Modelling Quantum-Classical Interactions: A Comparative Study of Q-UML and Quantum UML Profile Diagrams for the Variational Quantum Eigensolver in Qiskit

Supervisor: Carlos A. Pérez-Delgado
Programme: MSc Computer Science (Artificial Intelligence)
Word Count: 9,244

December 3, 2024

Acknowledgments

Abstract

List of Figures

List of Tables

List of Abbreviations

- AI - Artificial Intelligence
- CD - Class Diagram
- C-SE - Classical Software Engineering
- COBYLA - Constrained Optimization by Linear Approximation
- DSL - Domain-Specific Language
- GUI - Graphical User Interface
- IBM - International Business Machines Corporation
- ISA - Instruction Set Architecture
- MSc - Master of Science
- NISQ - Noisy Intermediate-Scale Quantum
- OMG - Object Management Group
- OOP - Object-oriented programming
- POP - Procedure-oriented programming
- PUB - Primitive Unified Blocs
- RSA - Rivest-Shamir-Adleman
- SD - Sequence Diagram
- SDK - Software Development Kit
- SU(2) - Special Unitary Group of Degree 2
- Q-UML - Quantum Unified Modelling Language
- Q-SE - Quantum Software Engineering
- UML - Unified Modelling Language
- VQE - Variational Quantum Eigensolver

Contents

1 Introduction

1.1 Problem Description and Incentive

Quantum Software Engineering (Q-SE) is an emerging research field seeking to develop and standardise software engineering principles in quantum technologies. Many of these standards have been adapted from classical software engineering (C-SE), attempting to keep them as familiar as possible to C-SE whilst being able to distinguish that quantum and classical systems are two fundamentally different hardware.

One area of interest in Q-SE is the adaptation of Unified Modeling Language (UML) to model quantum systems and illustrate their communication with classical systems. Two notable papers have introduced adaptations of UML for this purpose: *"A Quantum Software Modeling Language"*[38] and *"Design of classical-quantum systems with UML"*[37]. Both propose methodologies for incorporating quantum technologies into UML, aiming to broaden the scope of professionals who can contribute to the Q-SE field and promote early adoption of a standardised quantum modelling language whilst quantum technologies remain in relative infancy.

The critical question is which quantum UML adaptation offers the best solution. This involves evaluating their effectiveness for modelling quantum systems, their suitability in real-world applications and their potential for widespread adoption. Additionally, should the industry favour full-scale UML modelling or another simplified modelling approach, and which of the two adaptations best accommodates an alternative design preference?

1.2 Goals and Objectives

This paper aims to create and contrast UML diagrams using these two quantum UML approaches.

The first objective is to choose an appropriate real-world example as a starting point for the initial diagrams. The example chosen is the Variational Quantum Eigensolver (VQE), a hybrid quantum/classical algorithm. VQE is ideal for exploring how communication between quantum and classical machines can be represented in a modelling language, as it requires iterative interaction between a classical optimiser and a quantum circuit to find its solution.

The second objective is to choose the most suitable UML diagrams to model the algorithm and compare quantum UML methods. A sequence diagram was selected because it is one of the most commonly used UML diagrams and can illustrate the communication between quantum and classical elements in the VQE algorithm. In addition, class diagrams were included to provide a structural view of the VQE algorithm, offering insights into the relationships and attributes of the algorithm's components.

The third objective is to determine a robust method for comparison and analysis. This will involve combining the author's observations and following established guidance on good diagram design.

The final objective is to create multiple UML diagrams and explore other quantum applications. A key area of exploration will involve modelling fundamental quantum properties, such as entanglement and superposition, as suggested by the authors of the quantum UML literature: *"Although the scope of the paper focuses on extending UML for quantum software, it could be explored how to represent in UML other fundamental properties of quantum computers (e.g., superposition, entanglement, etc.)"*[37]. This approach will enable the application of a sequence or class diagram in a new context or using additional UML diagram types to represent these properties.

2 Literature Review

2.1 Background

This section provides historical and contextual insights relevant to the paper. It explores the inception of discussions on quantum software engineering from international workshops and examines efforts to adapt UML diagrams for quantum applications. The section includes an overview of UML's history and the implementation of sequence, class and profile UML diagrams. The section introduces quantum computing, discussing its significance, future potential, and currently available hardware. The Variational Quantum Eigensolver (VQE) algorithm is then explored, focusing on its implementation in Qiskit provided through IBM's Quantum Learning platform, which underpins the UML diagrams in this paper. Finally, the section concludes with an analysis of the two quantum UML adaptations central to the research.

2.1.1 Q-SE 2020

The Quantum Software Engineering (Q-SE) 2020 workshop marked the first international gathering aimed at fostering a community around quantum software engineering, focusing on *"devising methods, approaches, and processes to develop software for quantum programs efficiently and to ensure their correctness"*[2]. Originally scheduled to be held in Seoul, South Korea, it was adapted to a virtual format due to the COVID-19 pandemic, running over two days in July 2020. This workshop was co-located with the 42nd International Conference on Software Engineering (ICSE 2020), providing a platform for interdisciplinary collaboration within quantum and software engineering communities.

At the close of Q-SE 2020's first day, Carlos A. Pérez-Delgado presented, *"Towards a Quantum Software Modeling Language"*[39], a paper co-authored with Héctor G. Pérez-González. The collaboration began when Pérez-González, an expert in software engineering, approached Pérez-Delgado about submitting a paper to the Q-SE 2020 workshop. By combining Pérez-Delgado's expertise in quantum computing and quantum information with Pérez-González's background in software engineering, the duo developed the concept of Q-UML, which applies UML diagrams to quantum technologies[40].

2.1.2 UML

Unified Modelling Language (UML) is a general-purpose modelling language that is an industry standard for designing and documenting software systems. It is *"a consolidation of the best practices that have been established over the years in the use of modelling languages"*[54].

Modelling languages use graphical or textual notation to abstract complex technical information, facilitating communication between technical and non-technical professionals involved in information systems. Just as a blueprint simplifies the construction process by visually representing a building's structure, a modelling language provides an organised framework to convey the structure and interactions of software and hardware systems that work together to perform a task[54]. Their primary function is establishing a high-level language governed by a defined set of frameworks and rules[52].

The Unified Modeling Language (UML) framework is rooted in the object-oriented programming (OOP) paradigm. OOP is *"a computer programming model that organises software design around data, or objects, rather than functions and logic."*[23] Whereas Procedure Orientated Programming (POP) breaks down a task into smaller functions, executing them in a strict order [22], OOP encapsulates elements in a system as classes and objects which contain data (attributes) and behaviour (methods)[26]. The OOP approach facilitates a more natural, modular understanding of computing tasks inspired by real-world concepts[54]. As a small example, a `person` could be considered an instance (the object) of the class `Human`, inheriting shared attributes such as `age` and `hairColour` with each instance holding a specific attribute value such as `age = 32` and `hairColour = "blonde"`.

The development of the Object-Oriented Programming (OOP) paradigm is credited to the programming language SIMULA, which was introduced in 1962[62]. OOP quickly gained traction in the following decades, leading to the development of widely used programming languages today, such as C++, Java, and Python[54]. As OOP programming languages became more popular, there was an increasing demand for modelling languages to aid in their analysis and development. This surge in demand led to the creation of numerous modelling languages, each with its own notation, resulting in confusion and compatibility issues[54]. In response, the need for a unified modelling language arose, culminating in the creation of UML. Developed through collaborative efforts by experts in the field, UML aimed to integrate the best practices from various modelling languages[54]. Released in 1997, UML has since undergone several revisions, with the latest version, UML 2.5.1, released in 2017 and maintained by the Object Management Group (OMG)[1].

There are 14 types of UML diagrams, classified into two main categories: structural and behavioural diagrams [54]. Structural diagrams provide a static view of a system's architecture, illustrating its components' composition and relationships. Behavioural diagrams represent the dynamic interactions between system components at runtime, demonstrating how elements communicate and evolve throughout their lifecycle. The UML diagrams and their respective cat-

egories are given in Figure 1.

Two of the most widely used diagrams are the sequence diagram, which falls under the behavioural category, and the class diagram, which is part of the structural category.

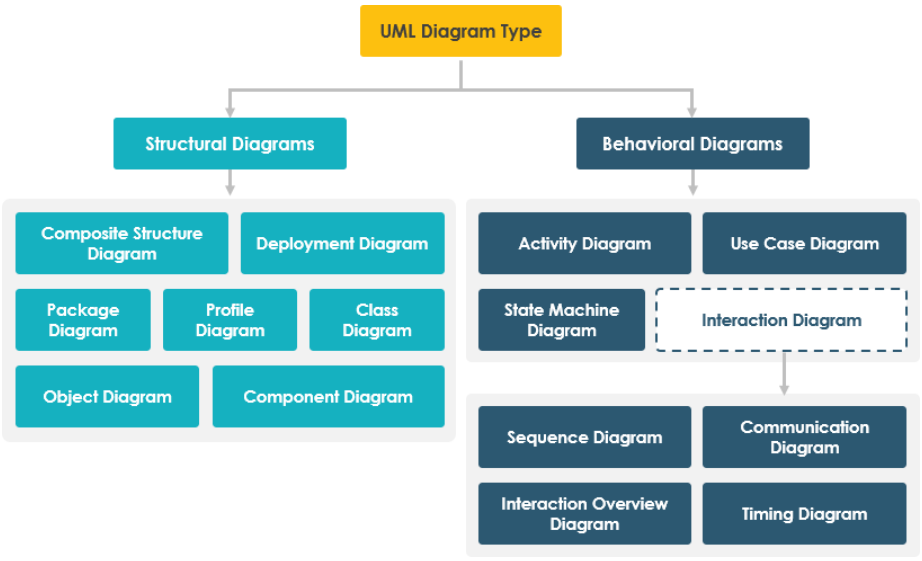

Figure 1: A map of the 14 UML diagram types, grouped into their respective categories, with an additional breakdown of the interaction diagrams subcategory[33].

2.1.2.1 Sequence Diagram

Sequence diagrams belong to a sub-category of interaction diagrams, which includes communication, timing, and interaction overview diagrams[54].

Sequence diagrams model communication protocols between human and non-human entities. The horizontal axis represents the sequence of communication messages, while the vertical axis shows the timing of interactions. Each element in a sequence diagram is represented by a lifeline extending vertically, which may terminate if it is no longer required in the system. Messages are represented by arrows connecting elements at various stages of their lifecycle.

Figure 2 presents a simplified sequence diagram that expands on the earlier analogy of a `Human` class, showcasing interactions between its `person` instances.

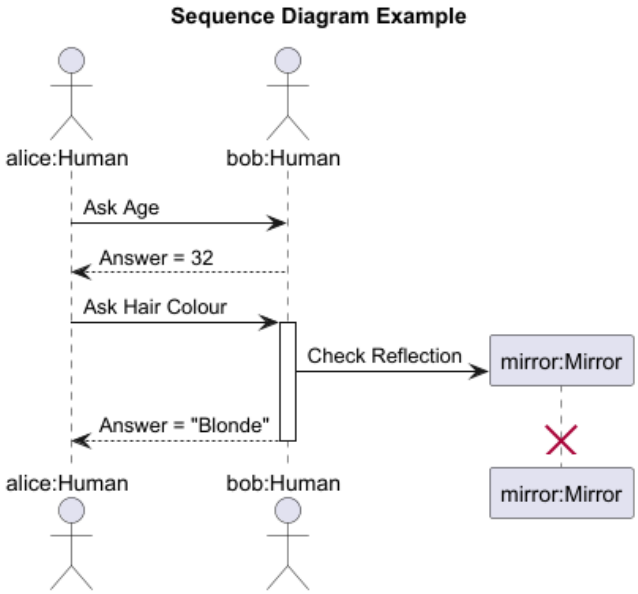

Figure 2: A simple sequence diagram: Alice asks Bob his age and receives the reply "32". Bob consults a non-human "mirror" object to answer the second query, and the mirror's lifeline terminates once it is no longer needed.

