applied sciences

MDPI

Article

On the Definition of Quantum Programming Modules

Pedro Sánchez * and Diego Alonso

Department of Information and Communications Technologies, Universidad Politécnica de Cartagena, 30201 Cartagena, Spain; diego.alonso@upct.es

* Correspondence: pedro.sanchez@upct.es; Tel.: +34-968-326-460

Abstract: There are no doubts that quantum programming and, in general, quantum computing, is one of the most promising areas within computer science and one of the areas where most expectations are being placed in recent years. Although the days when reliable and affordable quantum computers will be available is still a long way off, the explosion of programming languages for quantum programming has grown exponentially in recent years. The software engineering community has been quick to react to the need to adopt and adapt well-known tools and methods for software development, and for the design of new ones tailored to this new programming paradigm. However, many key aspects for its success depend on the establishment of an appropriate conceptual framework for the conception and design of quantum programs. This article discusses the concept of module, key in the software engineering discipline, and establishes initial criteria for determining the cohesion and coupling levels of a module in the field of quantum programming as a first step towards a sound quantum software engineering. As detailed in the article, the conceptual differences between classical and quantum computing are so pronounced that the translation of classical concepts to the new programming approach is not straightforward.

Keywords: software engineering; modularity; cohesion; design; quantum programming

Citation: Sánchez, P.; Alonso, D. On the Definition of Quantum Programming Modules. *Appl. Sci.* **2021**, *11*, 5843. https://doi.org/10.3390/app11135843

Academic Editor: Mario Piattini

Received: 23 May 2021
Accepted: 21 June 2021
Published: 23 June 2021

1. Introduction

Designing quantum programs (QP) is quite different from the way in which it is performed in classical computing, basically because our intuition is anchored in radically different mathematical models and computer architectures [1]. The (surprising) laws of quantum mechanics, which underpin quantum computers, allow us to formulate and solve problems in a new and radically different way from what we are used to in our macroscopic world. To harness the computational power offered by quantum computers, we, as programmers, have to change the way we conceive solutions, at least in terms of their design. At the same time, it only makes sense to resort to the quantum solution if it is better, from a computational point of view, than any solution available in the "classical" model. In other words, it would be of little use to have a quantum algorithm to solve a problem that could be solved with a conventional computer with the same order of complexity. This makes the quantum programming paradigm a challenge that becomes even more crucial in terms of what Software Engineering as a discipline can provide [2].

To give support to computation, several quantum architectures [3] and programming languages [2] have been proposed over the last twenty years. Since the first practical quantum programming language QCL was introduced in 1988, many other languages have appeared. They can be categorized into at least four groups [4] attending to the level of abstraction of the primitives they provide: from high level to hardware-specific ones. In all of them, QP are translated into a quantum circuit composed by a set of sequences of quantum gates that manipulate the state of qubits in order to perform a computation [5].

The processing of quantum information is also quite different to classical computing. Quantum computing is based on the phenomenon of quantum mechanics (superposition, entanglement, interference, etc.), whereas in conventional computing the state at a given

time is either "on" or "off" (purely binary) [6]. Moreover, with quantum machines the power of computation increases exponentially with the number of bits, in contrast to the linear proportion of conventional computers. Thus, QP are very well suited for tasks including optimization, analysis, and simulation of huge quantity of data [7]. On the downside, very common operations in classical programming are no longer possible for quantum computers, for instance, to produce perfect copies of unknown quantum states [8]. However, perhaps the main difference with classical programming is the fact that quantum algorithms are designed in a probabilistic way to solve the problems. These and other relevant differences form the perfect challenge for a community of software developers used to a set of concepts and rules that are no longer valid for the new paradigm [9].

Very recently, several authors have highlighted the importance of adopting the software engineering discipline in the field of quantum computing. One of the most representative works is [5], where the term "quantum software engineering" is redefined and introduces the "quantum software life-cycle". Furthermore, the authors identify the main challenges and opportunities for the establishment of the new discipline. The first remark of importance is that quantum software development is much more than only quantum programming. Thus, a new approach is needed to cope with quantum software development. According to that, and in analogy with the classical software engineering approach, quantum software has to be built economically, has to be reliable, and has to work efficiently on real quantum computers by means of using the well-known artifacts, methods, tools, and processes. In [5], the same phases of the classical software development life cycle are imported to the quantum one and are extended to consider the "quantum" aspect: quantum software requirements analysis, quantum software design, quantum software implementation, quantum software testing, and quantum software maintenance. Most of these steps are nowadays waiting for contributions from the software engineering community, as summarized below.

Firstly, to our knowledge, there is absolutely no contribution on requirements engineering. With respect to the design phase, a process to transform requirements into a suitable form so as to make relatively direct the transformation into executable quantum software is needed. Some authors have also given initial steps on the definition of design patterns for quantum software [10]. Other authors are extending UML as a notation to represent (at the same abstraction level as quantum circuits) the sequences of gates implementing QP [11,12].

Testing and maintenance phases are especially complex, and a real challenge given the singularity of quantum hardware and quantum execution limits and constraints [13]. Quantum testing, including quantum verification, is also a matter of research due to the inherent randomness of quantum mechanical systems. The verification of QP is complex in nature due to the underlying obstacles to design them, the probabilistic nature of quantum mechanics, and the current absence of error-free hardware to execute QP. Here, the use of quantum error corrections to emulate noisy-free computers is also of special relevance to researchers [14].

Of great interest is the *Talavera Manifesto for Quantum Software Engineering* [15] as the result of many discussions of academia and industry in 2020. In this manifesto, many principles and commitments about how to adapt software engineering principles in quantum software development have been collected. The main driver is that quantum software engineering can definitely contribute to the quantum computing field after adapting or defining the existing principles and methodologies from the classical software engineering field. The stakeholders who should be involved in this objective are mainly software developers, researchers, educators, customers, and governments, among others.

In software development, there is a long history of decomposing a system into smaller modules. Modularization is the technique for dealing with complex systems. Modules are well established in software engineering, since they were introduced in the 1970s [16] as a way to reduce the complexity of a system by applying the principle "divide and conquer", thus allowing a programmer to approach the development of each module separately

and almost independently of each other. With a judicious distribution of the complete functionality of the system among the modules that comprise it, the design of the system can be greatly simplified. This facilitates enormously the work of the programmer, who can focus on developing only a (small) part of the program's functionality and forget about the rest of the complexity of the system, at least momentarily. A good modular design facilitates not only the development phase, but also the debugging, maintenance, and reuse phases, among other desirable characteristics, of the modules created. Modules are a way to reduce the mental and intellectual load that the programmers of a system must bear. Modules also allow teamwork, in which each programmer or group of programmers is responsible for the development and maintenance of one or more modules. Modules encapsulate a particular functionality and have a well-defined interface that describes how to access their functionality in a pluggable way.

Modularization is also a factor affecting the software quality attributes maintainability and reuse. The former is important as modules are dynamic and require modifications in their lifetime. The latter enables more rapid development of new products or enhancements of existing products, avoiding developing from "scratch" each time, which would increase the development costs.

Not many authors have researched on the definition of module for quantum programming from a software engineering perspective, since most of the research results available come from the quantum mechanics field. One of the most significative works is described in [17], where the authors provided some insights on the definition of a formal framework for introducing modularity in QP. They extracted some important conclusions discussed further in this paper but, from our perspective, some rules for the definition of reusable modules in terms of software design are missing.

This article aims to lay the initial foundations for the design of QP by addressing the concept of "module", its fundamental characteristics, and the rules to design them for maximizing its reuse. The cohesion and coupling of a quantum module were also characterized, according to a set of specific rules, grading the cohesion levels in analogy as it was done for classical structured programming in the eighties. The main contribution of this work is, therefore, that it addresses the concept of module in the field of gate-based quantum programming applications and does so from a software engineering perspective, bringing together the main contributions in the field of quantum computing that, to a large extent, have an impact on the characteristics that has to fulfill a quantum module. To our knowledge, no reference has been found in the literature that addresses the definition of module, so this contribution can serve as a starting point for further work towards a robust software engineering approach to QP development.

The article is organized as follows. Section 2 gives a background on the quantum programming concepts to facilitate the reading of the content of the article and to describe the nomenclature used in it. Section 3 provides a definition of module and the rules that must be considered together with a definition of several levels of cohesion and coupling in order to help QP designers to decide on the quality of their modules. Section 4 is devoted to the discussion on the results of the work and presents some future research lines worth to be explored in the near future.

