A Quantum Software Modeling Language

Carlos A. Pérez-Delgado

Abstract In this chapter we will discuss the development of a quantum software modeling language. We begin by discussing the need for a quantum-specific modeling language, and why existing *'classical'* modeling languages may not be properly suited for the task of modeling quantum software. We then proceed with a discussion of the fundamental principles, or axioms, that any such modeling language should adhere to. We then present 'Q-UML' a quantum software modeling language based on the popular *Unified modeling Language* (UML). We conclude with some examples of Q-UML that showcase the expressive power of the language, as well as the importance of the aforementioned principles.

1 Introduction

Modeling languages are useful tools for designing, discussing, and presenting new software, hardware and complete systems. Software modeling languages, in particular, have been so useful that they can be partially credited with transforming the discipline of computer science. Software modeling has helped CS to grow from a solely mathematical research area in the mid-twentieth century, to a multi-disciplinary field that spans the entirety of the theory to end-product spectrum, employs millions worldwide, produces ubiquitous and pervasive technology, and has revolutionized every aspect of the human experience in the beginning of the twenty-first century.

The key insight behind software modeling in particular, and software engineering in general, is that as long as any one person is required to understand the entirety of a project, the complexity of projects that can undertaken by *homo sapiens* will be severely curtailed. The keys to surpassing said limitations are *encapsulation* and *abstraction.* Together these allow large groups of humans to collaborate on projects whose complexities are too high to be understood by any one person alone.

Carlos A. Pérez-Delgado
University of Kent, Canterbury, Kent, UK, e-mail: c.perez@kent.ac.uk

It is thus natural to expect *quantum* software modeling to similarly help quantum computation in its evolution. Today, quantum computation is studied and developed almost exclusively by very highly-trained specialists: mostly mathematicians, computer scientists, quantum chemists, and theoretical and experimental physicists.

If quantum information technologies are to achieve even a fraction of the ubiquity of their classical brethren, then the stage must be opened to a broader set of professionals. To achieve this, it will be necessary to be able to understand, and discuss, quantum software without having to delve down to the (atomic) details. Software engineers today do not (usually) develop, discuss, or analyze their work at a level of abstraction that includes half-adders, flip-flops, let alone voltages or resistances. Similarly, we will need a language that allows us to discuss quantum software that does not concern itself with Hamiltonians, unitary gates, or even quantum circuits.

In the year 2021 there aren't many, if any, large-scale quantum software projects. So it may seem premature to develop a quantum software modeling language. This brings us to next reason why software modeling languages are important: they act as an *intuition pump.* Language can indeed influence our ability to craft new ideas[2], and not just communicate them effectively[3]. All computer scientists are familiar with how different programming paradigms and their associated languages allow us think about and tackle problems in different ways. All physicists are similarly familiar with Dirac notation, and Feynman diagrams. And most mathematicians will be equally fluent in category and type theories—all to name but a few examples.

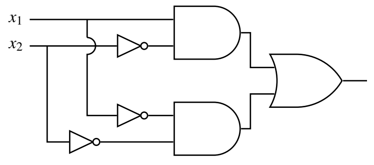
Software modeling languages have been amply credited as powerful intuition pumps in the past. Software modeling, and more generally software engineering has had a large measurable influence of lower abstraction level research in computer science, such as programming languages[7]. It therefore stands to reason that the development of a proper quantum software modeling language can also help in the development of lower abstraction level tools—like quantum programming languages.

What then does a *'proper'* quantum software modeling language look like? That is the topic of the next section.

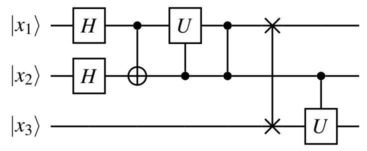
2 Fundamental Axiom of Quantum Software Engineering

A software modeling language is, above all, a *language.* As such, its utility is directly proportional to the square of the number of people that *'speak'* it. Therefore, while it may have some benefits, one should resist the temptation to start completely anew when developing a quantum software modeling language. On the contrary, it is quite clear that in order to derive the maximum possible value of a new quantum software modeling language one should aim to make it as close as possible to existing *classical* modeling languages. We can explicitly state this requirement as the first part of our Fundamental Axiom of Quantum Software Engineering: *quantum software engineering should be as similar to classical software engineering as possible.*

Is it possible then that a specific *quantum* software modeling language may be entirely superfluous, and a purely classical one would suffice? No. Quantum software



(a) Classical Logic Circuit



(b) Quantum Circuit

Fig. 1: Two logic circuits depicting, each depicting a software program. The one on the left is classical logical circuit, whereas the one on the right is a quantum circuit. Note how, at least superficially, there is little to tell each one apart.

and classical software, while they may share many important similarities, are different in fundamental ways. More importantly, they are different in fundamental ways that *need to be reflected in a design document.* In the section we will discuss exactly how so.