2.1.2.2 Class Diagram

The class diagram defines the structure of classes within a system[54]. Each class is depicted as a rectangular element containing information about the attributes it holds and the operations it can perform. Interconnected edges represent the associations between different classes in the system, defining their specific relationships and interactions. The class diagram provides an overview of the modelled system's structural architecture. Figure 3 continues with the `Human` class analogy to give a simplified class diagram.

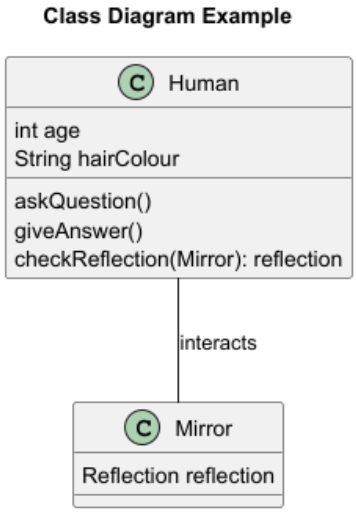

Figure 3: A simple class diagram: The `Human` class has attributes `age` and `hairColour` and can perform the operations `askQuestion()`, `giveAnswer()`, and `checkReflection()` if given a mirror. It has an "interacts" association with a `Mirror` class, which returns its `reflection` attribute when `checkReflection()` is called.

2.1.2.3 Profile Diagram

The profile diagram is another structural diagram used to modify general-purpose UML diagrams to become domain-specific.

The composition of a UML diagram is modelled using UML itself. Class diagrams illustrate the structure of each of the 14 UML diagram types, including their own structure. Together, these class diagrams form the UML meta-model[54].

There are three methods for tailoring the UML meta-model to a specific domain, with the appropriate approach chosen based on the required level of specificity. These methods, in order of decreasing intensity, are creating an entirely new meta-model for the domain, modifying the existing UML meta-model by introducing new meta-classes and meta-associations or leveraging UML's built-in extension mechanism—the UML profile diagram[54].

The profile diagram uses stereotypes, a type of meta-class, to modify existing meta-classes in the meta-model. For example, the meta-classes **Association** and **Class** are part of the class diagram meta-model, while **Message** and **Lifeline** are used in the sequence diagram meta-model. Stereotypes are denoted by the tag `<<Stereotype>>`, followed by their name. These stereotypes are connected to the meta-classes they modify through an extension relationship pointing from the stereotype to the meta-class it's modifying[54].

A stereotype can include meta-attributes or carry a note. Notes typically

provide additional context or clarification in natural language. The stereotype name, meta-attributes, and notes impose constraints on the meta-class being extended, thus refining the system's structure and behaviour by introducing domain-specific constraints[54].

Figure 4 illustrates how the profile diagram constrains the class diagram meta-model by applying stereotypes to its meta-classes. Figure 5 demonstrates how the class diagram is constructed using the stereotypes defined in Figure 4.

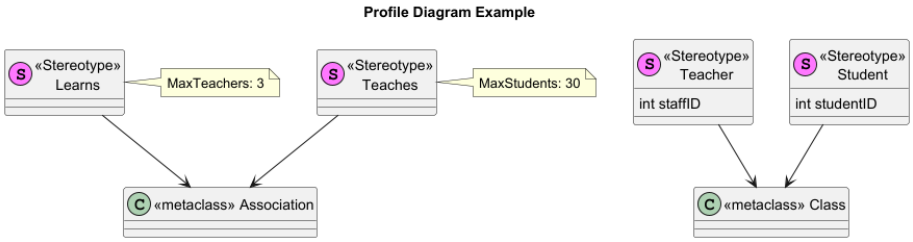

Figure 4: A simple profile diagram applied to the class diagram meta-model. The stereotypes `<<Teacher>>` and `<<Student>>` are applied to classes, each containing meta-attributes related to their respective IDs. The stereotypes `Learns` and `Teaches` are assigned to associations, with notes specifying limits on how many "teaches" or "learns" associations can exist within a system.

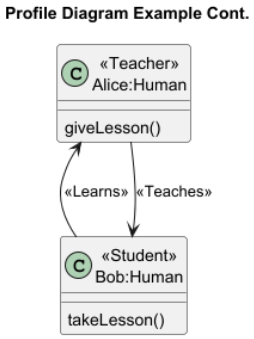

Figure 5: A class diagram implementing the profile diagram stereotypes. Classes and associations are tagged with their respective stereotypes, inheriting the constraints defined in Figure 4.

2.1.3 Q-SE 2021

The second Q-SE workshop held virtually in June 2021 included a segment focused on modelling quantum systems[3]. During this session, Luis Jiménez-Navajas presented the paper *"Modelling Quantum Circuits with UML"*[36], co-

authored with Ricardo Pérez-Castillo and Mario Piattini. The paper introduced the concept of creating a UML profile diagram that could incorporate a quantum domain, using the example of an activity diagram to model a quantum circuit[36].

Pérez-Castillo and Piattini further expanded on this work and, in 2022, published the paper *"Design of Classical-Quantum Systems with UML"*[37] in the Springer May 2022 *Computing* journal[6]. They also served as editors for *"Quantum Software Engineering"*[55], also published by Springer, which included a revision of Pérez-Delgado's Q-UML work. The follow-up paper, *"A Quantum Software Modeling Language"*[38], further developed Q-UML by introducing additional pictorial elements for modelling quantum components and establishing fundamental axioms and core design principles.

"Design of Classical-Quantum Systems with UML"[37] built upon earlier efforts to develop a quantum UML profile diagram by extending its application to multiple UML diagram types.

Pérez-Delgado's work with Q-UML modelled Shor's algorithm in the context of a quantum computer exhibiting quantum advantage, while Pérez-Castillo and Piattini modelled hybrid information systems. These hybrid systems, which combine classical and quantum processors, are already operational and can be executed on existing quantum hardware.

2.1.4 NISQ

As the introduction mentions, classical and quantum computers differ fundamentally in their underlying hardware. Classical computers operate in a binary, deterministic state where bits are clearly defined as 0 or 1. In contrast, quantum computers leverage the principles of quantum mechanics, using qubits instead of bits. A qubit can exist in a superposition of 0 and 1, with its final state determined probabilistically when measured.

The ability for qubits to exist in superpositions allows quantum computers to accelerate the speed at which certain types of problems can be solved exponentially[57]. A well-known example is factoring large prime numbers, a computational task that classical computers find difficult to solve within a reasonable time frame. Peter Shor's development of Shor's algorithm demonstrates how a sufficiently powerful quantum computer could theoretically perform prime factorisation significantly faster than its classical counterpart[56][51]. The Rivest-Shamir-Adleman (RSA) encryption algorithm is a widely used encryption protocol which relies on classical computers' difficulty in factoring large prime numbers[7]. A powerful enough quantum computer could pose a risk to modern-day security protocols as we currently know them.

Quantum computing aspires to reach the same level of ubiquity as classical computing while achieving quantum supremacy and quantum advantage. Quantum supremacy would enable quantum computers to solve tasks that classical computers cannot complete within a reasonable time frame, such as factorisation of large prime numbers[49]. Quantum advantage would allow quantum systems to solve real-world problems *"faster than any classical algorithm run-*

ning on any classical computer"[49]. To reach this point, quantum computers must scale up the number of qubits required for complex calculations—a critical challenge they face today.

Qubits are susceptible to environmental decoherence and noise, which can lead to errors and affect their ability to retain their quantum state[27]. Despite notable progress in increasing qubit counts, significantly more qubits are needed to enable error-correction methods for fault-tolerant quantum computing required for quantum advantage and quantum supremacy.[31].

The Noisy Intermediate-Scale Quantum (NISQ) era, representing near-term quantum computing, refers to hardware with qubit counts ranging from tens to hundreds[48]. This technology is currently accessible, with IBM's Qiskit Runtime Service as an example of NISQ hardware utilisation[25]. Despite their limited qubit capacity, these systems support computations through hybrid algorithms that leverage quantum and classical resources. One such algorithm is the Variational Quantum Eigensolver, which enables practical problem-solving within NISQ constraints.

2.1.5 VQE

The Variational Quantum Eigensolver (VQE) is a hybrid quantum-classical algorithm that can be used to find the ground state, the lowest energy, of a given physical system[63]. VQE can aid in quantum chemistry simulations where determining the ground state of a given molecule or atom *"provides essential information about the system's stability, reactivity, and other chemical properties"*[47]. VQE has many potential applications but can be considered a viable candidate for optimisation problems; it can be used to find a global or best local minimum (or maximum) solution within a search space[43][21].

VQE belongs to a class of near-term algorithms designed to run on NISQ devices. As the depth of a quantum circuit increases, error rates also rise due to decoherence, requiring more qubits for error correction. Near-term algorithms seek techniques for solving non-trivial problems using limited qubit count whilst negating decoherence[24]. VQE uses a shallow quantum circuit and delegates tasks between the quantum circuit and a classical optimiser to find its solution[43][41].

The VQE algorithm contains the following core components:

Hamiltonian - A matrix that represents the total energy of a physical system[44]. The eigenvalues of the Hamiltonian correspond to the system's energy levels, with the lowest eigenvalue representing the system's ground state[43].

Ansatz - A trial state for the Hamiltonian that represents an educated guess for finding the Hamiltonian's approximate ground state. The ansatz is a parametrised quantum circuit, with its parameters iteratively updated and the energy estimate evaluated against the Hamiltonian to find the lowest energy estimate[46] [60] [29].

Cost Function - A function that defines the objective of the problem, whether it involves minimisation or maximisation. The cost function executes

the parametrised ansatz quantum circuit to estimate the energy of the Hamiltonian[29][43].

Optimiser - A classical optimiser that evaluates the output of the cost function and iteratively adjusts the ansatz parameters, seeking to find the optimal set of parameters for the problem solution[29][43].

The pseudocode for the VQE algorithm is as follows:

Algorithm 1 Variational Quantum Eigensolver (VQE)

1: Define Hamiltonian $\mathcal{H}$
2: Prepare ansatz $\mathcal{A}$ with k parameters $\overrightarrow{\theta}$
3: Define cost function $\mathcal{C}$ as either minimisation ($\bigwedge$) or maximisation ($\bigvee$)
4: Initialise a classical optimizer O
5: **while** convergence criterion not met and max iterations not exceeded **do**
6: Calculate expectation value E by evaluating $\mathcal{A}(\overrightarrow{\theta})$ with $\mathcal{H}$
7: Update $\overrightarrow{\theta}$ using O to optimise E
8: **return** optimal parameters $\overrightarrow{\theta}$ and solution E

The VQE algorithm can be implemented and executed using quantum programming languages, with IBM's Qiskit SDK being a popular choice.

