this case, the analysis of Shor's algorithm [Shor, 97] is performed, which is famous for refactoring integers in polynomial time, allowing the factorization of sufficiently large integers.

Its representation in a quantum circuit is shown in Figure 4:

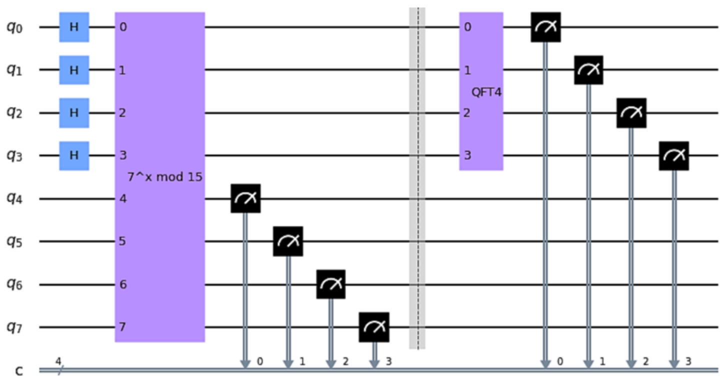

Figure 4: Shor's Algorithm Circuit

And the equivalent code to the previous circuit is shown in Algorithm 2:

```
def qft():

        """ Quantum Fourier transform """

        qreg_q = QuantumRegister(4, 'q')
        circuit = QuantumCircuit(qreg_q)

        circuit.h(qreg_q[3])
        circuit.cp(pi/8, qreg_q[0], qreg_q[3])
        circuit.cp(pi/4, qreg_q[1], qreg_q[3])
        circuit.cp(pi/2, qreg_q[2], qreg_q[3])
        circuit.h(qreg_q[2])
        circuit.cp(pi/4, qreg_q[0], qreg_q[2])
        circuit.cp(pi/2, qreg_q[1], qreg_q[2])
        circuit.h(qreg_q[1])
        circuit.cp(pi/2, qreg_q[0], qreg_q[1])
        circuit.h(qreg_q[0])
        circuit.swap(qreg_q[1], qreg_q[2])
        circuit.swap(qreg_q[0], qreg_q[3])

        gate = circuit.to_gate()
        gate.QFT = "QFT gate"
        return gate

def _7mod15()

        """ Take x and return (7^x)mod15 """

        qreg_q = QuantumRegister(8, 'q')
        circuit = QuantumCircuit(qreg_q)

        circuit.x(qreg_q[0])
        circuit.x(qreg_q[1])
        circuit.x(qreg_q[2])
        circuit.x(qreg_q[3])
        circuit.cx(qreg_q[0], qreg_q[5])
        circuit.cx(qreg_q[0], qreg_q[6])
        circuit.cx(qreg_q[1], qreg_q[4])
        circuit.cx(qreg_q[1], qreg_q[6])
        circuit.ccx(qreg_q[0], qreg_q[1], qreg_q[4])
        circuit.ccx(qreg_q[0], qreg_q[1], qreg_q[5])
        circuit.ccx(qreg_q[0], qreg_q[1], qreg_q[6])
        circuit.ccx(qreg_q[0], qreg_q[1], qreg_q[7])

        gate = circuit.to_gate()
        gate.QFT = "7 mod 15"
        return gate

qreg_q = QuantumRegister(8, 'q')
creg_c = ClassicalRegister(4, 'c')
circuit = QuantumCircuit(qreg_q, creg_c)

circuit.h (qreg_q[0])
circuit.h(qreg_q[1])
circuit.h(qreg_q[2])
circuit.h(qreg_q[3])
circuit.append(_7mod15(), range(8))
circuit.barrier(range(8))
circuit.append(qft(), range(4))
circuit.measure(range(4), range(4))
circuit.draw()
```

Algorithm 2: Shor's algorithm

Once the analysis of the algorithm has been performed, the tool offers the following results shown in Table 21, extracted from the SonarQube interface extended with the developed plugin. By analyzing them, the veracity of the measurements performed can be checked; as, for example, in the case of the qubits that have been initialized, which are eight, and among which a maximum of seven quantum gates have been applied. Another check that can be made is the number of controlled gates applied, which turn out to be 50% of the total.