Before we consider how they are different, however, let us first consider how quantum and classical software are similar or even the same. What sets quantum software apart from its classical counterpart is, of course, the use of quantum algorithms. However, despite the name, quantum algorithms are not, in themselves, quantum objects. Consider Fig. 1. Part A shows a classical algorithm, while Part B shows a quantum algorithm. In both cases we have an ordered set of operations, taken from a larger set, that are applied to some data called the *input.*

What distinguishes classical circuit from quantum circuits are two things. The first is the set of permissible operations. Classical gates are usually taken to come from a standard set of universal logical bit operations, such as {AND, NOT}, or simply {NAND}. Quantum gates, on the other hand, are taken from a universal set of *unitary* operators, such as {CNOT, H, S, T}.

The second distinction is deeply connected to the first (and is what gives rise to the operator disparity): quantum and classical algorithms (and software) operate on different types of *information.* This is both an obvious and a subtle point. It is obvious that quantum algorithms (and software) operate on quantum information, while classical algorithms (and software) operates on classical information.

As a software engineer, this distinction can be treated *much* in the same way that other, more traditional, data-type distinctions are treated. In traditional software engineering it would not be uncommon to deal with modules that, say, operate on `string` or `integer` types. Specifying that a function/module/method takes as input a `string` rather than an `integer` is a well understood design decision.

However, in classical software engineering this distinction is completely *artificial.* There is, fundamentally, no real distinction between a `string` and an `integer`. Both are merely an ordered set of *bits.* It is true that we often create an abstraction layer, on top of bits, that contains objects such as `strings` and `integers`, each such object with its set of permissible operations, and so on. However, this abstraction is there only for the benefit of the programmer or software engineer.

Quantum information, on the other hand, is *fundamentally* different from classical information. Fundamental laws of physics dictate different sets of permissible operations for each. Quantum information can be put in *superposition,* classical information cannot. Classical information can be *cloned* or copied, quantum information—in general—cannot. While quantum information can be converted to classical information and *vice versa,* the operation is constrained to following fundamental physical laws, such as the *Born rule* and is often an unavoidably *lossy* conversion[1].

Hence, the software engineer's decision to have a module/function/etc. operate on quantum or classical information is a *fundamental* one. And, it has inmense repercussions. Quantum information can only be stored in a quantum module, operated by a quantum module, and can only be sent/communicated to other quantum modules. And these are all *fundamental* requirements, rather engineering limitations. Even if quantum computers become as cheap and easy to operate as classical ones, for the reasons stated above, it will *always* be necessary to distinguish the use of classical from quantum information in a software design document.

This brings us to the the second part of the Fundamental Axiom of Quantum Software Engineering which we can now state in full: quantum software engineering should be as similar to classical software engineering as possible, *but no more.*

Fundamental Axiom of Quantum Software Engineering

Quantum software engineering should be as similar to classical software engineering as possible, but no more.

In the next section we discuss precisely how quantum software modeling needs to be different from classical.

3 Design Principles for a Quantum Software modeling Language

In the previous section we discussed the fundamental axiom behind our approach to software modeling, and software engineering in general. In this section we will describe a set of *guiding principles* that we argue are both necessary and sufficient for a quantum software modeling language to achieve the aforementioned central axiom.

We will introduce five design principles. Each principle establishes a way in which *quantum* software modeling must differentiate itself from classical software modeling. For each principle, we will discuss why it is an essential feature of a quantum software modeling language. Finally, we will make the argument as to why

[1] For a full discussion on the nature of quantum information, please see a proper introductory text, such as Nielsen & Chuang's classic *'Quantum Computation and Quantum Information'*[5].

these are the *only* five ways in which a quantum software modeling language should differentiate itself from a classical one.

From here forth we will adopt the language of object-oriented design when appropriate. This is to ensure a consistent nomenclature throughout this chapter, and because we will be extending an existing classical object-oriented software modeling language in the next section.

Quantum Classes Whenever a software module makes use of quantum information, either as part of its internal state/implementation, or as part of its interface, this must be clearly established in a design document.

The first and most obvious requirement is the proper labelling of modules or classes. As discussed in the previous section, whether a particular module is *classical* or *quantum* is an important design consideration, with important ramifications. A quantum module will operate on quantum information, using quantum functions/methods. It will need to run on quantum hardware that allows for the storing of said quantum information, and is capable of executing said quantum operations. Classical modules do not have any of these requirements.

While classical modules can be, in general, run on the same quantum hardware as the quantum modules, *not* doing so offers several strong advantages.

As such, it must be explicitly specified in any design document.

Below we will discuss some guidelines that are helpful to a software designer when deciding whether a particular module is classical or quantum.

Quantum Elements Each module interface element (*e.g.* public functions/methods, public variables) and internal state variables can be either classical or quantum, and must be labelled accordingly.

- **Quantum Variables** Each variable should be labelled as classical or quantum. If the model represents data types, the variables should also specify the classical (*e.g.* integer, string) or quantum (*e.g.* qubit, qubit array, quantum graph state) data type,
- **Quantum Operations** For each operation, both the input and output should be clearly labelled as either classical or quantum. Whether the operation internally operates quantumly should also be labelled.