2.1.6 Qiskit

Qiskit is an open-source software development kit (SDK) created by IBM to access and utilise their cloud-based quantum computing services. Written in Python, Qiskit provides tools and libraries for quantum programming, simulation, and experimentation.

IBM Quantum Learning offers online tutorials[29] that guide users through implementing quantum algorithms with the Qiskit SDK. The UML diagrams created for this paper are based on the IBM tutorial demonstrating the VQE algorithm in Qiskit[30].

2.1.6.1 Qiskit and VQE

This section aims to provide the reader with a comprehensive overview of the specific implementation of VQE following the Qiskit IBM tutorial[29]. The intention is to enable a thorough understanding of the associated UML diagrams. The code for this tutorial is provided in the appendix.

The instance **hamiltonian** is initialised from the class **SparsePauliOp** by calling the *.from_list()* method in its construction. The **hamiltonian** object is a classical representation of Pauli operators, where each operator is given as a string (e.g., "X", "Y", "Z", or "I" for identity). Pauli operators are 2×2 matrices corresponding to spin measurements along the x, y, and z axes[9]. Each operator string specifies actions on individual qubits. The VQE tutorial uses a 2-qubit system, acting on pairs of Pauli strings referred to as operator terms. The tensor product of these pairs is assigned a coefficient (represented as a

complex number in Python) that defines the strength of each operator term. The linear combination of these terms represents the system's total energy, the Hamiltonian. Only non-zero operators and coefficients are stored, resulting in a sparse representation of operator terms and the Hamiltonian as a whole to reduce computational expense[28][19][14].

The instance **ansatz** is initialised from the class **EfficientSU2** with the number of qubits **hamiltonian** holds passed to it in its construction. The **EfficientSU2** class provides a hardware-efficient classical representation of a quantum circuit capable of creating parametrised quantum states[46]. The circuit comprises layers of single-qubit operations along with C-NOT gates, which entangle the qubits. The Qiskit documentation defines SU(2) as *"the special unitary group of degree 2, its elements are 2×2 unitary matrices with determinant 1, such as the Pauli rotation gates"*[11], meaning that the circuit includes layers of operations that rotate the states of individual qubits, specifically using Pauli rotation gates. Each Pauli rotation gate holds a parameter which will be iteratively adjusted to find the lowest energy state of **ansatz**.

The instance **backend** is initialised from the class **QiskitRuntimeService**. The **QiskitRuntimeService** class interacts with the IBM Qiskit Runtime Service that provides cloud-based access to quantum hardware and quantum simulators. Creating an IBM account and giving a token when executing the code to access the service is necessary. The parameter **ibm_quantum** is given to **backend** during its construction to access the quantum computing platform available on the IBM cloud service. The method *least.busy()* is used to select the next available quantum hardware with the parameter **simulator** set to false to access the quantum hardware as opposed to a quantum simulator.

QiskitRuntimeService will instantiate an **IBMBackend** object which interacts with the selected quantum hardware. The attribute **target** of the **IBMBackend** object is accessed and passed as a parameter to the instance **pm** of the **StagedPassManager** class, created using the *generate_preset_pass_manager()* method. This process allows the pass manager to receive information regarding the constraints of the selected quantum hardware.

The pass manager will *"define a typical full compilation pipeline from an abstract virtual circuit to one that is optimized and capable of running on the specified backend"*[20]. The *.run()* method is executed on **pm** to transform **ansatz** to be compatible with the selected quantum hardware's instruction set architecture (ISA)[45], with the newly transformed ansatz stored in a new variable **ansatz_isa**. The *.apply_layout()* method is then called on **hamiltonian**, with **ansatz_isa** passed as a parameter, to modify its layout to be compatible with the selected quantum hardware ISA. The modified Hamiltonian is then stored as a new variable **hamiltonian_isa**.

A cost function method, defined as **cost_func**, is a user-defined function that facilitates access to the quantum hardware and will obtain an approximate energy estimate of the Hamiltonian. It accepts an estimator and the components of a primitive unified bloc (PUB) as its parameters.

Qiskit offers two primary primitives—"Estimator" and "Sampler"—designed to simplify foundational quantum tasks[17]. Estimator primitives *"accept com-*

binations of circuits and observables (or sweeps thereof) to estimate expectation values of the observables"[16]. The estimator accepts the quantum circuit **ansatz_isa** and the Hamiltonian **hamiltonian_isa** to produce an estimated expectation value of the Hamiltonian's energy. The estimator is defined later in the code, created by the class **EstimatorV2**. The PUB is the input given to the estimator primitive.

PUB comprises of an array of initial guesses for the parameters of the ansatz, the **hamiltonian_isa** given as a list, and the quantum circuit **ansatz_isa**. These components form a tuple and are assigned to the variable **pub**, which is given as a parameter for the estimator object's *.run()* method.

When *estimator.run()* is executed, the classical, parametrised **ansatz_isa** circuit is transpiled into a quantum circuit that is compatible with the selected quantum hardware. It is then executed on the quantum computer provided through the IBM cloud service, preparing a quantum state based on its current parameters[12][29]. The **hamiltonian_isa** list, which represents the system's Hamiltonian, is applied to this prepared quantum state. This guides the estimator in computing the energy expectation value of the system being modelled through the **ansatz_isa** circuit. Once the operation is complete, the *.result()* method returns a container of PUB results[15], with slicing used to access the estimated energy of the **ansatz_isa**, which is stored in the variable **energy**.

A dictionary named **cost_history_dict** stores the parameter, iteration, and energy estimate each time the estimator's *.run()* method is executed. Its initial values consist of a placeholder for the parameters, iteration set to zero and an empty list for the energy estimate, with each energy estimate being appended to the list per execution.

Finally, the cost function method returns the value of the **energy** variable as its result after being called.

A random array of initial guess parameters, assigned to the variable **x0**, is constructed using NumPy's constant π and its *random.random()* method. The code generates an array of random floating-point numbers, scaled to the range of [0,π2], to account for every quantum state that can be represented on the Bloch sphere. The **ansatz** attribute **num_parameters** sets the size of the array, having been stored earlier in the variable **num_params**, to match the number of parameters to assign to each of the Pauli rotation gates in the ansatz circuit. The **x0** array will be given to **cost_func** as the parameters required for the **pub** variable and updated iteratively to find the set of parameters that produce the lowest energy estimate of the system.

The instance **session** is initialised from the class **Session** with **backend** passed as a parameter to configure it to the selected quantum hardware. The instance **estimator** is then initialised from the class **EstimatorV2** with **session** passed to the estimator's **mode** attribute. This estimator, which operates within **cost_func**, uses **session** to execute computations on the specified quantum backend. Assigning a **Session** object as the estimator's **mode** facilitates the grouping of iterative calls to the quantum computer[18] when *estimator.run()* executes, efficiently managing the allocation of jobs to quantum resources.

The instance **res** of the classical **Minimize** function from the SciPy package is initialised with the Constrained Optimization by Linear Approximation (COBYLA) method, which is used to minimise a scalar function[53]. In this case, the scalar function is the **cost_func** method, which returns a scalar energy estimate as a float. The parameters for **res** include the **cost_func**, the initial guess array **x0** for the ansatz parameters, along with **ansatz_isa**, **hamiltonian_isa**, and **estimator**, all passed from **res** to **cost_func** for execution.

During the optimisation loop, **cost_func** calculates and returns the energy estimate of **ansatz_isa**, while **res** iteratively updates the parameters in **x0**, adjusting the ansatz's parameters to minimize the energy estimate. Each call to **cost_func** invokes the *estimator.run()* method and performs 10,000 shots on the quantum circuit before returning the energy estimate. The loop continues until the energy estimate converges to the lowest achievable value, representing the ground-state energy[29].

Upon completion of the optimisation loop, successful termination of the process is verified by comparing the solution parameter and evaluation count against the stored solution parameter and iteration count within **cost_history_dict** maintained by the **cost_func** method. These results are then visualised using the matplotlib package, plotting a graph with the number of iterations on the x-axis and the energy estimates on the y-axis.

2.2 Related Work

This section explores the two quantum adaptations to UML outlined in the Background. It summarises each paper's central argument and details the methodologies proposed for modifying UML or introducing extensions to accommodate quantum technology. These methodologies will be further examined and applied to the VQE algorithm, as detailed in the Design and Implementation section.

2.2.1 Q-UML

The paper *"A Quantum Software Modeling Language"* highlights the significant role modelling languages have played in enabling professionals from diverse disciplines to advance classical software engineering. This collaboration has been instrumental in making classical computers a ubiquitous part of modern life. Pérez-Delgado argues that while modelling large-scale quantum systems is not a pressing need, the introduction of a quantum modelling language could act as an *intuition pump*, accelerating the development of quantum software engineering to achieve the maturity and integration of its classical counterpart[38].

Pérez-Delgado defines one fundamental axiom and five core design principles when constructing a quantum software modelling language.

The fundamental axiom is that *"quantum software engineering should be as similar to classical software engineering as possible, but no more"*[38]. A quantum modelling language must allow professionals familiar with classical software modelling techniques to adapt to its notation quickly. It must also

clearly differentiate between its classical and quantum components as they handle fundamentally different types of information: *"Quantum information can be put in superposition, classical information cannot. Classical information can be cloned or copied, quantum information—in general—cannot"*[38].

The five core design principles establish when and how classical and quantum elements must be differentiated in a quantum modelling language. They are explained as follows:

Quantum Classes: If a class/object needs to use quantum information in its design or interactions, it must be labelled quantum.

Quantum Elements: Operations and attributes must be defined as either classical or quantum. The data types of variables must be labelled. The input and output of operations must be labelled.

Quantum Supremacy: If an object does not use any quantum information for its design, interactions or relationships with other objects, it will always remain classical. It will be upgraded to quantum when it requires even one quantum element.

Quantum Aggregation: An object composed of at least one other quantum object will be labelled as quantum.

Quantum Communication: Quantum and classical objects can communicate with each other if the message can be translated from quantum information to classical information. Quantum and classical messages must be labelled.

The axioms and design principles are culminated and applied to the UML extension Q-UML, with the paper providing examples of its implementation by example of a use case, class, state, activity, and sequence diagram modelling Shor's algorithm[38]. Q-UML uses bold text and double lines to distinguish quantum elements in the diagram. Sections 3.4.1 and 3.5.1 delve deeper into the methodology for applying Q-UML to sequence and class diagrams.

2.2.2 Quantum UML Profile

The paper *"Design of Classical-Quantum Systems with UML"* presents a quantum UML profile designed to address the gap in analysis and design techniques for hybrid classical-quantum information systems[37].

Pérez-Castillo et al. highlights two main challenges in the development of quantum software practices: first, the ad-hoc approach to quantum programming languages, which is often *"without any prior design or modelling"*[37], and second, the necessity for classical and quantum information systems to be *"analysed and designed together"*[37].

The authors argue that classical computers will not be replaced by quantum computers, as classical machines are still more cost-effective for many tasks. Instead, classical computers will serve as coordinators, managing requests to quantum software systems[37]. This underscores the importance of developing design documentation that accommodates classical and quantum computing systems and outlines their interactions for design and analysis.

Pérez-Castillo et al. present UML as a viable tool for modelling hybrid quantum-classical systems. They reference Pérez-Delgado's Q-UML as an ex-

isting approach to modifying UML but classify it as a Domain-Specific Language (DSL) rather than a valid UML extension[37]. This distinction arises because Q-UML employs specific notations, such as double lines and bold text, which do not conform to the official UML standard. In contrast, the UML quantum profile diagrams outlined in the paper are considered UML-compliant, adhering to the existing UML specifications[37].