2. Brief Background on Quantum Programming

This section is devoted to providing a brief introduction to the basic notions of quantum programming and to show the way in which QP are written.

The first aspect that distinguishes quantum programming from classical programming is the use of quantum bits (*qubits*) instead of bits. Unlike a traditional bit, which can only take the value 0 or 1, a qubit can be in a superposition of those two states, that is, a probabilistic combination of both (macroscopic) values. A quantum state will be given by the different values that the qubits can take plus the amplitude (α) which is set by a complex number. The magnitude associated with the amplitude is the absolute value of this complex number, while the relative phase is the angle of the complex number in polar

form. Thus, each qubit is also defined by a *magnitude* and a *relative phase* (see [18] for a complete explanation of the mathematical foundation).

Two qubits may be in four possible states, each parameterized by its amplitude, in which the quare represents the probability that the qubits, once read, will collapse to that value. A two-qubit state can be characterized as follows:

$$\begin{gathered} \textit{System state of 2 qubits} \equiv \alpha_0|00\rangle + \alpha_1|01\rangle + \alpha_2|10\rangle + \alpha_3|11\rangle \\ \textit{where } \alpha_i \in \mathbb{C} \textit{ and } |\alpha_0|^2 + |\alpha_1|^2 + |\alpha_2|^2 + |\alpha_3|^2 = 1 \end{gathered}$$

where α_0, α_1, α_2, and α_3 are the amplitudes of each state; thus, the square of the absolute value of each one is the probability of reading such values when collapsing the former two qubits system. Thus, the superposition of 2^n states can be described with just n qubits, which is a tremendous computational advantage over the classical n bits, where only a single fixed state is available. In each execution time, and prior to measuring, the qubit can be a superposition of the previous states. In other words, we no longer have a classical bit with a specific value (0 or 1), but a qubit given by a superposition of values 0 and 1.

Computation in quantum programming is performed, under the circuit representation of QP, by means of gates, which provide the primitive operations to manipulate the magnitude and phase of the system qubits. This notation resembles the origins of current electronic computation, where the gates implemented the traditional Boolean operations. A quantum computer implements in practice a set of primitive gates which are used to build others, using circuits as a collection of interconnected quantum gates and operations. The type of available gates depends on the hardware infrastructure of the quantum computer, determining the way in which a program is solved in a specific machine [19].

Without loss of generality, we restricted the discussion of this work to the universal set of gates described in the seminal article [20]. Among the basic set of gates, the Hadamard (H) is one of the most used as it provides an equal superposition of states. Other gates such as NOT (the quantum equivalent of the conventional gate), Z, ROTX, ROTY (for the manipulation of qubits properties by manipulating the relative phase in different axis), SWAP (for exchanging two qubits), and CNOT (a two-qubits gate that applies the NOT operation to a target qubit only if the condition qubit has the value 1) are among the basic operations that can be considered on a set of qubits before collapsing through a READ operation. A set of single-qubit gates plus CNOT is called universal when they can be used to construct any unitary matrix (see [20,21] for more details).

As an example, Figure 1 shows a system with two qubits in superposition after applying two H gates and Z to the second qubit. Each qubit has a 50% of probability of collapsing to 0 or 1 but the second one has a relative phase of 180° (see the angle of the radius drawn). The magnitude of the amplitude associated with each value a qubit can take (i.e., $|0\rangle$ and $|1\rangle$) is proportional to the radius of the filled in area shown in the circles.

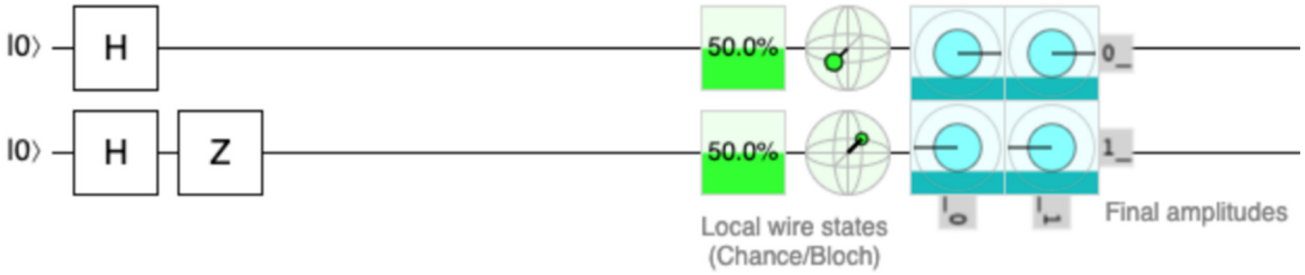

Figure 1. Two qubits in superposition and the second with a relative phase of 180° (image obtained using the https://algassert.com tool, accessed on 23 June 2021).

It is important to note that when a state is read, the superposition is lost. For instance, after reading any qubit on Figure 1, there is a 50% of probability to observe each value of the state, so the value read could be 0 or 1. This is one key aspect of quantum programming:

the exploitation of performing computations in superposition, but at the same time, one of the biggest headaches for programmers, along with quantum entanglement. Quantum programs usually consider the entanglement mechanical phenomenon where a part of the program cannot be described without considering another part, even if these parts are considerably separated in space. Thus, measurements done on one part instantaneously affect the other part entangled with it. This property has huge applications in quantum computing, especially in cryptography [22].

A quantum register is a collection of qubits used for quantum computation. A quantum circuit is a sequence of gates applied to a subset of the qubits of a quantum register. Figure 2 shows a quantum circuit demonstrating the increment and decrement of integers in superposition. More specifically, we implemented the increment and decrement operations using as input the value $|1\rangle$. As it can be seen, the decrement operation is simply the increment with its constituent operations reversed. The initial instructions write the value $|1\rangle$ to a quantum integer and then execute H and Z gates on the third qubit, resulting in a register as a superposition of $|1\rangle_4$ and $|5\rangle_4$.

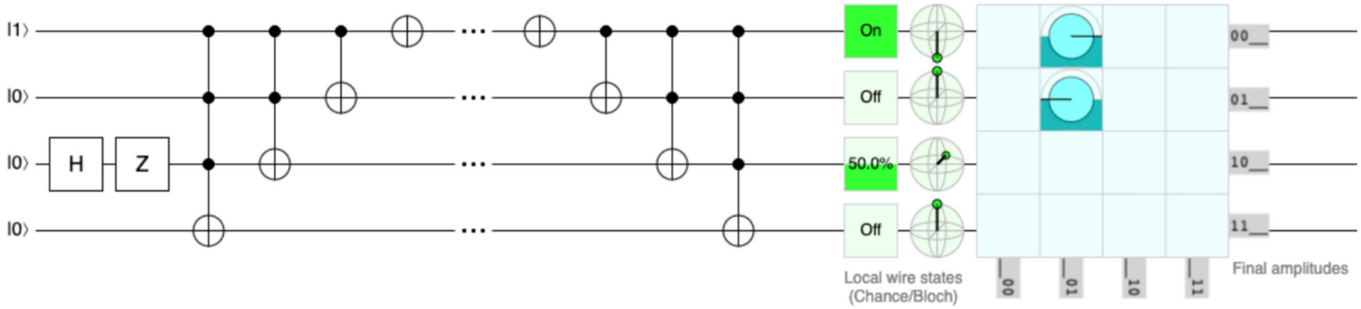

Figure 2. A quantum circuit including four qubits and implementing the increment (first part) and decrement (second part) of an integer (image obtained using the https://algassert.com tool, accessed on 23 June 2021).

Basically, the steps followed for quantum computing include: (1) the initialization of qubits, (2) several operations on those qubits by applying any combination of gates, and (3) the measurement of quantum states to get classic data. The second step can be the sequence of many different combinations of gates, carrying out well known abstract operations such as entanglement, the computation of function tables, the reuse of computations of another quantum algorithms (Oracle, QFT, etc.), uncomputing for removing the entanglement that resulted from computation, and operations with phases to converts inaccessible phase differences into readable magnitude differences, among others.

There are many quantum primitives and algorithms of high interest that can be used as building blocks of bigger modules. Next, we provide a summary of the main ones.