On a more basic level, data (variables) and operations that act on the data, are, as discussed at length in the previous section, either classical or quantum. Quantum information can generally *only* be stored in a quantum variable. And, while classical information can be stored in a quantum variable, this would be both wasteful and overly restrictive if the information to be stored is known to always be classical (*e.g.* while the information could be potentially cloned, since it is classical, the design document would imply that, in general, it cannot.)

Likewise, software operations (functions, methods) are either meant to operate on classical or quantum data, and are in general, not interchangeable. As such, it is important to label what kind of data the function takes as input, and produces as output.

Quantum Supremacy A module that has *at least* one quantum element is to be considered a quantum software module, otherwise it is a classical module. Quantum and classical modules should be clearly labelled as such.

One of the major considerations of any quantum software design is which modules are to be quantum, and which are classical. This principle states, in accordance with the central axiom stated in the previous section, that a module is to be quantum *if and only if* it contains quantum elements.

Stated differently, a *quantum* module can contain both classical and quantum (interface) elements. A classical module can only have classical elements. A module having *only* classical elements will always be a classical module unless it is *'upgraded'* by the next principle: *quantum aggregation.*

Quantum Aggregation Any module that is composed of one or more quantum modules will itself be considered a quantum module, and must be labelled as such.

Similarly to the *'quantum supremacy'* principle, if a software module aggregates (is composed of) at least one *quantum* module then it itself will also be labelled as a quantum module.

It could be argued that the previous two principles ought to be treated as a single, more general, principle: if a module uses quantum information, in any way, as part of its implementation then it is to be considered a quantum module. Otherwise, it is a classical one.

There are two reasons to state the two principles separately. First, in most software modeling languages, aggregation is considered and represented in separate and distinct ways from other internal elements. Fig. 3 gives an example of how aggregated sub-modules (sub-classes) are represented differently from other internal elements in both the quantum software modeling language Q-UML, and the original modeling language upon which it is based, UML.

A second important reason is that it allows us to explicitly make the distinction between aggregation, and *communication*, which is the next principle discussed.

Quantum Communication Quantum and classical modules can communicate with each other as long as their interfaces are compatible, *i.e.* the quantum module has classical inputs and/or outputs that can interface with the classical module.

In classical software engineering there are two different ways in which two distinct modules can interact. The first one is the aforementioned *aggregation*. This occurs when one module is subsumed as part of another module. The second is *communication.* This allows two separate modules (classes) to work together without one being an internal part of the other.

In classical software engineering there is really little distinction between the two. In either case, there are two modules that need to be aware of each other's interfaces, and are expected to *couple* or work well together. The major consideration for a software engineer when deciding whether class `B` is an internal class of `A`, or both classes `A` and `B` merely communicate with each other is whether packages/modules/classes other than `A` need to be aware of class `B`. If none do, then it makes sense to hide class `B` as an inner, aggregated, class of `A`.

In QSE there is another, more important, consideration. As noted earlier, a quantum class is one that makes use of quantum resources. It is important to note whether or not a class is quantum because that determines, among other things, what type of hardware resources are needed to run the module.

Let us suppose that class `A` is (otherwise) classical, and `B` is intrinsically quantum. By making `B` an aggregated internal class of `A`, the software engineer is making the implementer of class `A` responsible for any and all quantum resources incurred by class `B`. In short, the designer is making the statement that quantum hardware and resources are needed to implement `A`. Hence, although `A` has no quantum elements of its own, it itself becomes a quantum class.

In contrast, if the designer chooses to make both `A` and `B` distinct classes that merely communicate with one another, then class `A` can be implemented/run on fully classical hardware. Any communication between `A` and `B` must then happen through purely classical communication channels—given that `A` is classical it has no quantum interfaces and can therefore neither send nor receive quantum information messages. Class `B` is then responsible for transforming any quantum information meant for class `A` into classical (generally via measurement).

This consideration goes well beyond the differentiation between module aggregation and communication in classical software engineering; and it is a clear example of how and when QSE needs to go beyond its classical counterpart.

This concludes our discussion on the fundamental principles behind quantum software modeling language design. We present all five principles on page 8 for easy reference.

The principles discussed in this section are the immediate consequences of precisely two things. The first is the maxim we introduced in the previous section: that quantum software engineering should differ from classical software engineering only inasmuch as is absolutely necessary. The second is the intrinsic nature of quantum information, and its fundamental features that distinguish it from classical information.

These principles can—and we argue should—be applied when developing *any* kind of quantum software modeling language, regardless of its level of formality, or mathematical rigor. In the following section we will put these principles into practice with the presentation of a particular quantum software modeling language: Q-UML.