Pérez-Castillo et al. introduce a UML profile for various UML diagram types, including use case, class, sequence, activity, and deployment diagrams. This paper will focus specifically on the class and sequence UML profile diagrams.

2.2.2.1 Class Diagram Meta-Model

The class diagram meta-model contains the stereotypes `<<Quantum>>`, `<<Quantum Driver>>`, and `<<Quantum Request>>`.

The `<<Quantum>>` stereotype constrains the meta-classes **Package** and **Class**. It is used to model classes and packages that enable quantum functionality such as *"quantum circuits, algorithms, or similar artefacts"*[37].

The `<<Quantum Driver>>` stereotype constrains the **Class** meta-class. It is used to model classes that manage communication with quantum software[37].

The `<<Quantum Request>>` stereotype constrains the meta-classes **AssociationClass**, **Class**, **Dependency**, and **Operation**. It is an optional modelling choice used to represent calls from a `<<Quantum Driver>>` to a `<<Quantum>>` class, linking associated elements involved in the communication, such as cost functions and classical optimisers. It can also be used to model quantum operations within a class[37].

Figure 6 shows the class diagram meta-model with the application of the quantum UML profile extensions, as presented in the paper.

2.2.2.2 Sequence Diagram Meta-Model

The sequence diagram meta-model contains the stereotypes `<<Quantum Driver>>`, `<<Quantum>>`, `<<Quantum Request>>`, `<<Quantum Reply>>`, and `<<Quantum Computer>>`.

The `<<Quantum Driver>>` and `<<Quantum>>` stereotypes both constrain the **Lifeline** meta-class. They align with the definitions established in the class diagram meta-model: `<<Quantum Driver>>` represents elements that enable communication with quantum software. The `<<Quantum>>` stereotype represents elements that provide quantum functionality.

The `<<Quantum Request>>` stereotype constrains the meta-class **Lifeline**, modelling elements created or linked by the call between a `<<Quantum Driver>>` and a `<<Quantum>>` element[37].

The `<<Quantum Request>>` and `<<Quantum Reply>>` stereotypes constrain the meta-class **Message**, modelling communications that either send or receive quantum information[37].

The `<<Quantum Computer>>` stereotype constrains the meta-class **Actor**, which denotes quantum machines if modelled in the diagram.

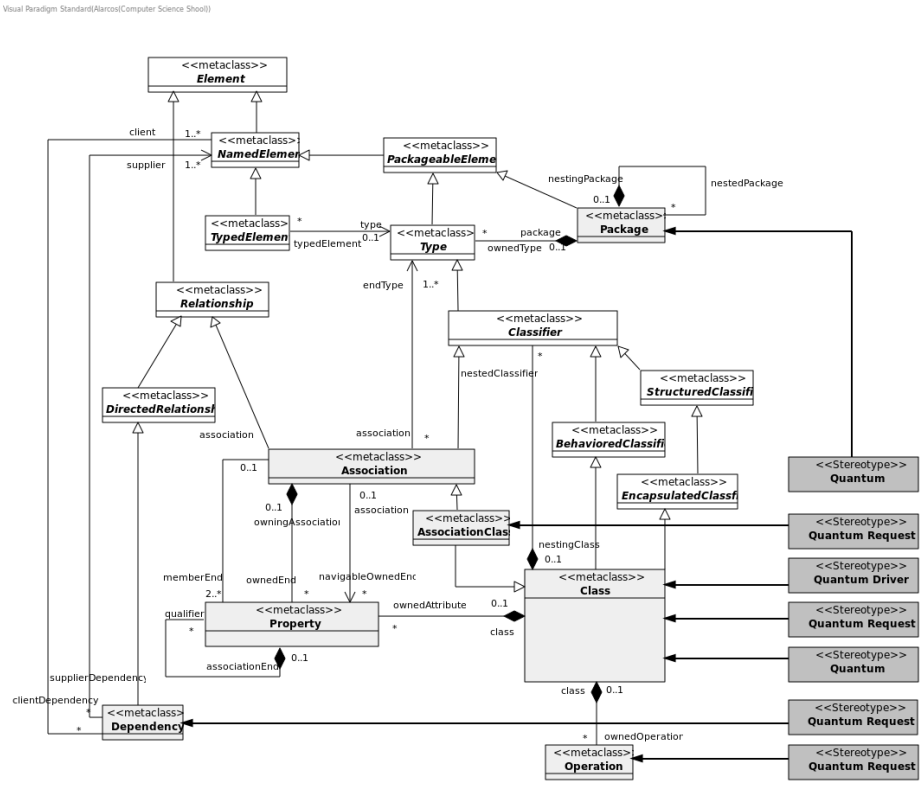

Figure 6: The quantum UML profile applied to the class diagram meta-model[35]

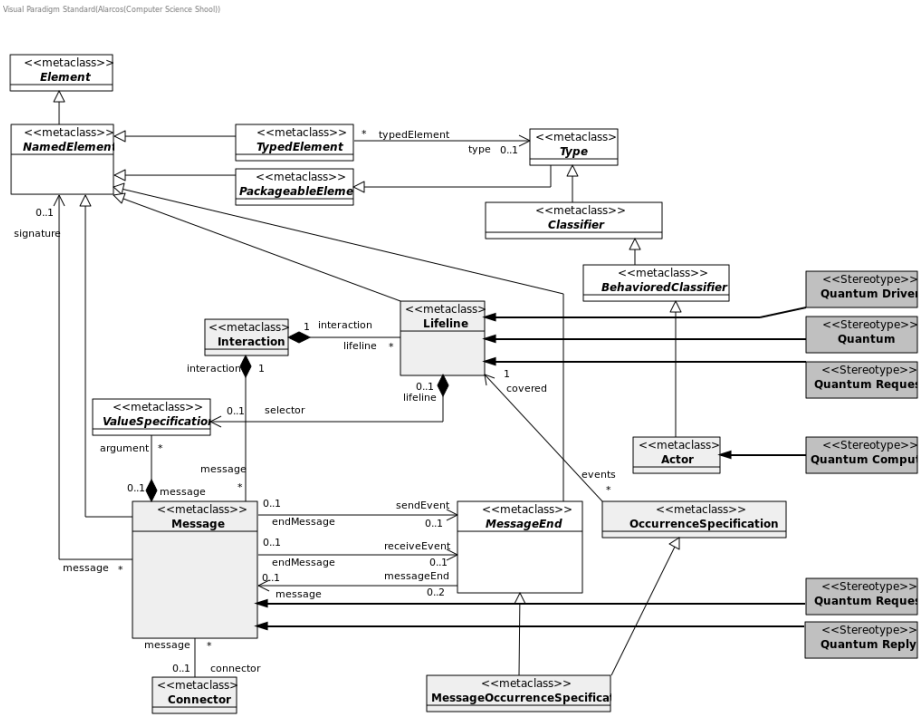

Figure 7: The quantum UML profile applied to the sequence diagram meta-model[35]

Figure 7 shows the sequence diagram meta-model with the application of the quantum UML profile extensions, as presented in the paper.

3 Design and Implementation

This section details the creation of UML sequence and class diagrams used to model the VQE algorithm, as implemented on IBM's Quantum Learning platform. It begins with an overview of the tools employed for diagram creation, followed by a step-by-step breakdown of the diagram construction process. General notation conventions for all diagrams are outlined first, followed by a detailed explanation of how the VQE algorithm is represented in the sequence diagram. Subsequent sections explore how these diagrams are adapted into the Q-UML and quantum UML profile formats. The same approach is then applied to the VQE algorithm represented as a class diagram.

3.1 Plant UML

PlantUML is an open-source tool for creating various diagram types, including UML. It enables users to generate diagrams using plain text descriptions, which are then rendered into a graphical output[42][61]. The diagrams for this paper were initially drafted using the PlantUML plug-in for PyCharm.

3.2 Lucidchart

Lucidchart is a web-based application that supports creating a wide range of diagrams, including UML. It offers a user-friendly graphical interface for customisation and flexibility. To accommodate the Q-UML adaptations—such as the double-lined pictorial elements, which are not natively supported in PlantUML—the diagrams originally drafted in PlantUML were replicated in Lucidchart. This allowed for the necessary visual modifications and ensured consistency in comparison across both diagram adaptations.

3.3 UML Design Choice

UML offers users flexibility in choosing the level of detail required to represent system information within a diagram. It is essential to strike a balance between including sufficient information for clarity and avoiding over-complication, as UML diagrams should provide a high-level abstraction of the system. This section outlines the general formatting principles applied to all UML diagrams in this project. Subsequent sections on the VQE sequence and class diagrams will delve into specific design choices and their underlying rationale. Much of the reference material for constructing these diagrams has been sourced from the book UML @ Classroom[54].

All UML diagrams consist of a content area populated by boxes and edges, which together form the specific design of each diagram type. An optional framing element was included in the UML diagrams for this project, enclosing these components within a boundary. The frame header displays the namespace of the system the diagram represents[58]. For this project, the namespace "Variational Quantum Eigensolver" is used, with abbreviations indicating the type of diagram preceding it.

A sequence and class diagram have been created to model Qiskit's implementation of the VQE algorithm. Additional versions of these diagrams have been adapted to illustrate the QUML and Quantum UML Profile extensions.

3.4 VQE Sequence Diagram

The elements in the sequence diagram represent instances of classes from Qiskit's implementation of the VQE algorithm. Each instance is depicted as a rectangle, with a dashed, vertical lifeline extending downward from its creation to connect to a duplicated element at the bottom of the frame, where all the system elements are aligned. The naming convention follows the format *instance:Class*.

Message sequences are simplified where possible to illustrate the provision of an attribute from one instance to another or to illustrate an operation's execution. For example, the message *.num_qubits* passed from the **hamiltonian** to the **ansatz** could have been a sequence of messages where the **ansatz** first requests the **.num_qubits** attribute from the **hamiltonian** and the **hamiltonian** provides it as a response message, depicted by a dashed line and open arrowhead. This is more accurate; however, it would result in an over-complicated diagram of many messages.

All messages in the diagram are synchronous, pointing from a sender to a receiver. A synchronous message indicates that the message must be received before the sender can continue with any further instructions[54]. For example, the instance **pm:PassManager** cannot execute its *pm.run(ansatz)* message until **backend:QiskitRuntimeService** passes its constraints and optimisation level. A continuous line and a filled triangular arrowhead depict a synchronous message.

A decision was made to omit the creation of an explicit **IBMBackend** element in the sequence diagram, opting instead to show messages invoking this object as originating from **backend:QiskitRuntimeService**. The naming convention effectively conveys that **QiskitRuntimeService** serves as a "backend" instance. Adding **IBMBackend** would introduce unnecessary complexity to the sequence diagram, where the emphasis is on message flow rather than object details. These specifics are more appropriately captured in the class diagram, where **IBMBackend** has been included.

The transformation of **hamiltonian** and **ansatz** to **hamiltonian_isa** and **ansatz_isa** is depicted by a destruction event; a red cross on the lifeline at the point where **hamiltonian** and **ansatz** are no longer used in the system.

Activation bars depict the activation of multiple elements within the diagram when an operation is executed[8]. This occurs when the pass manager executes its *run.()* method, the **hamiltonian** executes *apply_layout(layout=ansatz_isa.layout)* and the execution of both **cost_func** and **res**. Multiple elements within the system must be active to execute these operations.