Many of the quantum algorithms that have a speedup on quantum versus conventional computers require calls to an oracle. An oracle provides, in a black-box fashion, information about a variable or a function, without revealing the variable or functions itself. The task in some algorithms is to determine the variable or function used by the oracle in as few calls as possible. In other words, an oracle is a function which we supply with data, and it responds with a yes/no. Some of the most well-known oracle-based algorithms are Deutsch–Jozsa and Grover's. The Deutsch–Jozsa algorithm solves, with exactly one query, a black-box problem which probably requires exponentially many queries for any deterministic classical computer. The algorithm determines whether a function is either constant (0 on all inputs or 1 on all inputs) or balanced (returns 1 for half of the input domain and 0 for the other half). Grover's algorithm enables one to find, with a high probability, a specific item within a randomly ordered set of N items in $\sqrt{N}$ operations (it is a quadratic speedup, since a classical computer requires O(N) operations). In general, Grover's algorithm can be applied when having a function which returns true for one of its possible inputs, and false for all the others. The job of the algorithm is to find the one that returns true. The basic ideas that comprise this algorithm are applicable in a much broader

context than searching on a database as it could be used for the implementation of other algorithms (for example, the factoring Shor's algorithm for factoring integer numbers).

The following are building blocks and patterns that appear in many algorithms. Quantum amplification converts inaccessible phase differences inside a QPU register into readable magnitude differences (and vice versa). It is a generalization of the Grover's search algorithm that can be used as a subroutine providing performance speedup there where it is needed to check the validity of a solution (such as Boolean satisfiability, finding global/local minima, among others). Quantum Fourier transform (QFT), the quantum analogue of the inverse discrete Fourier transform, is a primitive allowing quantum software developers to access hidden patterns and information stored inside a QPU register's relative phases and magnitudes. While the amplitude amplification primitive allows information to be extracted about discrete values encoded in the phases of the register, the QFT primitive aims to obtain patterns of information encoded in the QPU register. QFT is part of other well-known algorithms such as Shor's algorithm and quantum phase estimation algorithm. Quantum phase estimation is also a very useful tool that, as with amplitude amplification and QFT, extracts tangible, readable information from superpositions. Phase estimation helps determine (to some precision) the eigenphases associated with the eigenstates of a QPU operation, returning a superposition of all the eigenphases. In other words, it finds a way of moving information about the global phase into another register in a readable form.

There are many other well-known quantum algorithms such as Shor's algorithm for solving the discrete logarithm problem and the integer factorization problem in polynomial time, and Simon's algorithm to find how the values of a function on whole numbers repeat themselves. All these and other primitives are common and frequent assets for building QP. A more detailed description of all the algorithms and applications can be found in [23].

Given this initial background to quantum computing and how QP are built by means of gates and circuits, we discuss how to complete the first steps into the modular development of QP.

3. Towards Modular Quantum Programming

This section provides, after detailing the existing gap between classical and quantum programming, some insights with respect to the concept of modularity in the quantum computing field. First of all, some background on the existing gat between classical and quantum ways of programming is given in Section 3.1. Afterwards, the main existing works dealing with modularity in the quantum regimen are given in Section 3.2, and finally, the definition of module for QP is given together with the definition of cohesion and coupling metrics.

3.1. Impedance Mismatch between Classical and Quantum Programming

In traditional software engineering, the engineer makes use of conceptual tools to model the target system in the problem domain and, with gradual transformations, develops software using a programming language, where the design concepts have some correspondence with the implementation ones. Object orientation is the culmination of this process, where classes normally mimic concepts of the problem domain. In other words, through an iterative process of successive refinement, engineers arrive at a solution where the common concepts identified in the previous phases are not lost along the way.

In contrast, when developing a QP, the engineer has to make use of mathematical structures and algebra operators to manipulate qubits. More specifically, they make use of magnitude and phase manipulation tools to provide a solution to a well-defined problem. Such tools do not come from a direct translation from concepts and rules used at the design level, but rather from the quantum mechanical field of physics itself. There are no classes or functions or similar high-level concepts (yet) at the programmer's disposal to develop QP. Therefore, engineers face a big conceptual gap between conceptualization and implementation. This implementation, in terms of low-level concepts, is in a manner equivalent to how a programmer would implement the Quicksort algorithm in assembler,

which reminds us to a large extent to the beginnings of computing, with wiring programs directly on the primitive ENIACS.

The clear proof of this conceptual gap is the fact that, although most of the algorithms conceived and widely known in quantum programming (such as Grover, QFT, and Shor to mention a few) are very well described and studied [23], they are difficult to understand, reuse, and much less to be extended by most of the "classical" programmers, a barrier that cannot be minimized as of today by learning the fundamental concepts of quantum programming.

To add more difficulties, very basic operations of classic computing do not have a direct equivalence in quantum computing and have to be carried out in different, and sometimes very creative, ways. For instance, copying the value of a variable into another, a basic operation in traditional computing, is not possible in quantum computing, as information cannot be replicated (because of the no-cloning theorem [8]). The only way to do that is by means of teleportation [24] but at the price of destroying the qubit being teleported, the original one. This is only an example of the big conceptual gap between our way of thinking and the way in which quantum mechanics constraints what can be programmed with quantum computers.

Authors of [23] analyzed up to twenty well-known quantum computing algorithms which are each one classified according to five paradigms (Grover operator, Quantum Fourier Transform, Harrow–Hassidim–Lloyd, Variational Quantum Eigenvalue solver, and direct Hamiltonian simulation). A combination of such paradigms can be found in the implementation of any quantum program. The number of quantum paradigms is reduced compared to classical software case because of the limitations imposed by unitarity and reversibility requirements of quantum operations and the intrusiveness of measurements. According to the area of application, each algorithm can also be classified into a class of quantum algorithm, namely, algebraic applications, graph applications, learning applications, inverse function computation, number-theoretic applications, quantum simulation, and quantum utilities.

To add, if possible, more complexity to the new software development paradigm, QP have to be developed using the minimum number of gates in order to finish execution before decoherence and other quantum errors arise, which reduce the probability of success in the computation. Classical computers are also affected by many sources of errors, but they can be fixed with a reduced and well-defined extra memory and logic. However, the loss of coherence in quantum computers (coming from vibrations, fluctuations of temperature, electromagnetic interactions with the environment, among others) produce a loss of the amazing properties of the quantum computer. To minimize this problem, engineers have tried to develop methods for algorithm optimization (also known as transpilation) to be able to execute them on noisy quantum computers (see [3,25] for more detail on this subject). Many companies are trying to minimize decoherence by using more robust quantum processes and studying different ways for detecting errors (mainly by using temperatures close to absolute zero and trapping atoms in electromagnetic fields).

To sum up this motivation, quantum computers are not a simple evolution of classical computer in the same way as quantum programming is not yet another programming paradigm, and has nothing to do one with the other. As described in the literature, there are some problems where quantum algorithms (such as Deutsch–Jozsa and Simon) achieve exponential speedups over any known classical algorithms, which is a revolution in the computation field as quantum computers might surpass existing computers for some kinds of problems.

The continuity of classic computers and software development tools developed has not been questioned in recent years. Quite the contrary. Current computers remain, at least in the medium term, as the most simple and economical way to solve most of the problems. At the same time, quantum computers have the opportunity of promoting radical advances in many fields such as chemistry, materials science, cryptography, machine learning, engineering, and finance, among others.

3.2. Background on Modularity in Quantum Programming

In general, modularization is the process of breaking down a software into several modules, each one developed separately. One module should have limited or no dependency on other modules, in such a way that introducing changes in one of them affects the minimum number of other modules in order to reduce error propagation and maximize reuse. Modules also reduce intellectual burden on programmers, who can focus on just the functionality offered by the module. This concept has permeated and is present, in various forms, in all "traditional" programming languages.

Quantum programming, with its special and counterintuitive rules and principles, should not be an exception to the inclusion of modules. Problem decomposition and module reuse have served us very well over the last 50 years in computer science (and for centuries to Humankind in other disciplines), and we should at least try to include modules and its key features into this new software development paradigm. However, to be able to talk about problem decomposition and reuse, we first need to explore the minimum reuse unit, the module, as well as the rules that constraint how to integrate and connect these units to other parts of a QP.

Module independence is a key characteristic of any design. To measure it, at least qualitatively, the concepts of cohesion and coupling have been defined [16]. Cohesion is a measure of the relationship strength of the elements included in the definition of the module. In classical structured programming, seven levels of cohesion are distinguished, ranging from functional (the best) to co-incidental (the worst). Inter-related with cohesion, coupling measures the relationship strength among modules themselves. Six types of coupling have been distinguished, ranging from data (the best) to content (the worst). A good software system design should look for designing modules with high cohesion and low coupling [26]. These quality attributes have guided software developers for decades to achieve software systems of the highest possible quality.