4 Q-UML

In this section we present Q-UML. Q-UML is an extension of the Unified modeling Language (UML) that allows it to properly model quantum software. It was first introduced, alongside several other ideas covered in this chapter, at the Quantum Software Engineering Workshop, of the ACM/IEEE International Conference on Software Engineering (ICSE) 2020[6].

! Quantum Software modeling Language Core Design Principles

Quantum Classes Whenever a software module makes use of quantum information, either as part of its internal state/implementation, or as part of its interface, this must be clearly established in a design document.

Quantum Elements Each module interface element (*e.g.* public functions/methods, public variables) and internal state variables can be either classical or quantum, and must be labelled accordingly.

- **Quantum Variables** Each variable should be labelled as classical or quantum. If the model represents data types, the variables should also specify the classical (*e.g.* integer, string) or quantum (*e.g.* qubit, qubit array, quantum graph state) data type,
- **Quantum Operations** For each operation, both the input and output should be clearly labelled as either classical or quantum. Whether the operation internally operates quantumly should also be labelled.

Quantum Supremacy A module that has *at least* one quantum element is to be considered a quantum software module, otherwise it is a classical module. Quantum and classical modules should be clearly labelled as such.

Quantum Aggregation Any module that is composed of one or more quantum modules will itself be considered a quantum module, and must be labelled as such.

Quantum Communication Quantum and classical modules can communicate with each other as long as their interfaces are compatible, *i.e.* the quantum module has classical inputs and/or outputs that can interface with the classical module.

UML was chosen as the *'base'* classical modeling language for this first quantum modeling language for two closely-related reasons. The first is that UML is an exceptionally easy to learn and use software modeling language, requiring very little training and background knowledge to understand. The second is its consequently large user base. By using UML as our basis, we can easily focus on developing and discussing the quantum extensions.

These extensions aim to *minimally* change base UML. A direct line can be drawn from each change to base UML to principle discussed in the previous section. The changes are also implemented in a way as to make Q-UML maximally *backwards compatible* with base UML. The goal is that for a purely classical piece of software, both the UML and Q-UML models ought to be identical. And, indeed, that is the case.

It did not, however, have to be this way, nor is this a direct result of the principles detail in the previous section, or the maxim (axiom) from two sections past. The first QSE principle discussed in the previous section states that quantum classes (modules) and classical ones need to be differentiated from one another. In Q-UML we choose to make that distinction by presenting classical classes just as they would appear in base UML, and adding new notation for quantum classes.

4.1 UML

UML (base) is a visual language, that represents and models software via diagrams. UML is agnostic with respect to programming languages tools, platforms, and software development processes. That said, UML is an *object-oriented* modeling language.

As its name suggests, UML attempts to be usable in any complex system design and engineering. In many such systems software may be merely a small component of the overall whole. There are in total 14 different types of UML diagrams, split into two general categories: *structure* and *behavior* diagrams.

Structure diagrams are used to model and represent the static elements of a (software) system. The seven structure diagram types are: *class, package, object, component, profile, composition structure,* and *deployment.*

Behavior diagrams are used to model and represent the dynamic elements of a (software) system. The seven behavior diagram types are: *state machine, use case, activity, sequence, interaction overview, communication,* and *timing*. These last four are commonly referred to as *interaction* diagrams.

Of these fourteen diagram types, the most widely used (and hence important) diagrams are: use case, class, object, state machine, sequence, and activity diagrams.

Use case diagrams are used to specify the functionality of a (software) system. Class and object diagrams, as their names suggest, show the classes and objects of the system, including their internal members, and their relationships (inheritance, aggregation, communication) with each other. State-diagrams are used to represent the intra-object dynamics of the software system. Activity diagrams represent the general logical and control flow of the entire system[2].

4.2 Q-UML Extensions

Following the aforementioned QSE axiom and our design principles, all Q-UML diagrams are identical to (base) UML diagrams, except for one thing: all instances of quantum information, whether it is being stored, communicated, or processed, are to be clearly labelled as such. Since UML is a graphical language, the Q-UML extensions are also graphical.

Generally speaking, there are two ways in which UML presents information in its diagrams. The first is pictorially. Classes and objects are represented by rectangles, in class/object diagrams, and their relationships to one another are represented via connecting lines/arrows/etc. The second is via text, usually used as labels. For instance, the names of classes and objects, and the internal members of either.

[2] There are many good introductory texts to UML. That said, the text *'UML @Classroom: An introduction to object-oriented modeling'* by Seidl *et. al.*[9] is not only excellent, it happens to cover in-depth precisely the UML diagrams discussed here.

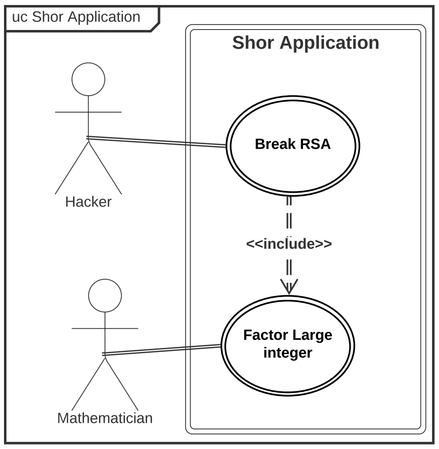


Fig. 2 Q-UML use case diagram of Shor Application. Note how any use case of the software that requires quantum resources (in this case, all of them) is distinguished by use of a double-line.