Code size	
Width	8
Depth	7
Conditionals operations	
N_CO	16
N_CCO	16
Measurements	
N_QM	4
% QM	50

Base gates	
N_NOT	4
N_H	8
N_CNOT	4
N_CP	6
N_SWAP	2
N_Toff	4
Derived gates	
N_TG	28
N_GP	4
N_GCliff	8
N_GSQ	12
% GSQ	42,9
N_CPG	4
N_G2Q	12
N_G3Q	4
N_MQG	16
% MQG	57,1

Table 21: Results obtained for the metrics after the measurement carried out on Shor's algorithm

It can also be commented that, of the 28 gates applied in total, 57.1% of them are multi-qubit; that is, they modify the state of more than one qubit through their application. Meanwhile, the other 42.9% are gates applied to a single qubit.

Finally, the number of gates of each type can be checked, such as the four X-controlled gates, the eight controlled gates or the other four Toffoli gates; as well as the number and percentage of qubits that have a measurement operation.

6 Conclusions

Quantum software engineering is necessary to deal with challenges related to producing quantum software in a systematic manner and with sufficient quality levels [Piattini, 21]. One of these challenges is how to produce hybrid software that could be easily maintained and so that could evolve as the quantum software technology improves and consolidates.

In this paper we propose a novel environment to assess the complexity of hybrid software through the defined metrics. However, the presented environment is only a first effort in the line of measurement and evaluation of the quality of hybrid software and currently several lines of work are open to allow not only to measure, but also to give an evaluation of the quality. Among these lines we can highlight:

- **Define quality thresholds for quantum metrics**. Certain values for the thresholds of quantum metrics must be defined, but -due to the youth of quantum computing- there are not yet sufficient theories to easily obtain these values.
- **Diagnose the quality of the hybrid code through the environment**. The plug-in developed for SonarQube performs measurements of quantum metrics on the code, but a diagnosis of its quality once an analysis has been performed is still pending.
- **Extend the work to other quantum languages**. The currently implemented environment is compatible with the Qiskit language developed by IBM, but the development of an environment compatible with more quantum languages, such as Q# or Cirq, is pending for the future.

Acknowledgments

This research was supported by QSERV: Quantum Service Engineering: Development Quality, Testing & Security of Quantum Microservices (PID2021-124054OB-C32) projects funded by the Spanish Ministry of Science and Innovation and ERDF; and Q2SM: Quality Quantum Software Model (EXPTE: 13/22/IN/032) financed by the Junta de Comunidades de Castilla-La Mancha and FEDER funds.

References

[Bernhardt, 20] Bernhardt, C.: Quantum Computing for Everyone. The MIT Press, 2020, doi:10.5555/3351840

[Crowder, 23] Crowder, C.: Prospetor is a tool to analyse Python code by aggregating the result of other tools, 2023, Retrieved from https://pypi.org/project/prospector/

[Cruz-Lemus, 21] Cruz-Lemus, J.A., Marcelo, L.A., Piattini, M.: Towards a set of metrics for quantum circuits understandability. International Conference of the Quality of Information and Communications Technology (QUATIC 2021). 1439, pp. 233-249. Algarve, Portugal: Communications in Computer and Information Science. doi:10.1007/978-3-030-85347-1_18

[Gift, 10] Gift, N.: Writing clean, testable, high quality code in Python, 2010, Retrieved from https://developer.ibm.com/articles/au-cleancode/

[Google, 23] Google: Google, 2023, Retrieved from https://www.google.es/

[Guaman, 17] Guaman, D., Sarmiento, P.A-Q., Barba-Guamán, L., Cabrera, P., Enciso, L.: SonarQube as a Tool to Identify Software Metrics and Technical Debt in the Source Code through Static Analysis. Proceedings of 2017 the 7th International Workshop on Computer Science and Engineering, 2017, (pp. 171-175). Beijing, China. doi:10.18178/wcse.2017.06.030

[Hedfords, 12] Hedfords, K.: pyTickets are light-weight symmetrically signed data containers with optional encryption, serialization and compression of their contents, 2012, Retrieved from https://pypi.org/project/pytickets/

[IBM, 23] IBM: IBM, 2023, Retrieved from https://www.ibm.com/

[ISO, 11] ISO: ISO/IEC 25010. Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - Guide to SQuaRE. Retrieved from Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models, 2011.