In the context of quantum programming and to our knowledge, there is only one work analyzing the concepts of module, cohesion, and coupling. In [14], several quantum architecture design characteristics to measure these attributes were identified. It distinguished two levels of cohesion, depending on the use of qubits superposition (temporal cohesion) and the existence of "correlation between neighbouring qubits in order to interact". Regarding coupling, the authors identified four levels, ranging from measure coupling (which exists between a quantum state and its environment) to non-coherent coupling (caused by loss of energy from emission of photon to the environment). From our point of view, both cohesion and coupling are defined there from a hardware perspective, and therefore, are not a useful tool for software modeling. Moreover, the definition of module is not clearly stated.

Prior to these definitions, we must clearly define what a module is or can be in quantum computing terms. In [17], a formal framework for introducing modularity in quantum circuits is included. They demonstrate that it is not always possible to reuse a quantum circuit without knowing its implementation and, in certain situations, a small change in the algorithm can lead to a fully re-conception of the quantum circuit itself. Figure 3 shows an example of an abstracted quantum circuit with two slots to parametrize different implementations of unitary transformations (U), which range over the set of possible inputs $\{U\}$.

We can have many algorithmic realizations of each U_x, provided that each one produces the same output (at the vector level) given the same input. According to [17], given U_x, we can say that a circuit P is *modular* with respect to U_x if it is possible to construct a U_x-independent module with different implementations provided that: (1) there is independence with respect to the infrastructure used for its realization (this requires that the physical systems implementing P and U are well-known, including the respective Hilbert spaces and computational basis), and (2), no changes on P are needed for replacing one U_x implementation by other equivalent.

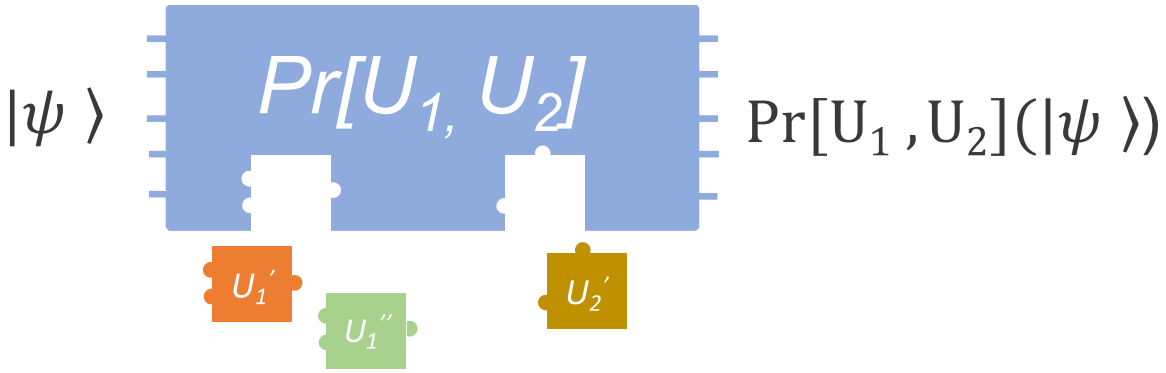

Figure 3. A modular approximation for quantum computing. Given a set of qubits at input, they are processed according to a given circuit, which is parametrized by many entry points where different U modules can be plugged into.

There is a constraint that must be satisfied if we want to have U modular with respect to P. Here, the classical concept of black box is needed. A black box [27] is a device or system which can be viewed in terms of its inputs and outputs, without any knowledge of its internals. Two algorithms are equivalent and indistinguishable if, given any inputs, they generate the same outputs. Thus, if P is modular with respect to U, then $P[U]$ does not reveal which algorithm has actually been used for implementing U. This is a constraint limiting what kind of quantum computations can be made modular. As described in [17], there are many algorithms (like phase estimation, quantum factoring, solution of linear equations, among others) that are not modular (because they work by extracting information beyond the black-box properties of the embedded module), thus precluding the implementation of circuits that work for every possible input and forcing to be adapted for each possible entry. In these cases, some redesign is needed in order to make them modular.

Thus, in spite of the importance of being able to define QP as a set of separately constructed modules that are combined as needed without knowing their internal implementation, scarce theoretical results have demonstrated in which situations the modularity is not possible, although they also have provided methods to avoid the situation for certain algorithms. In some cases, modularity can be achieved at the cost of losing part of the information computed, while, in other cases, the information sacrificed is not crucial (for more detail about this subject, readers can refer to [17]). In any case, more guidelines for modular design are needed in order to facilitate the development of QP.

3.3. Quantum Modules

Modules are considered, under the circuit representation, as "regular" gates that take qubit lines as inputs, apply the operations included inside the module to the joint state represented by the input qubits, and have the same number of qubits lines as output. Each operation (gate) corresponds to applying a concrete matrix to the quantum state, so the gates that make the module up are written from left to right. Figure 4 shows a simple quantum module with four (input and output) qubits (a), and a program composed by two sequential modules (b).

It is also possible to apply a module to a subset of the total program qubits, as shown in Figure 5a, when there is no entanglement between the involved qubits, the equivalence with the application of identity operators to the other qubits, as shown in Figure 5b. This is because $(I \otimes I \otimes U)(|\psi\rangle \otimes |\theta\rangle \otimes |\xi\rangle)$, where $\otimes$ is the tensor product, is the same as $|\psi\rangle \otimes |\theta\rangle \otimes U|\xi\rangle$ (again, when there is no entanglement among input qubits). When the qubits are entangled, as in Figure 5c, the state cannot be factored as a product of the individual qubit states. Tensor product is the machinery used to build quantum systems from other existing quantum systems. See [18] for more details on the mathematical foundations.

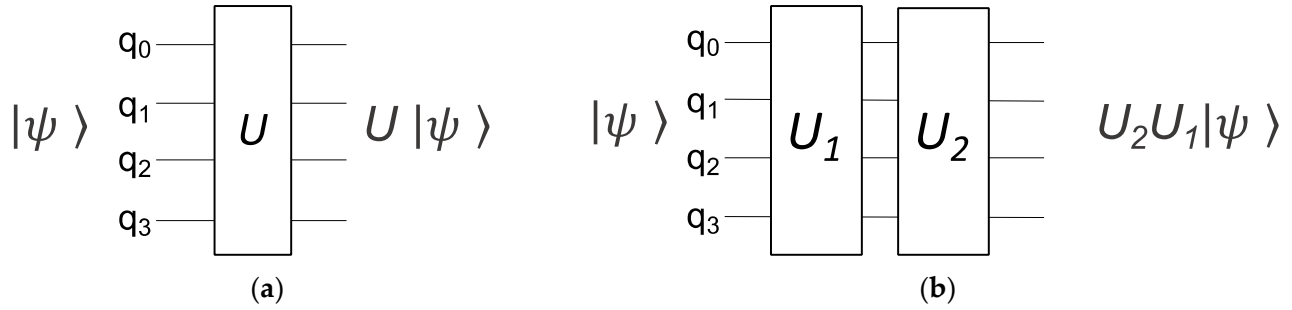

Figure 4. (**a**) Module a simple gate. (**b**) Module with two sequential gates.

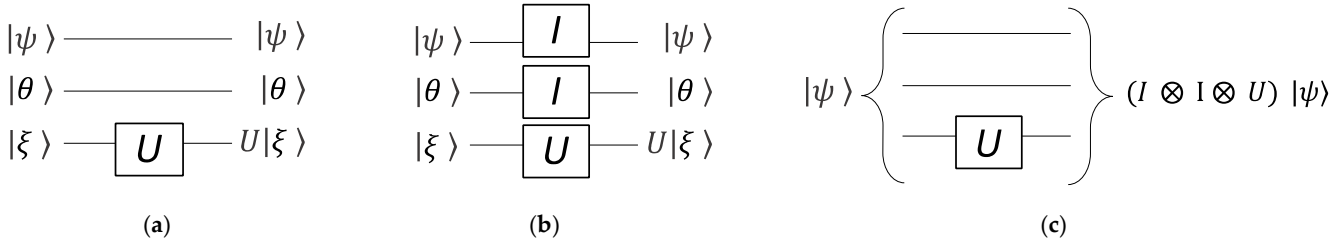

Figure 5. Module with a gate applied to a subset of qubits: (**a**) a single gate to a qubit; (**b**) the equivalence using identity gates; and (**c**) the situation when there is entanglement.

It should be noted that in quantum computing, not every unitary matrix is allowed for implementing quantum systems. Those gates (as unitary matrices) are only valid coming from the combination of gates from the basic set, equivalent to the basic instruction set of a CPU. This set is dependent on the selected quantum hardware for executing the quantum program.