Quantum information can represented either pictorially, or textually in Q-UML. When it represented textually, quantum information will be typeset in **bold** font, to distinguish it from classical. When quantum information is represented pictorially, double lines will be used to set it apart. Whenever possible, *both* bold font *and* double lines are used to represent quantum objects/processes/*etc.* This covers the *syntax* of Q-UML.

As for the semantic rules, these follow from the previously discussed principles. All static structures and dynamic processes in Q-UML are by default classical. A static structure (*e.g.* class) is set to quantum *if and only if* it a) directly stores quantum information, or b) one of its constituent structures (a variable, aggregated class, *etc.*) stores quantum information. Likewise, a dynamic structure (*e.g.* process) is quantum if and only if it is itself a quantum information process, or one of its own sub-processes is a quantum information process.

The rules are best presented through example. For this purpose we present Shor Application: a quantum software implementation of the well-known Shor's algorithm[10]. This software system would have obvious applications in cybersecurity: Shor's algorithm can be used to easily break RSA2048 encryption. Of course, there are other more benign applications: factoring large integers is useful in many number-theoretic, combinatorial, and optimization problems. There are likely many more use cases for this software system, but we can focus on these two and provide a Q-UML *use case diagram* that details them—see Fig. 2.

Use case diagrams in Q-UML are a bit subtle and slightly different from other diagrams in their portrayal of quantum information resources. Obviously, human actors (users of the system) cannot be quantum in nature[3]. Further, no quantum communication is sent to/from the user. Rather, the double-lines in the use case

[3] Putting aside any philosophical discussions about the *quantum nature* of the Universe; here we use *quantum* as a shorthand for non-classical, or *coherent* information.

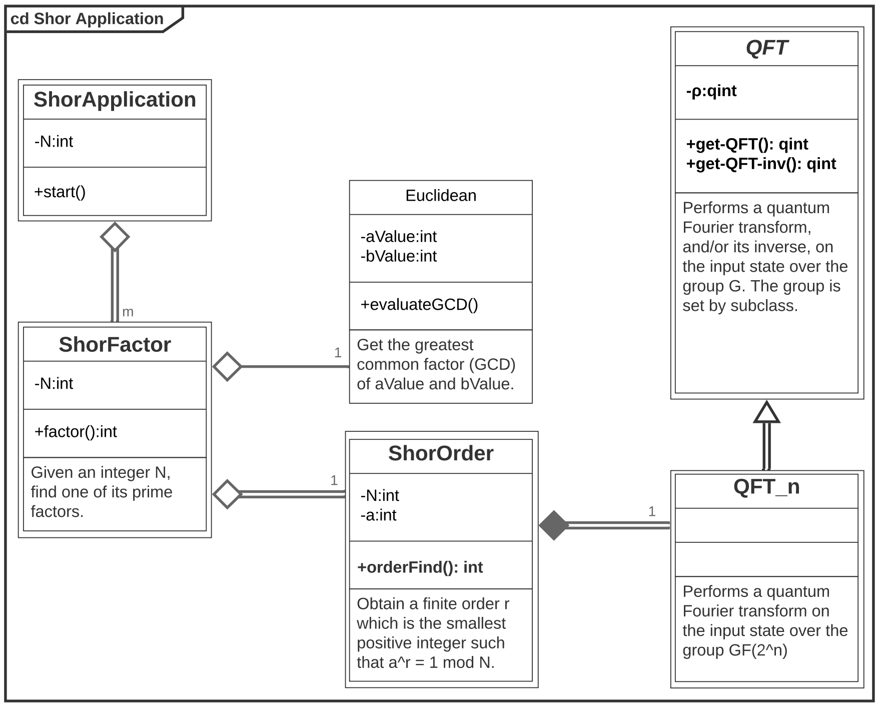

cd Shor Application
ShorApplication
-N:int
+start()
m
ShorFactor
-N:int
+factor():int
Given an integer N, find one of its prime factors.
Euclidean
-aValue:int
-bValue:int
+evaluateGCD()
Get the greatest common factor (GCD) of aValue and bValue.
1
ShorOrder
-N:int
-a:int
+orderFind(): int
Obtain a finite order r which is the smallest positive integer such that a^r = 1 mod N.
1
QFT
-ρ:qint
+get-QFT(): qint
+get-QFT-inv(): qint
Performs a quantum Fourier transform, and/or its inverse, on the input state over the group G. The group is set by subclass.
QFT_n
Performs a quantum Fourier transform on the input state over the group GF(2^n)
1

Fig. 3: Q-UML class diagram of Shor Application. Quantum classes and elements are presented in bold text, while quantum classes and relationships use double-lines.

diagram are meant to denote the *use* of quantum resources, by the software system, in satisfying the use case requirements.