The diagram uses two *loop* fragments and an *alt* fragment. The loop fragment expresses a sequence that is repeatedly executed[54] with a boundary encompassing the messages involved in the repeated sequence. In the VQE sequence diagram, an outer loop fragment depicts the repeated exchange of messages between **cost_func** and **res** as they seek to find the lowest energy estimate. An inner loop fragment depicts the repeated runs of the quantum circuit when *estimator.run()* is executed. The upper right corner of the fragment contains a heading of the fragments label and a description of how long the process executes; the outer loop running until [**lowest energy estimate found**] and the inner loop running [**10,000 shots**]. The response message from the estimator is shown outside of this loop, as it does not return a result after each shot but once the circuit has completed 10,000 shots.

The alt fragment in UML represents alternative sequences[54]. Here, it evaluates whether the results of the **cost_history_dict** match those from the completed minimisation routine. A true or false boolean outcome determines

whether the verification is successful or unsuccessful.

3.4.1 SD QUML

The elements **backend**, **res**, **estimator** and **cost_func** are depicted as quantum through the use of bold text for their names, double lines for their rectangle elements and double lines for their lifelines.

The instance **backend** facilitates communication between quantum and classical machines. Although the messages it sends in the diagram remain classical, the design of the class from which it is called, **QiskitRuntimeService**, is fundamentally quantum, following the first design principle **Quantum Classes**. The **estimator** element is also the instance of a quantum class that executes the ansatz circuit through its *.run()* operation. The **cost_func** element contains **estimator** in its design and is a user-define class in the context of this model. It, therefore, also follows the first design principle. The **res** element is a classical object that must be passed the **cost_func** as parameters for its execution. As it contains at least one quantum element, it is upgraded to a quantum label, following the third design principle **Quantum Supremacy**.

The sequence diagram includes one quantum message: **cost_func** sends a request to **estimator** invoking its *.run()* method on the PUB object at that iteration. This request triggers **estimator** to access quantum hardware and compute the energy estimate using quantum information based on the current parameters. The quantum nature of this message is denoted using double lines for the synchronous message, with the message details displayed in bold text. The reply message from **estimator** remains classical as the results are converted to classical information before being returned to **cost_func**.

The corresponding diagram is presented in Figure 8.

3.4.2 SD Quantum UML Profile

The **backend** element, responsible for facilitating communication with quantum software, is assigned the stereotype `<<Quantum Driver>>`.

The **estimator** element, whose functionality executes quantum circuits, is assigned the stereotype `<<Quantum>>`.

The **cost_func** element is directly tied to the interaction where a `<<Quantum Driver>>` invokes the `<<Quantum>>` element. It includes the operation *estimator.run()* to execute this call, making it appropriate to assign the stereotype `<<Quantum Request>>`. The classical optimiser, **res**, is involved in this process but does not directly call *estimator.run()*. Instead, it uses the classical output from **cost_func** and provides classical input back to it. As its interactions remain classical, **res** is not assigned a quantum stereotype.

Similarly to Q-UML, the message from **cost_func** calling the estimator's *.run()* method is assigned the `<<Quantum Request>>` stereotype as it is a message containing quantum information. There is no `<<Quantum Reply>>` stereotype as the estimator converts the message back into classical information before being received by **cost_func**.

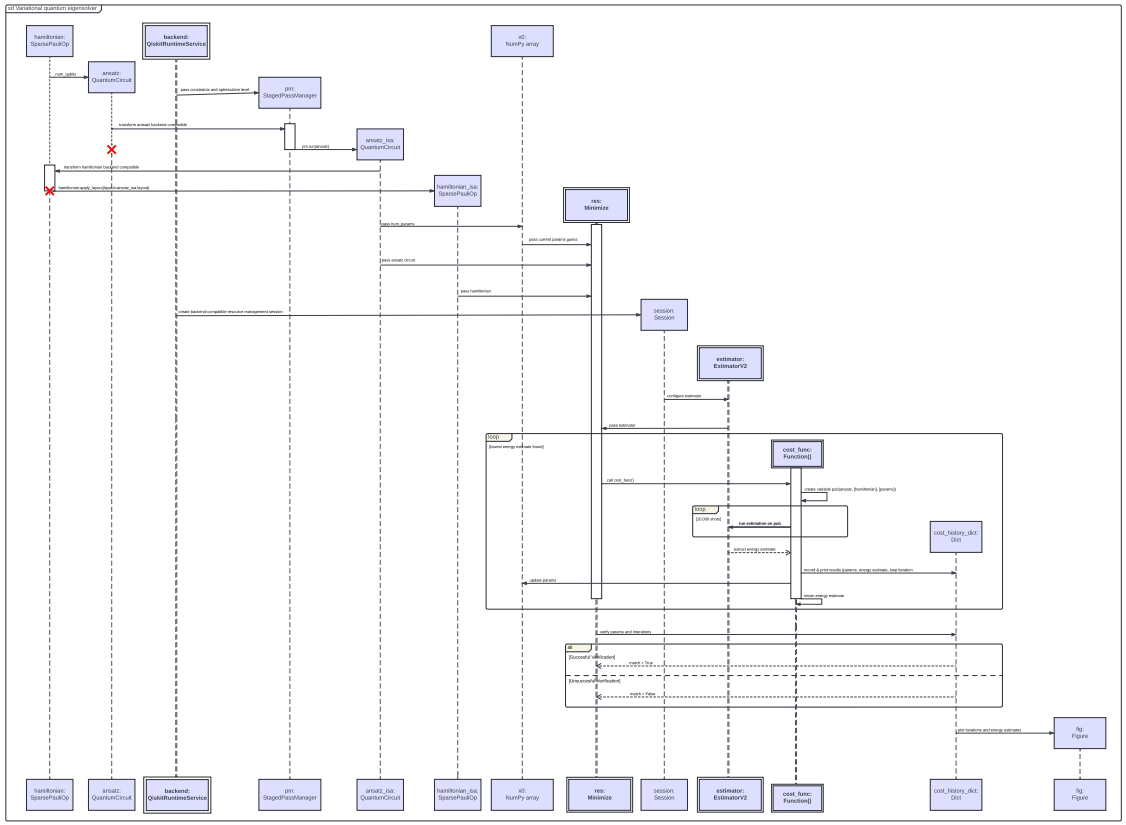

Figure 8: The application of Q-UML to a sequence diagram modelling the IBM Quantum Learning VQE Algorithm.

The corresponding diagram is presented in Figure 9.

3.5 VQE Class Diagram

Packages group the imported Python libraries used to execute the VQE algorithm in Qiskit. Although the Qiskit SDK is more than a library, encompassing a collection of libraries alongside other utilities[4], it is depicted as a single package for simplicity. Packages in UML are typically used for larger, more complex software systems[34]; however, it was elected as a design choice to provide a clear visual of the required imports to run the algorithm. The design of a package element in UML resembles the structure of loop and alt fragments in sequence diagrams, grouping related elements within defined boundaries. However, unlike loop and alt fragments, which display their names in the upper-left corner inside the boundary, a package's name appears in a smaller rectangle positioned outside of the boundary. The package **NumPy** is embedded within the Qiskit package—a design choice to prevent overlapping edges rather than imply a nested structure.

The class diagram elements, depicted as rectangles, illustrate the classes used in Qiskit's VQE algorithm implementation, with each rectangle divided into three sections: namespace, attributes, and operations. A design choice was made to keep the same notation of *instance:Class* for the namespace, as used in the sequence diagram. This notation is typically used in object diagrams. Class diagrams provide a static system view and typically only include the class name. In contrast, object diagrams capture the system at runtime and use either *instance*, *instance:Class* or *:Class* as its naming convention[54]. The decision to retain the instance in the notation is for clarity's sake, providing a minimal but effective indication of which class represents what aspect of the VQE algorithm by the instance name. Without an in-depth knowledge of Qiskit, it may be difficult to determine that **EfficentSU2** represents the ansatz or **SparsePauliOp** represents the Hamiltonian.

A design choice was also made to create a separate diagram for the **Minimize** class, distinct from the **CostFunction** class. Attempts to depict all class connections in a single diagram led to overlapping edges and clutter. The resulting diagram remains clear and readable by isolating **Minimize** and connecting it with simplified classes from the Qiskit package. The simplified classes are enclosed within the Qiskit package to indicate the part of the system to which they belong. The only simplified class not included is **CostFunction**, as it caused visual alignment issues. Since **CostFunction** is a user-defined method rather than a Qiskit class, it was decided to keep it outside the package in the minor diagram. Its meaning should still be evident within the complete diagram's context.

Only the attributes and methods relevant to the VQE algorithm are shown; additional attributes and methods documented in these classes have been omitted for clarity.

Some elements in the diagram, such as **Numpy.ndarray**, **CostFunction**, and **Dict**, are user-defined classes and therefore lack documentation detailing

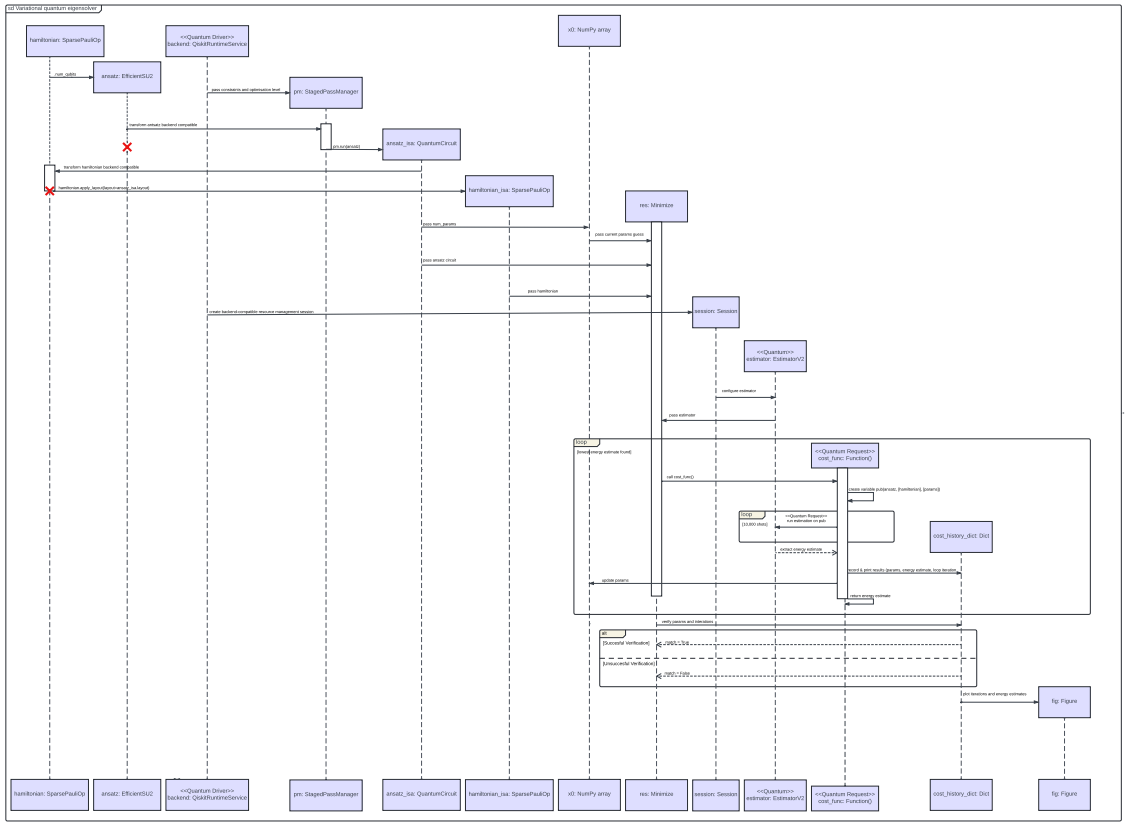

Figure 9: The application of the quantum UML profile to a sequence diagram modelling the IBM Quantum Learning VQE Algorithm.

their attributes and operations. The variables and methods within these objects are presented directly as the attributes and operations of these elements in the diagram.