Considering all the above, the definition of module can be as follows. A quantum module is the basic reuse unit in quantum programming, which provides a specific functionality and has an interface given by the same number of input and output qubits. A module has to satisfy the following rules, summarized in Table 1:

1. Any quantum gate belonging to the universal gate set is considered itself a quantum module;
2. A quantum module has to be unitary and reversible. This is because every operation on a state must keep the sum of probabilities of all possible outcomes at exactly 1, and it must be possible to obtain the previous state by using only the output state, without additional information. In order to be reversible, no measurement of any of the input qubits can be made inside the module. In addition, the designer of a module U may need to include additional input qubits in order to allow the development of the module $U^{\dagger}$ that performs the inverse operation: reconstructing the original input qubits that entered U. This $U^{\dagger}$ module is computed as the Hermitian matrix (transposed, complex conjugate) of module U;
3. In case the module needs scratch qubits to store the result of temporary computations, it has to make sure that they are returned back to the value they had before entering the module. This can be a hard process in case the scratch qubits got entangled inside the module. Scratch qubits that are entangled with data qubits alter the state of the data qubits if they are reset or measured. Reclaiming scratch (ancilla) qubits is the process of returning them to their original state for future reuse [28]. This facilitates the reuse of the scratch qubits on subsequent computations, and, at the same time, the modification of the scratch qubits downstream does not affect the output qubits in case they are entangled;
4. A quantum module can be replaced indistinguishably by any other equivalent module which, given the same inputs, produces the same outputs. In other words, it

should be possible to make use of the module without knowing its construction, as a black box. Please note that, according to the "no-go" theorem described in [17], the implementation in circuits of certain quantum algorithms may force to rebuild almost entirely the circuit after changing some input data. These types of circuits are not considered modular according to the given constraints, as they can reveal information regarding which algorithm was used beyond its black-box properties;

5. It is possible to include a controlled sub-module to decide, at run-time, if the sub-module should be executed or not. Controlled modules are a particularly important class of modules where one qubit (the controller) turns on or off a unitary operation U. This is a very relevant question in quantum computing because in many cases it is not clear how to directly add a control to a quantum operation, especially in situations where the module implementation is unknown. This is a matter of research that has generated a lot of theorical work [29]. For our purposes and considering the desired black-box properties of the modules, the only way to accomplish that is through the use of the SWAP gate, as it is described in [30];
6. Any explicit sequence of universal gates or sub-modules integration is also considered a quantum module, provided that all the stated constraints are satisfied;
7. Input qubits should not be entangled (though superposition is allowed) with other qubits that do not enter the module or a sub-module. Otherwise, the execution of the module will generate changes beyond the output qubits specified in the definition of a module as a black box, effectively breaking encapsulation. In addition, it could hinder reversibility since the module does not have enough information to reverse the computation. This last rule is added more as a recommendation than an obligation but is needed in order to achieve universal module reuse (for more details on this see the Discussion Section).

Table 1. Summary of rules for building up quantum modules.

Property	Description
Uniformity	Basic quantum gates are modules
Composability	Any valid sequence of modules is a module
Indistinguishability	Given inputs, if same outputs then same module
Unitarity and reversibility	Modules have to be reversible because quantum mechanics is reversible
Ancilla reclaiming	Reclaiming ancilla qubits via uncomputation
Entanglement limitation	Input qubits should not be entangled with others external to the module
Feasibility of controlled sub-modules	A control qubit can decide whether or not to call a submodule

Figure 6 shows a possible abstract structure of a general module. Some scratch (ancilla) qubits could be considered (as auxiliary) in order to make the computation reversible. The result of the computation could be stored in the relative phase of the output qubit/s (or directly as magnitude). The scheme showed corresponds to a module which itself includes the reverse of the operation performed. As long as the computation does not interfere with the relative phases of the output registers, the information can survive the uncompute step [30]. Please note that not all parts of the figure are mandatory. For instance, a module may not need scratch qubits or to perform the uncompute step.

Figure 7 shows an example of a module developed to add two integer numbers encoded in 4 and 3 qubits, respectively. This module implements the expression $a = a + b$. This in an example of a module that follows the aforementioned structure, but which: (1) does not require scratch qubits to store temporary values, (2) does not require additional qubits to store the result of the computation, and (3) does not require an uncompute step, since the result is stored on the input qubits. In addition, the operation is performed in such a way that is reversible: Input values can be re-constructed from output one just by subtracting the second number from the first one.

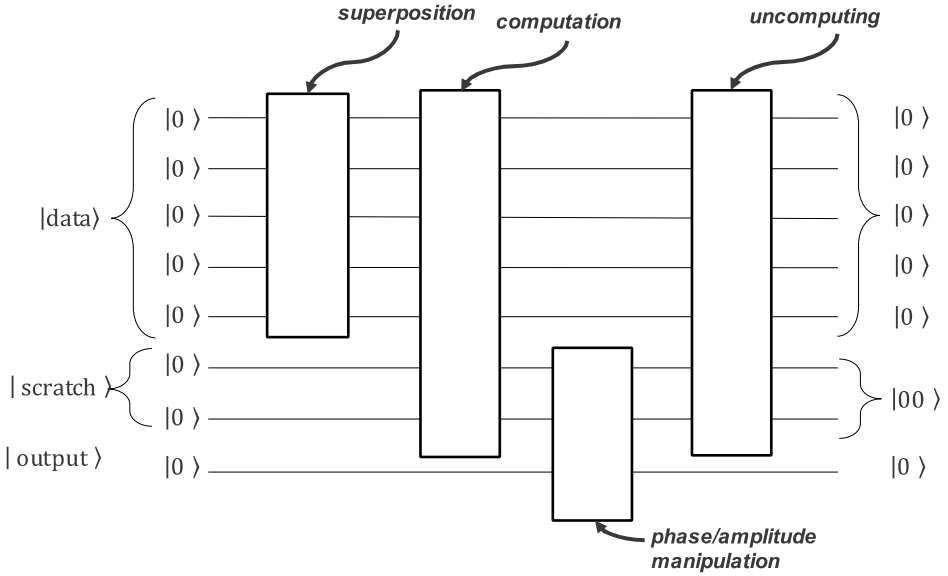

Figure 6. General scheme for a reusable module.

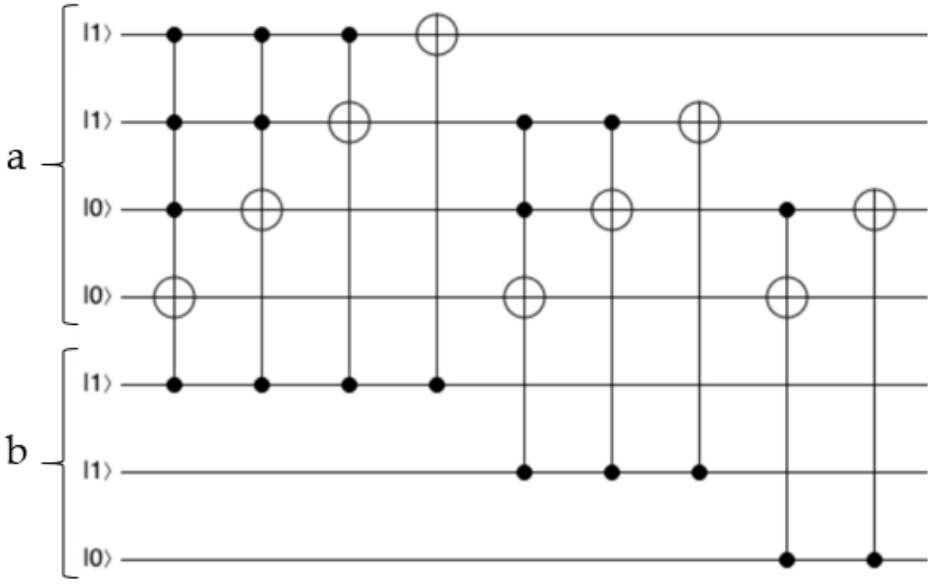

Figure 7. Example of reusable module for adding two numbers represented by the register a and b. Despite it does not include all the elements indicated above, it is a module that fulfils the constraints required to be a module (image obtained using the https://algassert.com tool, accessed on 23 June 2021).