A more common portrayal of quantum information in Q-UML is in class and object diagrams, which we discuss next.

4.2.1 Class and Object Diagrams

Class (object) diagrams in Q-UML, as in base UML, denote not just classes (objects) in the software system but also their internal elements, and the relationships (communication, aggregation, inheritance) between them.

Fig. 3 showcases a Q-UML class diagram of our Shor Application. We can use it to showcase the rules as they apply to both class and object diagrams. Shor Application makes use of six classes—five of them quantum. The class Euclidean is the only non-quantum one. The classes **ShorApplication, ShorFactor, ShorOrder,** and **QFT_n** are all quantum. Note how their names are all typeset in bold to emphasize this fact. The class ***QFT*** is both quantum and *abstract*; as such, its name is typeset in both

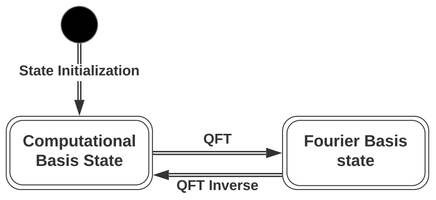


Fig. 4 Q-UML state diagram of an object of class **QFT_n**. One can tell immediately that both represented states are quantum states, and the transitions between them are mediated via quantum operators, due to the use of bold text and double-lines.

bold and italics. The later following the (base) UML rule to italicize abstract class names.

Finally, note how for all quantum classes the border of the rectangle denoting the class uses a double line. This is a departure from the previous version of Q-UML[6], which used only bold typeface to denote quantum classes.

This change was made for three reasons. The first reason is consistency. Q-UML has—*essentially*—two syntactic rules, one for pictorial and one for text representations. Since classes in class diagrams (and objects in object diagrams) are represented in *both* ways (a rectangle and a name), it makes that they follow *both* sets of rules.

The second reason is readability and clarity. It is common for the class names to be typeset with a slightly larger font than class elements. Merely using bold typeface for quantum classes may not be clear or readable enough in some conditions.

Third, the use of double-lined borders for quantum classes does not seem to add much visual clutter. Hence, both mentioned advantages can be achieved without any discernible disadvantage.

Bold text is also applied to class members. Any attributes that store quantum quantum information will have their name typeset in bold. Representing quantum methods is slightly more complex. If any of the inputs are quantum, these are bold. If the output or datatype of the method is quantum, then the datatype should also be bold. For backwards compatibility with regular UML, whenever the input or output datatypes of a method are omitted, these will be assumed to be classical in nature. In accordance with the previously established rules, if a class/object has any quantum attributes or methods then it itself is considered quantum. In this case, its name shall also be bold, and its border will use double-lines.

Finally, relationships between classes follow the same rules, using double-lines whenever the relationship is quantum in nature. For inheritance, if the superclass is quantum then the subclass, and the inheritance relationship, will also be quantum—however, the converse is not necessarily true. In the case of aggregation and composition, if a class/object being aggregated/composed is quantum, then the class/object to which it is aggregated/composed into, as well as that relationship will also be quantum.

In contrast to the above, association relationships do not have any special rules. Two classes can communicate together regardless of whether one, both, or neither is quantum. However, a classical class cannot (as already established) have the capacity to receive, send, or store quantum information. Hence, any communication between a quantum and a classical class must be through purely classical channels.

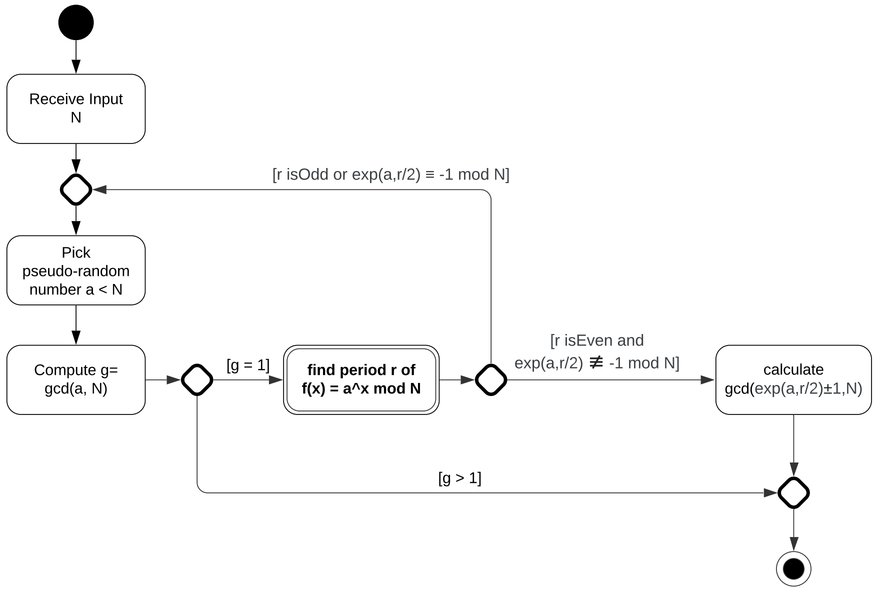

Receive Input N
[r isOdd or exp(a,r/2) ≡ -1 mod N]
Pick pseudo-random number a < N
Compute g= gcd(a, N)
[g = 1]
find period r of f(x) = a^x mod N
[r isEven and exp(a,r/2) ≢ -1 mod N]
calculate gcd(exp(a,r/2)±1,N)
[g > 1]