Each class attribute and method contains a visibility marker and it's name with the format "visbility**name:**". The visibility of all attributes and methods within the system is public, distinguished by the character +, meaning they are accessible to all other objects within the system[54]. This was determined by none of the attributes or methods containing a leading underscore, which is the convention used in Python to define private attributes and methods[32].

Class attributes contain their datatype with the format "visbility**name:** data type". Some class attributes also include multiplicities, which can be seen in the classes **SparsePauliOp**, **EfficientSU2**, **QiskitRuntimeService**, **EstimatorV2**, **Dict**, and **Pyplot**. Attribute multiplities are used in these classes to convey how many values the attribute can hold[54], with [0...1] indicating either none or one value or [1...*] to indicate one or many values. Attributes where multiplicities are omitted are considered to have only one value. Attributes with multiplicities also have additional information as to whether the values can be duplicated with the terms "unique" or "non-unique" and whether the values must be in a fixed order with the terms "ordered" or "unordered"[54]. The format for an attribute which contains all of these details is "visbility**name:** data type[multiplicity]{duplication,order}". When an attribute's data type is a tuple, such as the **pub** attribute in **CostFunction**, each item's name and data type are specified, with square brackets enclosing the entire tuple to indicate its structure. Attributes can contain multiple data types. For example, the **mode** attribute of **Estimatorv2** can be assigned other data types than just Session. The name of the data type used in the specific algorithm is used for clarity. Including attributes is optional, with some classes not requiring this information, such as **StagedPassManager** and **Session**.

Class methods are shown with either empty parentheses if no parameter information is needed or with detailed parameter information listed within the parentheses. The data type of the returned value is provided after the parentheses. The typical notation for a method with parameter information is "visbility**name**(**parameter name:** parameter data type): returned data type. For methods returning tuples—such as the *from_list()* method in **SparsePauliOp**, the *run()* method in **Estimator**, and the *res()* method in **Minimize**—each item's name and data type are specified and enclosed in square brackets, mirroring the method's format in code. The methods *plot()*, *set_xlabel()*, and *set_ylabel()* in the **Pyplot** class do not have a return data type, as they do not return a value. Instead, they update a Figure object by assigning relevant information. The final output, of type Figure, is produced when the *draw()* method is executed. Some classes, like **CostFunction** and **Minimize**, are methods themselves. These classes include methods with the class instance name, indicating that the class can be executed as an operation. Including operations is optional, with some classes not requiring this information, such as **IBMBackend** and **Dict**.

Relationships, known as associations, between classes, are depicted by the

edges that join the rectangles of the diagram. Edges with a solid line and filled triangular arrowhead pointing from one class to another indicate a binary association, where one class can view the visible attributes and operations of another class, but not the other way round[54]. For example, **Session** can view the attributes and operations of **QiskitRuntimeService**, but **QiskitRuntimeService** cannot view the attributes and operations of **Session**. The edges between **SparsePauliOp** and **EfficentSU2** do not have arrowheads, meaning they are bidirectional[54]; each class can view the other's attributes and operations. As we established earlier, all attributes and operations in the system are public and can technically be accessed by all objects. In the context of the VQE algorithm, the associations indicate which objects share and which objects receive information. Specifically, **SparsePauliOp** and **EfficientSU2** provide each other with information throughout the VQE algorithm. In contrast, **QiskitRuntimeService** provides information to **Session**, but **Session** does not provide information to **QiskitRuntimeService**.

The **IBMBackend** class depends on the **QiskitRuntimeService** class. This is depicted by a dashed line with an open arrowhead pointing from the dependent object to the object it depends on. The Qiskit documentation states that IBMBackend must not be instantiated directly and instead should be interacted with using the methods in **QiskitRuntimeService**[13]. The **IBMBackend** class is included in this diagram as its documentation contains the **target** attribute, which must be passed to **StagedPassManager** to access information regarding the selected quantum hardware constraints. The relationship has the stereotype `<<instantiate>>` as **IBMBackend** requires **QiskitRuntimeService** for it's full implementation[10]

The class **Dict** is a composition of the class **CostFunction**. A composition is a binary association signifying that one class is contained as part of another class and cannot exist without it; if the aggregate(whole) object is destroyed, the contained part is also destroyed[59]. The relationship is depicted on the association edge by a filled diamond attached to the aggregate class. In this case, **Dict** is an integral part of **CostFunction** as it's written inside the method, remembering that the class **CostFunction** represents a user-defined method. If **CostFunction** were destroyed, **Dict** would also be destroyed.

Shared aggregation is similar to composition in that it signifies that one class belongs to another[54]. However, unlike a composition, the contained classes can exist outside the aggregated class. A hollow diamond attached to the aggregate class depicts the relationship on the association edge. Shared aggregation is used to describe classes that are passed as parameters to other classes in the VQE algorithm; **Numpy:ndarray**, **EfficentSU2**, **EstimatorV2**, and **SparsePauliOp** are all passed as parameters to **CostFunction** and therefore have shared aggregation. The classes along with **CostFunction** are also passed to **Minimize** as parameters and, therefore, have shared aggregation.

Some associations may be given an association name with a reading direction indicated by a filled triangular arrowhead pointing from one class to another. This arrowhead signifies the flow of the association's action, helping to clarify the direction in which information or control is passed between the classes involved.

For example, the association between **StagedPassManager** and **EfficientSU2** shows that the pass manager will transform the ansatz and not the other way around. Although the reading direction points in the same direction as the binary association in this example, it does not have to[54].

The relationships between the classes in the diagram are defined by multiplicities, which specify how many objects are involved in each association. For instance, the binary association between **StagedPassManager** and **EfficientSU2** demonstrates that one instance of **StagedPassManager** is associated with exactly one instance of **EfficientSU2**. Multiplicities in the diagram are either **1** or **1..***, with the option to label shared attributes. In the case of the **Minimize** class, it shares between one and many values of the attribute **res.x**—which contains parameters it iteratively updates—with a single **Numpy.ndarray** object. Subsequently, **Numpy.ndarray** shares only one array object with **CostFunction**. The use of multiplicities conveys a single-instance relationship between **Numpy.ndarray** and **CostFunction** while showing that **Minimize** dynamically updates and shares multiple values of **res.x** to the **Numpy.ndarray** class.

3.5.1 CD QUML

The classes **CostFunction**, **QiskitRuntimeService**, **IBMBackend**, **EstimatorV2**, and **Minimize** are depicted as quantum through the use of bold text for their names and double lines for their rectangles. This remains consistent with the sequence diagram quantum object labelling, with the inclusion of **IBMBackend** being a dependent object of **QiskitRuntimeService**.

No quantum attributes are shown, as all quantum data is translated into classical information for storage within these classes.

Quantum operations and input and output data types are highlighted in bold to indicate their quantum nature. The **QiskitRuntimeService** class includes two operations that return the class **IBMBackend** as a quantum data type. Similarly, the sole operation of the **Session** class also outputs an **IBMBackend** quantum data type. The **EstimatorV2** class features a quantum operation, *.run()*, which interacts with quantum hardware and returns an object of the class **RuntimeJobV2**. While the returned information remains classical, **RuntimeJobV2** is intrinsically tied to a quantum communication process of extracting and translating quantum information as a classical output. Consequently, **RuntimeJobV2** is classified as a quantum data type in this context, similar to the depiction of **QiskitRuntimeService** as a quantum object in the sequence diagram.

The **CostFunction** class contains itself as a quantum operation and produces a classical return value of type **numpy.float64**. Similarly, the **Minimize** class also contains itself as a quantum operation returning the classical data type **OptimizeResult**. The **Minimize** class includes parameters structured as a tuple, requiring the quantum input data type **EstimatorV2** to be provided.

A single quantum aggregation relationship exists between **EstimatorV2** and **CostFunction**, represented by a double-lined edge. In this relationship,

the **EstimatorV2** quantum class is passed as a parameter to **CostFunction**.

The corresponding diagram is presented in Figure 10.

3.5.2 CD Quantum UML Profile

The Qiskit package is assigned the stereotype `<<Quantum>` as it is a package that enables quantum functionality, allowing users to access quantum computers through the Qiskit SDK.

The classes **CostFunction**, **EstimatorV2**, and **QiskitRuntimeService** inherit the same stereotypes as those assigned to their counterpart lifelines in the sequence diagram. Additionally, the **IBMBackend** class, which depends on **QiskitRuntimeService**, is assigned the `<<Quantum Request>>` stereotype. This assignment reflects its ability to perform the same quantum-related operations as the class on which it depends.

The operation *.run()* in **EstimatorV2** and *cost_func()* in **CostFunction** are assigned the stereotype `<<Quantum Request>>`. Both operations are quantum as discussed previously; *estimator.run()* invokes a call from the `<<Quantum Driver>>` to the `<<Quantum Element>>` and **CostFunction** contains this operation in it's design.

The corresponding diagram is presented in Figure 11.

4 Results and Analysis

4.1 Diagram Construction

A combination of PlantUML and Lucidchart was required to create the Q-UML versions of the UML diagrams. While efforts were made to incorporate double-lined notation using PlantUML, a Q-UML extension for PlantUML would be necessary to create diagrams using the complete Q-UML notation.

PlantUML proved significantly easier to use for sequence diagrams compared to Lucidchart, which required extensive manual adjustments that were both cumbersome and time-consuming. The quantum UML profile for sequence diagrams could have been created entirely in PlantUML, as its notation is fully UML-compliant and supported by built-in extensions, simplifying the process.

This was not the case for class diagrams. Attempts to construct them using PlantUML and Mermaid, another open-source text-based diagramming tool, produced messy and convoluted results, making direct translation into Lucidchart impractical. Design adjustments were necessary, such as isolating the **Minimize** class into a stand-alone diagram and linking it to the central diagram. While this may have been achievable in either PlantUML or Mermaid, it was unclear how to do so immediately. For simplicity and due to time constraints, the class diagrams were constructed solely in Lucidchart. Insights gathered during the research and development phase highlighted PlantUML's limitations in managing the scope of larger projects[50].