[ISO, 14] ISO: ISO/IEC 25000. Systems and software engineering -- Systems and software Quality Requirements and Evaluation (SQuaRE) -- Guide to SquaRE, 2014.

[Kumar, 23] Kumar, A.: Formalization of Structural Test Cases Coverage Criteria for Quantum Software Testing. International Journal of Theoretical Physics, 2023, 62, 1-16. doi:10.1007/s10773-022-05271-y

[Lacchia, 23] Lacchia, M.: Radon: Code Metrics in Python, 2023, Retrieved from https://pypi.org/project/radon/

[Pettersen, 21] Pettersen, G., Rubio, A.: Una tecnología verdaderamente disruptiva que va a cambiar el mundo. Sesión Magistral de Computación Cuántica II. Barcelona: AUSAPE, 2021.

[Piattini, 20] Piattini, M., Peterssen, G., Pérez-Castillo, R., Hevia, J.L., Serrano, M.A., Hernández, G., García, I., Paradela, C.A., Polo. M., Murina, E., Jiménez, L., Marqueño J.C., Gallego, R., Tura, J., Phillipson, F., Murillo, J.M., Niño, A., Rodríguez, M.: The Talavera Manifesto for Quantum Software Ingenieering and Programming. QANSWER: QuANtum SoftWare Engineering & pRogramming, 2020, Talavera de la Reina.

[Piattini, 21] Piattini, M., Serrano, M., Perez-Castillo, R., Petersen, G., Hevia, J.L.: Toward a quantum software engineering. IT Professional, 2021, 23(1), 62-66. doi:10.1109/MITP.2020.3019522

[Python, 23] Python Code Quality Authority: Pylint: python code static checker, 2023, Retrieved from https://pypi.org/project/pylint/

[Qiskit, 23] Qiskit Contributors: Qiskit: An Open-source Framework for Quantum Computing, 2023, doi:10.5281/zenodo.2573505

[Rivas, 21] Rivas: Ventajas de Python, 2021.

[Rocholl, 23] Rocholl, J.C., Lee, I.: Python style guide checker, 2023, Retrieved from https://pypi.org/project/pycodestyle/

[Rodríguez, 14] Rodríguez, M., Piattini, M.: Software Product Quality Evaluation Using ISO/IEC 25000. ERCIM News, 2014 (99). Retrieved from http://ercim-news.ercim.eu/en99/special/software-product-quality-evaluation-using-iso-iec-25000

[Rodríguez, 15] Rodríguez, M., Piattini, M., Fernández, C.M.: A hard look at software quality: Pilot program uses ISO/IEC 25000 family to evaluate, improve and certify software products. Quality Progress, 2015, 48(9), 30-36.

[Rodríguez, 19] Rodríguez, M., Piattini, M., Ebert, C.: Software Verification and Validation Technologies and Tools. IEEE Software, 2019, 36(2), 13-24. doi:10.1109/MS.2018.2883354

[Rossum, 01] Rossum, G., Warsaw, B., Coghlan, N.: Style Guide for Python Code, 2001, Retrieved from https://www.python.org/dev/peps/pep-0008/

[Rossum, 09] Rossum, G., Drake, F.L.: Python 3 Reference Manual. Scotts Valley, CA: CreateSpace, 2009, doi:10.5555/1593511

[Shor, 97] Shor, P.W.: Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. SIAM Journal on Computing, 1997, 26(5), 1484-1509. doi:10.1137/S0097539795293172

[Steiger, 18] Steiger, D.S., Häner, T., Troyer, M.: ProjectQ: an open source software framework for quantum computing. Quantum, 2018, 2, 49. doi:10.22331/q-2018-01-31-49

[Thomson, 21] Thomson, P.: Static Analysis. Commun. ACM, 2021, 65(1), 50-54. doi:10.1145/3486592

[TS2, 23] TS2: Explorando el Futuro de la Computación Cuántica Híbrida: Oportunidades y Desafíos, 2023, Retrieved from https://ts2.space/es/explorando-el-futuro-de-la-computacion-cuantica-hibrida-oportunidades-y-desafios/

[Ziade, 23] Ziade, T., Cordasco, I.S.: Flake8: the modular source code checker, 2023, Retrieved from https://pypi.org/project/flake8/