Fig. 5: A Q-UML activity diagram for Shor Application, showing an overview of the logical control flow of the program. The sole quantum operation is displayed using a double-lined border, and bold text.

All of these rules translate directly to objects, their members and relationships, in object diagrams. Next, we move onto the features of behavior Q-UML diagrams.

4.3 Activity and State Diagrams

Much in the same way that class and object diagrams are so closely related that they merit being discussed together, so do activity and state diagrams. However, while state and object diagrams are so similar that presenting one is sufficient to understand both, activity and state diagrams have subtle and important difference between them in Q-UML. Hence, it is important to detail them both with examples. Fig. 4 presents a state diagram of an object belonging to the class **QFT_n**. Fig. 5 presents a flowchart of the main algorithm followed by Shor Application.

Both follow the same basic rules: quantum information states and processes are denoted through the use of bold text and double-lines.

In our state diagram (Fig. 4), we have two quantum states: a computational basis state, and a Fourier basis state. Each are properly denoted as being quantum states. The operation that transforms one into the other is necessarily a quantum operator.

Hence, the transition between both states—the arrows—are denoted as quantum states through the use of double lines.

Now consider the activity diagram in Fig. 5. Once again, the only quantum operation—the period-finding step—is properly denoted as a quantum operation through the use of both double-lines and bold text. However, note the complete lack of double-lined arrows throughout the diagram. Why is this the case?

In Q-UML diagrams, as in base UML, activity diagrams represent multi-step processes. Each rectangle (node) represents an activity. And the arrows merely denote the passing of *control* from one logical activity to the next. Unlike in state diagrams, where arrows denote operations or processes that can be either classical or quantum, in activity diagrams they denote the flow of control. It would be a category error to even attempt to classify these as either classical or quantum.

This relates back to the discussion in Sec. 2 on Page 2. While algorithms may operate on classical and/or quantum information, the algorithms themselves—the logical flow-control of a program—are always classical objects.

Next, we discuss sequence diagrams in Q-UML.

4.4 Sequence Diagrams

Sequence diagrams in Q-UML, like in base UML, allow us to portray the *dynamic* relationship between modules in a software program. Fig. 6 shows a Q-UML sequence diagram for Shor Application.

Like before, we make use of **bold** text and double-lines to portray quantum information textually and pictorially. Names of quantum classes, and the labels of quantum messages are typeset in bold. The arrows depicting quantum messages use double-lines.

In another departure from the original version of Q-UML[6] we now also have the borders of quantum classes, their lifelines, active objects and threads using double-lines. Once again, this change was made for consistency, and readability.

Note that although the *relationship* between **Shorfactor** and **ShorOrder** is quantum, the messaging between them is *not.* A module is marked as quantum if it uses quantum resources in any form, either directly as part of its internal implementation or as part of an aggregated module. If an aggregated sub-class is quantum, then the encompassing class must also be marked as quantum. In a class diagram, the quantum composition relationships inform us—especially in the case of a seemingly classical module that does not *in itself* use quantum resources—which composed modules are using quantum resources.

Also, note the communication between the objects **ShorOrder** and **QFT_n**. The module **QFT_n** operates on a quantum state. Hence, both 'set' messages are quantum. Likewise, the return messages ρ and ρ' are quantum states. However, the request to perform a quantum Fourier transform (QFT) or a QFT inverse operation can (and therefore must) be communicated classically. This diagram showcases the level of granularity available to us using these diagrams with the proposed extensions.