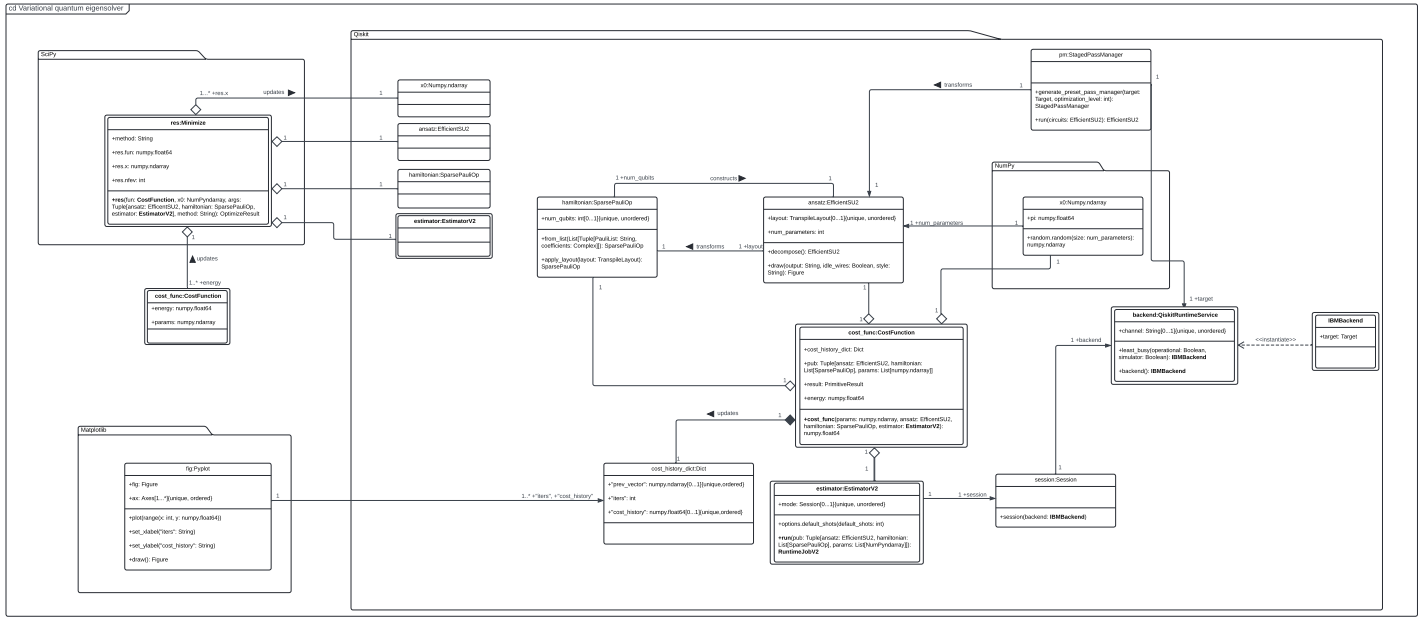

Figure 10: The application of Q-UML to a class diagram modelling the IBM Quantum Learning VQE Algorithm.

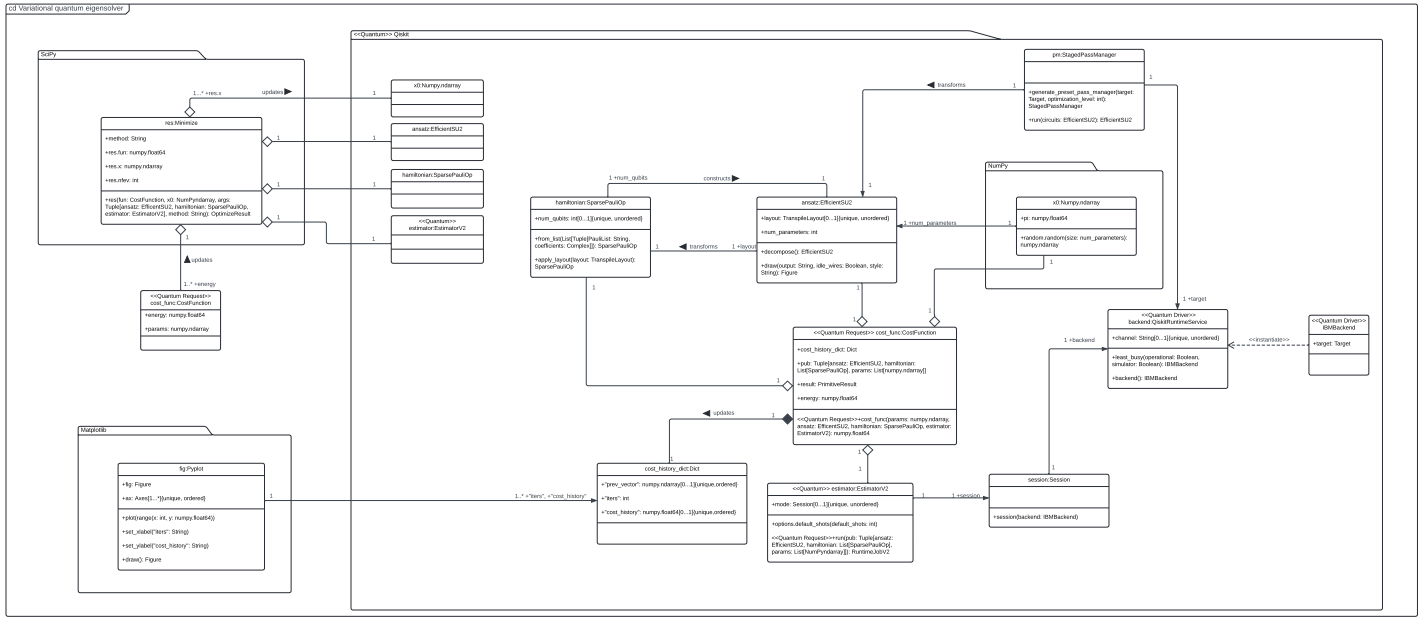

Figure 11: The application of the quantum UML profile to a class diagram modelling the IBM Quantum Learning VQE Algorithm.

4.2 Diagram Visibility

When observing the diagrams in full, Q-UML stands out for its clear indication of quantum elements. The lifelines, quantum messages, and double-lined frames highlighting quantum components in the sequence diagram are particularly distinct. In the class diagram, quantum classes are identifiable, though quantum associations and bold text for quantum data types are less prominent. In contrast, the quantum UML profile adaptations require closer inspection to discern their details.

The diagrams are notably large, representing a relatively small code snippet, which reduces the visibility of the content when the diagrams are observed in full. While the sequence diagram closely follows the output of PlantUML and might have adopted a different scale if drawn manually, the class diagram was manually constructed. Whether the size of these diagrams poses a concern is open to debate. They accurately depict the exact implementation of the VQE algorithm as outlined in IBM's tutorial, raising a fundamental question: should UML have been used to document a specific code implementation, or would it have been more appropriate to model the general steps required to implement VQE in Qiskit? This distinction could significantly affect the design and scale of the diagrams.

UML is a versatile modelling language suitable for both designing new systems and analysing existing ones. While using UML to model an exact implementation of code is not inherently incorrect, it limits the ability to compare the diagrams' effectiveness against other examples. The Q-UML and quantum UML profile papers are the only works that implement UML diagrams in a quantum context. Comparing the diagrams in this paper to those in the Q-UML and quantum UML profile papers is challenging for several reasons. Q-UML also uses Lucidchart, which allows for a fairer comparison, but it models a fundamentally different algorithm—Shor's algorithm—rather than a specific implementation of an algorithm in code. Meanwhile, the quantum UML profile examples focus on theoretical hybrid algorithms and include a class diagram illustrating the structure of a financial web application[37]. These examples differ in content from those in this paper but were also created using Papyrus, making direct comparisons challenging.

To fully evaluate the suitability of the created diagrams, they would need to be compared to similar ones—specifically, UML diagrams depicting the implementation of hybrid algorithms in a modern programming language. Given more time, revisions could include exploring alternative diagram versions, focusing on different implementations of VQE, modelling the algorithm rather than its specific realisation in Qiskit, or alternative hybrid algorithms implemented in code. Producing additional diagrams would enhance the design quality through iterative practice and provide a broader body of work for cross-comparison.

Despite these limitations, the scale of the diagrams may not be a significant issue, as their details remain readily accessible upon closer examination.

4.3 Diagram Information

A critical distinction in the quantum classifications between the two diagrams is that Q-UML identifies the class **Minimize** as quantum, whereas the quantum UML profile does not. Determining whether **Minimize** should be classified as a quantum element requires further exploration, as it is not immediately apparent in the context of this paper. Fundamentally, **Minimize** is a classical package. However, in this specific implementation of VQE, Q-UML classifies it as quantum because it takes the user-defined class **CostFunction** as a parameter. This class contains quantum information and facilitates quantum communication, upgrading it following the design principle **Quantum Supremacy**.

This issue may stem from the challenges of modelling a specific code implementation and raises an important question: what is the ultimate purpose of distinguishing quantum elements in UML? It is essential to differentiate classical and quantum elements due to their fundamentally different data types. However, when using an SDK like Qiskit, quantum computations are performed in the cloud, and any quantum information is ultimately translated back into classical information for the user. Determining which classes and data types should be considered quantum required a deep analysis of Qiskit's documentation. This involved examining both the documentation and the role of each class or data type in the operation. Despite this, the data types remain classical from the user's perspective. Only classes directly interacting with quantum hardware can be definitively classified as quantum.

This highlights a broader limitation of UML diagrams closely tied to specific code samples. An alternative UML diagram for VQE, agnostic to any programming language, might offer a clearer perspective on whether **Minimize** should be classified as quantum.

Another approach to address this question could involve applying UML on a larger scale. For instance, a UML diagram depicting the structure of the Qiskit Runtime Service could provide valuable insights into the significance of distinguishing classical and quantum elements. This would help contextualise their roles in the broader architecture.

Given the significantly higher expense of quantum hardware, UML could also be utilised for cost analysis. Distinguishing between classical and quantum elements becomes critical when assessing a system's dependency on each type of hardware. Understanding these dependencies allows for more accurate resource cost estimations. However, since the diagrams in this paper focus on a small code snippet requiring minimal quantum hardware access, a broader model that integrates classical and quantum hardware would be needed for more comprehensive insights. The class diagram presented in the quantum UML profile paper offers a strong example of effectively distinguishing quantum and classical elements within a complete system[37].

In the context of this paper's UML diagrams, the quantum UML profile provides more detailed insights into quantum-classical interactions than Q-UML. It explicitly differentiates between components of the quantum algorithm, elements facilitating quantum communication, and messages containing quantum

information, offering greater clarity and context for the quantum-classical hybrid algorithm under study. However, Q-UML excels in representing quantum data types with bold text notation for quantum input and output in class operations. In contrast, the quantum UML profile's use of stereotype tags for data types can appear visually cluttered, making it less distinguishable and informative than Q-UML.

A hybrid approach, combining the quantum UML profile with Q-UML's use of bold text for quantum data types, could present a promising alternative.

4.4 Diagramming Literature

The diagrams will be compared following guidance on good modelling language etiquette to enhance the above-given observations.

Once both diagrams are completed, it will be crucial to consult relevant literature on effective diagram design to make an informed assessment. Research papers exploring good diagram design, particularly concerning UML, should be considered. An example of such work is the paper "Improving Information System Design: Using UML and Axiomatic Design" [5], which provides valuable insights into improving system information design.

5 Conclusions

Is there a point for UML? Id use papyrus if writing UML for nice diagrams, otherwise would adopt my own way of doing a diagram that isn't UML and could specify quantum/classical elements just with colour coding.

The paper will conclude with an overall assessment of the advantages and disadvantages of each method. It will address whether the primary goal and all four main objectives were achieved. Additionally, there will be a discussion on whether UML is the most suitable approach for representing quantum systems or if a more general flowchart format might be more appropriate. This will include considering which method could be used in such a case. I assume that Q-UML would remain the most applicable, as the quantum UML profile is inherently based on UML formatting.