Figure 8 shows an example of a module which implementation is closer to the abstract scheme shown in Figure 6. It solves a concrete 3-SAT problem including three Boolean inputs. After a superposition of the inputs, two logical clauses are evaluated, followed by a flip of phase to highlight the selected valid combination of Boolean values; afterwards, uncomputing is performed to obtain the original values that satisfied the Boolean equation, followed by a mirror operation as an amplitude amplification mechanism.

The next section is devoted to classifying modules according to their levels of cohesion and coupling.

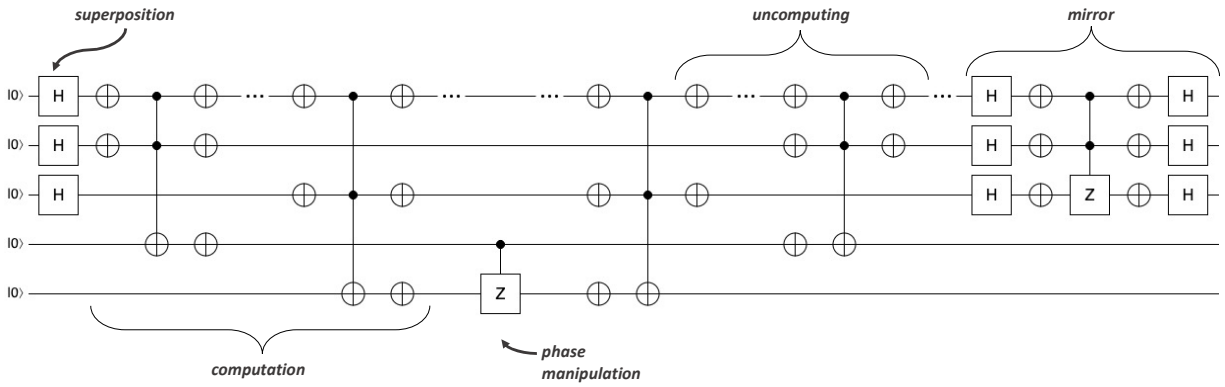

Figure 8. Sample of reusable module for solving a 3-SAT to find whether exists a combination of the three Boolean inputs to evaluate as true a given logic expression (image obtained using the https://algassert.com tool, accessed on 23 June 2021).

3.4. Cohesion and Coupling Levels for Quantum Modules

After the definition of quantum module and its main characteristics, we can now tackle the study of the cohesion and coupling levels. We followed, to some extent, the naming conventions of cohesion levels from classic programming; however, in some cases, we adopted new names and constraints, according to the quantum context. From best to worst, we considered that a module U can have functional, sequential, communicational, and clustered cohesion. For all the definitions we always assumed the same Hilbert space and communicational basis among the quantum information exchanged by modules.

We argued that a module *U* has functional cohesion when its sub-modules contribute to the realization of a single well-defined task. Figure 9 shows an example of functional cohesion, where there is a module (called "incr") for incrementing an integer number represented by three qubits. Please note that the other constraints coming from the definition of module were also fulfilled (specially, its reversibility). Functional cohesion is the greatest level and the most desirable in terms of reuse and modifiability of a QP.

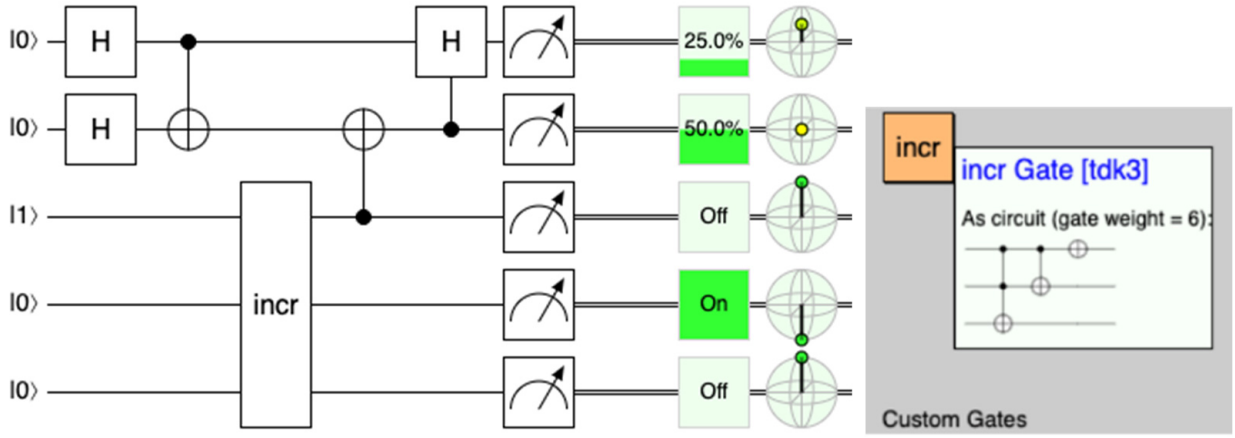

Figure 9. Circuit including the reuse of a module (called "incr" and detailed on the right side) with functional cohesion (image obtained using the https://algassert.com tool, accessed on 23 June 2021).

When the module performs more than one task, but still the order in the execution of the gates is relevant, we say that the module *U* has sequential cohesion, being a lower level than functional one. An example could be a module performing the increment and the multiplication by a factor of two.

The next level is communicational cohesion. We consider that a module *U* has communicational cohesion when the embedded sub-modules use, each, only a subset of the input qubits to carry out separate tasks, and their partial results are returned by the module.

However, there are some execution dependencies among some of the tasks. Figure 10 shows the situation of a module that computes both the Boolean OR and AND operations (the last two qubits of the circuit are where the outputs are calculated and stored). This level of cohesion is not desirable because the reuse of the module is not so feasible. It should be preferrable to split the tasks of *U* into two separate modules embedded sequentially into the circuit, as is taught on first year courses on classical programming.

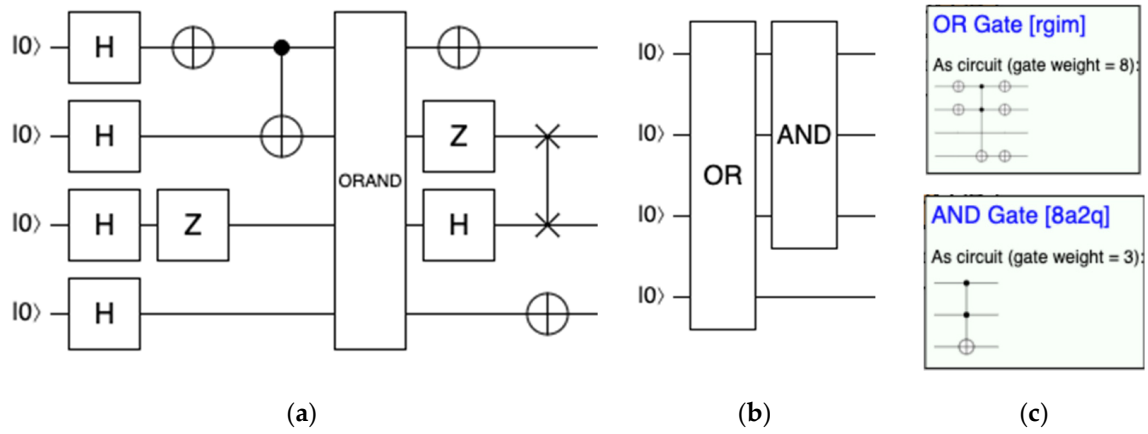

Figure 10. (**a**) A sample of a circuit embedding a module with communicational cohesion. (**b**) The implementation of the module including OR/AND submodules. The tasks performed within the module (AND/OR operations) share the first three incoming qubits as input data. (**c**) Detail of the implementation of the submodules.

Finally, a module has clustered cohesion when the embedded sub-modules carry out independent tasks that overlap in time, and therefore, there is no order. A control qubit can activate or deactivate the execution of some tasks of the module. This is the worst level of cohesion because it greatly prevents the reuse of the module.

Coupling is a metric used to measure the degree of interdependence between software modules. In this case there is no direct correspondence with the coupling levels of classical software as the concept of "module call" does not exist as such in quantum programming. In this context, we studied two situations: (a) the coupling between a module (Ma) and the submodules ($M_{a,i}$) used to build M_a, and (b) the coupling between the submodules Ma,i themselves as part of the realization of M_a.

Regarding (a), we argued that $M_{a,i}$ is data coupled with Ma when the only information exchanged between the Ma and $M_{a,i}$ is a subset of the qubits handled by Ma. This is the coupling level by default. We argued that $M_{a,i}$ is control coupled with Ma when one or more input qubits of $M_{a,i}$ are used to control internally its execution. Coupling by data is the desirable level for all the modules and it is generally related to functional, sequential, and communicational cohesion. Control coupling may be more frequent for clustered cohesion and should be avoided to have a reusable module.