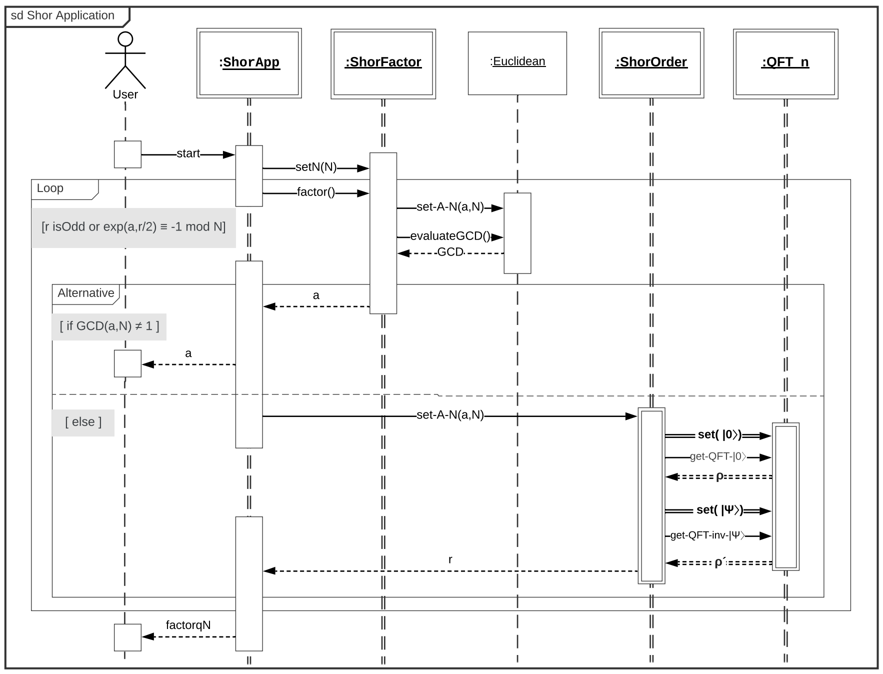


Fig. 6: Q-UML sequence diagram for Shor Application. Once again, bold text and double-lines are used to represent quantum classes/objects, their lifelines and threads, and quantum communication between them.

4.5 Discussion and Further Reading

Q-UML, its design principles, and the axioms upon which those are built were first introduced at the Quantum Software Engineering Workshop of the IEEE/ACM 42nd International Conference on Software Engineering, in 2020[6, 1]. This was among the first major international events centered around quantum software engineering.

QSE is still a very young field. Q-UML—and the work surrounding—has been from its inception an attempt to shape the future direction of this field. At its core, Q-UML is what it says it is: a modeling language for quantum software. Hopefully, it will become a pragmatically useful one: used in the design, development, and discussion of complex quantum software. This chapter serves as an introduction, for the working quantum software engineer, to Q-UML.

Hopefully, Q-UML will also serve as a template for further QSE development. Q-UML is a testbed for the axiom put forth in Sec. 2. If Q-UML succeeds as a modeling language it will prove the utility of said axiom, and the design principles that spawn from it. Regardless of the speed of adoption of Q-UML as a practical tool, its development serves one other purpose: to spark a useful discussion about the direction of QSE research and development. There is already some positive evidence of this happening[1, 4, 8].

In short, this is a very interesting and exciting time for the field of quantum software engineering. Many interesting discoveries and developments await those willing and able to search them out. Q-UML is work in *one* possible direction in this field. But it is also an attempt to argue how work in *any* direction in this new field should be conducted. For example, development of a formal specification language for quantum software (something that Q-UML is certainly *not*) could also be based on the same axiom/principles as Q-UML. And so, hopefully, the contents of this chapter are useful to any researcher doing work in this field, regardless of the direction they wish to take.

Acknowledgements The author would like to acknowledge funding through the EPSRC Quantum Communications Hub (EP/T001011/1). The author would like to thank Hector Perez-Gonzalez for his valuable work, knowledge, and insight, during the collaboration that gave birth to Q-UML, and would also like to thank Joanna I. Ziembicka for useful comments during the preparation on this manuscript.

References

1. Abreu, R., Ali, S., Yue, T.: First international workshop on quantum software engineering (q-se 2020). ACM SIGSOFT Software Engineering Notes **46**(2), 30–32 (2021)
2. Jackendoff, R.: How language helps us think. Pragmatics & Cognition **4**(1), 1–34 (1996)
3. Mercer, N.: Words and minds: How we use language to think together. Routledge (2002)
4. Moguel, E., Berrocal, J., García-Alonso, J., Murillo, J.M.: A roadmap for quantum software engineering: Applying the lessons learned from the classics. In: Q-SET@ QCE, pp. 5–13 (2020)
5. Nielsen, M.A., Chuang, I.: Quantum computation and quantum information (2002)
6. Pérez-Delgado, C.A., Perez-Gonzalez, H.G.: Towards a quantum software modeling language. In: Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops, pp. 442–444 (2020)
7. Ryder, B.G., Soffa, M.L., Burnett, M.: The impact of software engineering research on modern progamming languages. ACM Trans. Softw. Eng. Methodol. **14**(4), 431–477 (2005). DOI 10.1145/1101815.1101818. URL https://doi.org/10.1145/1101815.1101818
8. Sánchez, P., Alonso, D.: On the definition of quantum programming modules. Applied Sciences **11**(13), 5843 (2021)
9. Seidl, M., Scholz, M., Huemer, C., Kappel, G.: UML@ classroom: An introduction to object-oriented modeling. Springer (2015)
10. Shor, P.W.: Algorithms for quantum computation: Discrete logarithms and factoring. In: Proceedings 35th annual symposium on foundations of computer science, pp. 124–134. IEEE (1994)