Regarding (b), two submodules $M_{a,i}$ and $M_{a,j}$ are coupled by entanglement when some output qubits of $M_{a,i}$ are entangled with input qubits of $M_{a,j}$ as a consequence of the execution of $M_{a,i}$. This is a high level of coupling and should be avoid because when this occurs the effect of the module $M_{a,j}$ is given not only by its input qubits but also by the interactions that its qubits may have with $M_{a,i}$, canceling the expected properties of a black-box module. Please take into account that we were not ruling out the use of entanglement but rather limiting its scope to the boundaries of the module.

Both the definition of module and its rules, together with the different levels of cohesion and coupling, were a useful tool for the design of any QP.

4. Discussion

For a successful achievement of a quantum software engineering discipline, we considered crucial the establishment of abstractions for quantum software development. In

object-oriented software development, as an evolution from the original abstract data types, the main abstractions used were the concepts of object and classes, as factories for creating objects according to a given template. Before that, procedures and functions were the only abstractions available to implement software, defined as a sequence of instructions that manipulate (common) data, and allowing hierarchical interactions through procedure-to-procedure calls.

At first sight, the notion of class may not be probably an adequate abstraction for quantum software development. The software engineering community may have the temptation to quickly adopt well-known abstractions (classes, objects, etc.) and their corresponding techniques and look for the equivalence for developing quantum software, trying to do something that is radically different. As stated in [31], researchers should focus their contributions having in mind to "think and model in quantum".

This is exactly the rationale behind the seven rules that define a module that have been described in this work. We think that they are the minimum set of rules necessary to allow the successful creation of modules in quantum programming. However, they can nevertheless be extended with additional ones by quantum software designers according to their specific needs and limitations (for instance, characteristics of the hardware where they execute their QP).

The rule #1 states that the elementary quantum gates are modules, while #6 says that all groupings of gates that meet the previous rules are also modules. That is, they are the rules that establish that it is possible to create new modules by composing modules that meet certain properties. These types of rules are present in the definition of reusable entities in all classical programming languages.

Rules #2 to #5 are derived from the intrinsic characteristics and limitations imposed by quantum mechanics as follows. Rules #2 and #3 are directly related to the fact that computation must be unitary and reversible. If additional qubits are used to store temporary results (ancilla qubits), they must be restored to the previous state after performing the computation before being used in a new computation to avoid unwanted effects of their (possible) entanglement with other qubits. Rule #4 is, perhaps, the most surprising one, derived from deeper research on modularity of quantum circuits. It states that a set of gates and sub-modules can be considered modular, and therefore, can be reused as a building block in other QP only if its black box properties, that is, considering only the input and output values, do not allow the user of the module to identify how it is implemented. From this it follows that, unlike classical programming languages, not every grouping of quantum gates is modular per se. Rule #5 only applies if, in a module, it is needed to include control-U functionality. That is, if one or more qubits are going to control whether a certain part of the module, which implements a U operation, should be executed or not. As such, this rule is not essential since many algorithms do not need to make use of this functionality. However, if needed, it should be included carefully, for instance, by using SWAP gates.

Rule #7 states that, if any entangled qubit enters a module, then all the qubits that are entangled with it must enter the module too. This is a recommendation, rather than an obligation, considering that we aim to achieve universal module reuse, that is, being able to reuse the module without knowing anything about the rest of the quantum program in which it will be integrated into. The objective is to mimic the characteristics of the standard libraries that come with programming languages, like <math>, which provide functionality that can be used and reused many times, and which introduce no restrictions or assumptions about the source code that surrounds their use. Entanglement opens up the possibility that qubits entering a module are modified by some other parts of the quantum program, outside the module's control. It is like having global variables or pointers to the local variables of functions, which can be modified from any point of the source code, in a concurrent program. This is, in general, a bad design. If the programmer is careful, one can still create a program that does its job and uses global variables, since one knows and controls how the code is implemented. However, it would be hard and error-prone

to reuse these functions in other programs, especially by third parties. Our position with entanglement is similar to this one, and this is why we state that, in order to maximize reuse, all entangled qubits or none at all should enter a module. This rule may seem too restrictive but, from our point of view, is needed if we want to converge to an off-the-shelf scheme of module cataloguing and reuse in quantum programming.

We have also purposely avoided, in general, references to concrete quantum programming languages when defining the main properties that a module should have. Our objective is to provide a "language-agnostic" module definition, which could be used in any such language, given the variability in their implementation (from imperative languages like Q#, functional languages like QPL to specialized libraries and SDKs for general-purpose languages like Qiskit).

We adapted the classical definitions of cohesion and coupling between modules to the characteristics of the (circuit-based) quantum programming and quantum modules. The different levels of cohesion highly determine the coupling level between the modules. As expected, the higher the cohesion, the lower the coupling, which provides much better reuse, among other benefits from modular design. The provided classification can be subjected to other points of view and updates, given the novelty of quantum programming and the small quantity of examples to take as input to evaluate it. For instance, since there is no support for datatype definitions right now (data are just qubits) it makes no sense to consider stamp coupling. However, this may change in the future. Further, most of the developed algorithms were designed for a specific task and they are not usually conceived thinking in their reuse as part of a more general program. As the development of new software and algorithms for quantum programming progresses, new case studies become available to validate the usefulness of heuristics such as those provided in this work. On the other hand, we identified a new type of coupling that is only present in QP: coupling by entanglement.

We also made a proposal for a general internal structure that a module could have, according to the module definition rules described in the article. Not all the blocks proposed on the structure are mandatory, but rather the objective is to provide a starting point, it is, a template for helping the quantum module designer to keep in mind all the options that he should consider when developing quantum modules.

As with general purpose languages where the use of generics is important, the parameterization of modules in such a way that concrete realizations can be made from a template is also of interest. In terms of module parameterization, we identified two scenarios at the design level: (1) when including gates with a concrete configuration parameter (e.g., indicating the angle of the phase rotation gate) into a module, (2) when considering a conceptual parameterization at the schematic design level (e.g., the one we have when it is indicated to the phase estimation algorithm that it receives the operator U on which such estimation is made). In the first case, the parameterization is seen as the concrete realization of the gate being embedded on the module. In the second case, it is an abstract representation aspect of the algorithm whose only implication is the fact that when implementing the specific algorithm, the gates necessary for its implementation have been included as part of the module realization (in the case of the example of phase estimation, it would be equivalent to the unitary operator U on which the invoked estimation is made, applying all the controlled operations $C - U^{2^j}$). The first scenario is fully compatible with the definition of module provided above. However, the second one requires a deeper research since in theory it would be possible to provide a concrete operation U that could make the whole module not valid according to our definition.

From our point of view, one important contribution comes from the definition of domain-specific languages for quantum software development (QDSL). To that aim, the extensions of the UML notation for enabling classical and quantum aspects to be modeled together are very useful for software developers (for example, see the quantum UML profile introduced in [12] to design and model quantum circuits using UML activity diagrams).

Together with the use of QDSL, another major contribution to the establishment of quantum software engineering discipline comes from working on the use of patterns [10]. The definition of a repository of useful patterns will serve as generic way solutions to recurring problems in the context of quantum computing. The independence of platform implementations, assumed by the definition of pattern itself, is still a matter of research in the quantum computing field given the differences between quantum execution platforms. Nevertheless, it serves as a very interesting starting point for the success of the quantum software engineering. The combination of patterns with a QDSL for defining not only the patterns, but also the quantum applications using these patterns, would be tremendously useful for software practitioners. It is needed a means to support not only the definition of patterns and the combination of them, but also a refinement process to directly applicate these patterns to concrete solutions modeled using the QDSL. This needs to be done taking into consideration the previously mentioned limitations of unitary transformations reuse and having a well-defined concept of module.

Prior to the definition of QDSLs and patterns for reuse, the software engineering community must agree upon adopting a set of common criteria, including the definition of quantum program modules and a way to measure their modularity. The promotion in short term of reuse facilities and the achievement of the well-known benefits from traditional software modeling techniques is, without a doubt, of great interest for the adoption of software engineering techniques in quantum computing discipline.

As stated in [32], there is some urgency in developing and adapting classical software engineering techniques to the new challenge of quantum programming. Our work is a first step in a non-easy journey, as it has been explained in the paper. The progress on the establishment of a sound quantum software engineering discipline will depend on how solid the foundations for managing the basic gears are. Without a doubt and from our point of view, a proper definition of quantum module is part of the solution.

The next steps from our side will be, on one hand, the definition of initial methodological heuristics for guiding the developers in the design of QP from a software engineering perspective and considering the definition of module given in this work, and on the other hand, the identification of useful basic patterns to serve a design by reuse process. We also want to further research how to parameterize quantum modules in order to maximize their reuse in as many QP as possible, how mainstream quantum programming languages and libraries support the concept of module, and what similarities and differences exist between their definition of module and the one proposed in this article.

Author Contributions: Conceptualization, P.S. and D.A.; methodology, P.S. and D.A.; formal analysis, P.S. and D.A; investigation, P.S. and D.A.; writing—original draft preparation, P.S. and D.A.; writing—review and editing, P.S. and D.A.; supervision, P.S. and D.A.; funding acquisition, P.S. and D.A. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by Fundación Séneca, Agencia de Ciencia y Tecnología de la Región de Murcia under the 'Excelence Group Program 19895/GERM/15'.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Data sharing not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Nielsen, M.A.; Chuang, I.L. *Quantum Computation and Quantum Information*, 2nd ed.; Cambridge University Press: Cambridge, UK, 2010.
2. Heim, B.; Soeken, M.; Marshall, S.; Granade, C.; Roetteler, M.; Geller, A.; Troyer, M.; Svore, K. Quantum programming languages. *Nat. Rev. Phys.* **2020**, *2*, 709–722. [CrossRef]

3. Bertels, K.; Sarkar, A.; Hubregtsen, T.; Serrao, M.; Mouedenne, A.A.; Yadav, A.; Krol, A.; Ashraf, I. Quantum Computer Architecture: Towards Full-Stack Quantum Accelerators. In Proceedings of the 23rd Conference on Design, Automation and Test in EuropeMarch (DATE' 20), Grenoble, France, 9–13 March 2020; pp. 139–144.
4. Zhao, J. Quantum Software Engineering Landscapes and Horizons. *arXiv* **2020**, arXiv:2007.07047.
5. Smith, R.S.; Curtis, M.J.; Zeng, W.J. A practical quantum instruction set architecture. *arXiv* **2016**, arXiv:1608.03355.
6. Ladd, T.D.; Jelezko, F.; Laflamme, R.; Nakamura, Y.; Monroe, C.; O'Brien, J.L. Quantum computers. *Nature* **2010**, *464*, 45–53. [CrossRef] [PubMed]
7. Brown, K.L.; Munro, W.J.; Kendon, V.M. Using Quantum Computers for Quantum Simulation. *Entropy* **2010**, *12*, 2268–2307. [CrossRef]
8. Ortigoso, J. Twelve years before the quantum no-cloning theorem. *Am. J. Phys.* **2018**, *86*. [CrossRef]
9. Piattini, M.; Peterssen, G.; Pérez-Castillo, R. Quantum Computing: A New Software Engineering Golden Age. *SIGSOFT Softw. Eng. Notes* **2020**, *45*, 12–14. [CrossRef]
10. Leymann, F. Towards a Pattern Language for Quantum Algorithms. In *Quantum Technology and Optimization Problems (QTOP), 2019, Lecture Notes in Computer Science (LNCS)*; Springer: Munich, Germany, 2018; pp. 218–230. [CrossRef]
11. Pérez-Delgado, C.A.; Perez-Gonzalez, H.G. Towards a Quantum Software Modeling Language. In Proceedings of the ICSEW'20: Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops, First International Workshop on Quantum Software Engineering Q-SE 2020, Seoul, Korea, 27 June–19 July 2020; pp. 442–444. [CrossRef]
12. Perez-Castillo, R.; Jiménez-Navajas, L.; Piattini, M. Modelling Quantum Circuits with UML. *arXiv* **2021**, arXiv:2103.16169.
13. Honarvar, S.; Mousavi, M.; Nagarajan, R. Property-based testing of quantum programs in Q#. In Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops, First International Workshop on Quantum Software Engineering (Q-SE 2020), Seoul, Korea, 27 June–19 July 2020; pp. 430–435. [CrossRef]
14. Dey, N. QDLC—The Quantum Development Life Cycle. *arXiv* **2020**, arXiv:2010.08053.
15. Piattini, M.; Peterssen, G. The Talavera Manifesto for Quantum Software Engineering and Programming. In *Short Papers Proceedings of the 1st International Workshop on the QuANtum SoftWare Engineering & pRogramming, Talavera de la Reina, Spain, 11–12 February 2020*; CEUR Workshop Proceedings, 2020; Available online: http://ceur-ws.org/Vol-2561/ (accessed on 20 June 2021).
16. Yourdon, E.; Constantine, L. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, 2nd ed.; Yourdon Press, Pearson Education: Englewood Cliffs, NJ, USA, 2008.
17. Thompson, J.; Modi, K.; Vedral, V.; Gu, M. Quantum plug & play: Modular computation in the quantum regime. *New J. Phys.* **2017**, *20*, 1–11.
18. Sutor, R.S. *Dancing with Qubits: How Quantum Computing Works and How It Can Change the World*, 1st ed.; Packt Publishing: Birmingham, UK, 2018.
19. Jain, S. Quantum computer architectures: A survey. In Proceedings of the 2nd International Conference on Computing for Sustainable Global Development (INDIACom), New Delhi, India, 11–13 March 2015; pp. 2165–2169.
20. Barenco, A.; Bennett, C.; Cleve, R.; DiVincenzo, D.; Margolus, N.; Shor, P.; Sleator, T.; Smolin, J.; Weinfurter, H. Elementary gates for quantum computation. *Phys. Rev. A* **1995**, *52*, 3457–3467. [CrossRef] [PubMed]
21. Gimeno-Segovia, M.; Harrigan, N.; Johnston, E.R. *Programming Quantum Computers: Essential Algorithms and Code Samples*, 1st ed.; O'Reilly Media, Inc.: Sebastopol, CA, USA, 2019.
22. Bennett, C.H.; Brassard, G. Quantum cryptography: Public key distribution and coin tossing. *Theor. Comput. Sci.* **2014**, *560*, 7–11. [CrossRef]
23. Abhijith, J. Quantum Algorithm Implementations for Beginners. *arXiv* **2020**, arXiv:1804.03719.
24. Pirandola, S.; Eisert, J.; Weedbrook, C.; Furusawa, A.; Braunstein, S.L. Advances in quantum teleportation. *Nat. Photon* **2015**, *9*, 641–652. [CrossRef]
25. Unruh, W.G. Maintaining coherence in quantum computers. *Phys. Rev. A* **1995**, *51*, 992–997. [CrossRef] [PubMed]
26. Page-Jones, M. *The Practical Guide to Structured Systems Design*, 2nd ed.; Prentice-Hall International, Inc.: Upper Saddle River, NJ, USA, 1988.
27. Rice, H. Classes of Recursively Enumerable Sets and Their Decision Problems. *Trans. Am. Math. Soc.* **1953**, *74*, 358–366. [CrossRef]
28. Ding, Y.; Wu, X.C.; Holmes, A.; Wiseth, A.; Franklin1, D.; Martonosi, M.; Chong, F.T. SQUARE: Strategic Quantum Ancilla Reuse for Modular Quantum Programs via Cost-Effective Uncomputation. In Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), Valencia, Spain, 30 May–3 June 2020.
29. Araújo, M.; Feix, A.; Costa, A.; Brukner, Č. Quantum circuits cannot control unknown operations. *New J. Phys.* **2014**, *16*, 093026. [CrossRef]
30. Zhou, X.Q.; Ralph, T.; Kalasuwan, P.; Zhang, M.; Peruzzo, A.; Lanyon, B.P.; O'Brien, J.L. Adding control to arbitrary unknown quantum operations. *Nat. Commun.* **2011**, *2*. [CrossRef] [PubMed]
31. Moguel, E.; Berrocal, J.; García-Alonso, J.; Murillo, J.M. A Roadmap for Quantum Software Engineering: Applying the lessons learned from the classics. In Proceedings of the 1st Quantum Software Engineering and Technology Workshop Co-Located with IEEE International Conference on Quantum Computing and Engineering (IEEE Quantum Week), Denver–Broomfield, CO, USA, 13 October 2020.
32. Piattini, M.; Serrano, M.; Perez-Castillo, R.; Petersen, G.; Hevia, J.L. Toward a Quantum Software Engineering. *IT Prof.* **2021**, *23*, 62–66. [CrossRef]