6 Future Work

Create quml extension Stuff about agnostic VQE and multiple diagram types Didnt achieve the fourth objective

Some aspects of the Q-UML notation may require revision. Firstly, a minor issue in the example diagrams from the Q-UML paper shows multiplicities depicted using "m" rather than "*". Secondly, the UML standard for representing class names in bold font[54] presents a potential readability issue for Q-UML, which uses bold font for all quantum elements. To address this, Q-UML may

need to adopt an alternative notation for quantum elements, such as combining bold text with underlining or another distinguishing feature.

The implementation of Q-UML notation for packages would also be worthwhile.

The quantum UML profile could benefit from clearer guidelines on when and how to apply its stereotypes. Currently, this information is primarily inferred from bullet-pointed explanations accompanying example diagrams. Adopting a methodology similar to Q-UML, which encapsulates such information within core design principles, could enhance usability for future researchers. Including notes or meta-attributes within the depicted meta-classes alongside the stereotypes could provide this clarity and aid in consistent implementation.

Depending on whether the fourth objective is achieved, further exploration into modelling fundamental quantum concepts would be beneficial for evaluating the effectiveness of applying UML to quantum systems. Plant UML is currently the most convenient tool for creating UML diagrams, as Lucidchart is significantly more labour-intensive. Developing an extension for Plant UML or other open-source tools like Mermaid to incorporate Q-UML formatting would be highly advantageous for future diagram creation. Such an extension would also increase the utility of this method, making it more accessible for continued exploration and broader adoption in the QSE field.

References

[1] Object Management Group (OMG). *UML Specification Version 2.5.1*. 2017.

[2] Q-SE 2020. *Q-SE 2020*. 2020. URL: https://q-se.github.io/qse2020/.

[3] Q-SE 2021. *Q-SE 2021*. 2021. URL: https://q-se.github.io/qse2021/.

[4] Sherief Abul-Ezz. *What is an SDK? SDK vs Library vs Framework*. 2021. URL: https://www.instabug.com/blog/what-is-an-sdk-sdk-vs-library-vs-framework.

[5] Luís Cavique et al. "Improving information system design: Using UML and axiomatic design". In: *Computers in Industry* 135 (2022), p. 103569. ISSN: 0166-3615. DOI: https://doi.org/10.1016/j.compind.2021.103569. URL: https://www.sciencedirect.com/science/article/pii/S0166361521001767.

[6] "Computing". In: *Computing* 104.10 (2022). DOI: 10.1007/s00607-022-01091-4.

[7] Encryption Consulting. *What is RSA? How does an RSA work?* 2024. URL: https://www.encryptionconsulting.com/education-center/what-is-rsa/.

[8] Creately. *Sequence Diagram Tutorial – Complete Guide with Examples*. 2022. URL: https://creately.com/guides/sequence-diagram-tutorial/.

[9] Ivan Djordjevic. “Chapter 2 - Quantum Mechanics Fundamentals”. In: *Quantum Information Processing and Quantum Error Correction*. Ed. by Ivan Djordjevic. Oxford: Academic Press, 2012, pp. 29–89. ISBN: 978-0-12-385491-9. DOI: https://doi.org/10.1016/B978-0-12-385491-9.00002-2. URL: https://www.sciencedirect.com/science/article/pii/B9780123854919000022.

[10] IBM Documenation. *Dependency relationships*. 2021. URL: https://www.ibm.com/docs/en/dmrt/9.5?topic=diagrams-dependency-relationships.

[11] IBM Quantum Documenation. *EfficientSU2*. 2024. URL: https://docs.quantum.ibm.com/api/qiskit/qiskit.circuit.library.EfficientSU2.

[12] IBM Quantum Documenation. *EstimatorV2*. 2024. URL: https://docs.quantum.ibm.com/api/qiskit-ibm-runtime/qiskit_ibm_runtime.EstimatorV2.

[13] IBM Quantum Documenation. *IBMBackend*. 2024. URL: https://docs.quantum.ibm.com/api/qiskit-ibm-runtime/qiskit_ibm_runtime.IBMBackend.

[14] IBM Quantum Documenation. *PauliList*. 2024. URL: https://docs.quantum.ibm.com/api/qiskit/qiskit.quantum_info.PauliList.

[15] IBM Quantum Documenation. *PrimitiveResult*. 2024. URL: https://docs.quantum.ibm.com/api/qiskit/qiskit.primitives.PrimitiveResult.

[16] IBM Quantum Documenation. *Primitives*. 2024. URL: https://docs.quantum.ibm.com/api/qiskit/primitives.

[17] IBM Quantum Documenation. *Qiskit Runtime*. 2024. URL: https://docs.quantum.ibm.com/api/qiskit-ibm-runtime/runtime_service.

[18] IBM Quantum Documenation. *Session*. 2024. URL: https://docs.quantum.ibm.com/api/qiskit-ibm-runtime/qiskit_ibm_runtime.Session.

[19] IBM Quantum Documenation. *SparsePauliOp*. 2024. URL: https://docs.quantum.ibm.com/api/qiskit/qiskit.quantum_info.SparsePauliOp.

[20] IBM Quantum Documenation. *StagedPassManager*. 2024. URL: https://docs.quantum.ibm.com/api/qiskit/qiskit.transpiler.StagedPassManager.

[21] Stack Exchange. *Generally, could VQE be used to prepare the highest energy state?* 2021. URL: https://quantumcomputing.stackexchange.com/questions/20920/generally-could-vqe-be-used-to-prepare-the-highest-energy-state.

[22] GeeksforGeeks. *Difference between OOP and POP*. 2023. URL: https://www.geeksforgeeks.org/difference-between-oop-and-pop/.

[23] Alexander S. Gillis. *object-oriented programming (OOP)*. 2024. URL: https://www.techtarget.com/searchapparchitecture/definition/object-oriented-programming-OOP.

[24] He-Liang Huang et al. "Near-term quantum computing techniques: Variational quantum algorithms, error mitigation, circuit compilation, benchmarking and classical simulation". In: *Science China Physics, Mechanics and Astronomy* 66.5 (Apr. 2023). ISSN: 1869-1927. DOI: 10.1007/s11433-022-2057-y. URL: http://dx.doi.org/10.1007/s11433-022-2057-y.

[25] IBM. *Qiskit 1.0 release summary*. 2024. URL: https://www.ibm.com/quantum/blog/qiskit-1-0-release-summary.

[26] Thivaharan Kalyanasundaram. *Object-Oriented Programming (OOP) and Procedural-Oriented Programming(POP): Two Paradigms, One Dilemma*. 2023. URL: https://medium.com/@kalyanasundaramthivaharan/object-oriented-programming-oop-and-procedural-oriented-programming-pop-two-paradigms-one-7a11cc143f0c.

[27] Dr Salini Karuvade. *Why is it hard to build quantum computers?* 2024. URL: https://futurumcareers.com/why-is-it-hard-to-build-quantum-computers.

[28] EITC/AI/TFQML TensorFlow Quantum Machine Learning. *How does the tensor product (Kronecker product) of Pauli matrices facilitate the construction of quantum circuits in VQE?* June 2024. URL: https://eitca.org/artificial-intelligence/eitc-ai-tfqml-tensorflow-quantum-machine-learning/variational-quantum-eigensolver-vqe/variational-quantum-eigensolver-vqe-in-tensorflow-quantum-for-2-qubit-hamiltonians/examination-review-variational-quantum-eigensolver-vqe-in-tensorflow-quantum-for-2-qubit-hamiltonians/how-does-the-tensor-product-kronecker-product-of-pauli-matrices-facilitate-the-construction-of-quantum-circuits-in-vqe/#:~:text=The%20tensor%20product%2C%20also%20known,of%20TensorFlow%20Quantum%20(TFQ)..

[29] IBM Quantum Learning. *Catalog*. 2024. URL: https://learning.quantum.ibm.com/catalog/tutorials.

[30] IBM Quantum Learning. *Variational Quantum Eigensolver*. 2024. URL: https://learning.quantum.ibm.com/tutorial/variational-quantum-eigensolver.

[31] NewScientist. *Record-breaking quantum computer has more than 1000 qubits*. 2023. URL: https://www.newscientist.com/article/2399246-record-breaking-quantum-computer-has-more-than-1000-qubits.

[32] Stack Overflow. *What is the meaning of single and double underscore before an object name?* 2009. URL: https://stackoverflow.com/questions/1301346/what-is-the-meaning-of-single-and-double-underscore-before-an-object-name.

[33] Visual Paradigm. *Overview of the 14 UML Diagram Types*. 2024. URL: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/overview-of-the-14-uml-diagram-types/.

[34] Visual Paradigm. *What is Package Diagram?* 2024. URL: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-package-diagram/.

[35] R. Pérez-Castillo. *Quantum UML Profile Repository*. 2021. URL: https://github.com/ricpdc/quml.

[36] Ricardo Pérez-Castillo, Luis Jiménez-Navajas, and Mario Piattini. "Modelling Quantum Circuits with UML". In: *2021 IEEE/ACM 2nd International Workshop on Quantum Software Engineering (Q-SE)*. 2021, pp. 7–12. DOI: 10.1109/Q-SE52541.2021.00009.

[37] Ricardo Pérez-Castillo and Mario Piattini. "Design of classical-quantum systems with UML". In: *Computing* 104.11 (2022), pp. 2375–2403. ISSN: 1436-5057. DOI: 10.1007/s00607-022-01091-4. URL: https://doi.org/10.1007/s00607-022-01091-4.

[38] Carlos A. Pérez-Delgado. "A Quantum Software Modeling Language". In: *Quantum Software Engineering*. Ed. by Manuel A. Serrano, Ricardo Pérez-Castillo, and Mario Piattini. Cham: Springer International Publishing, 2022, pp. 103–119. ISBN: 978-3-031-05324-5. DOI: 10.1007/978-3-031-05324-5_6. URL: https://doi.org/10.1007/978-3-031-05324-5_6.

[39] Carlos A. Pérez-Delgado and Hector G. Pérez-Gonzalez. "Towards a Quantum Software Modeling Language". In: *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*. IEEE, 2020, pp. 442–444. DOI: 10.1109/ICSEW48907.2020.00073. URL: https://doi.org/10.1109/ICSEW48907.2020.00073.

[40] Carlos A. Pérez-Delgado and Hector G. Pérez-Gonzalez. *Towards a Quantum Software Modeling Language*. 2020. URL: https://www.youtube.com/watch?v=Di6rlch2_BU.

[41] Alberto Peruzzo et al. "A variational eigenvalue solver on a photonic quantum processor". In: *Nature Communications* 5.1 (2014), p. 4213. ISSN: 2041-1723. DOI: 10.1038/ncomms5213. URL: https://doi.org/10.1038/ncomms5213.

[42] PlantUML. *PlantUML at a Glance*. 2024. URL: https://plantuml.com/.

[43] Qiskit. *Variational Quantum Eigensolver — Qiskit Global Summer School 2023*. 2023. URL: https://www.youtube.com/watch?v=AhEnvYgoA34.

[44] Qiskit. *What is a Hamiltonian? Quantum Jargon Explained*. 2023. URL: https://www.youtube.com/watch?v=BusR0WQ_Gxo.

[45] IBM Quantum. *ISA Circuits*. 2024. URL: https://www.ibm.com/quantum/blog/isa-circuits.

[46] QuEra. *Ansatz*. 2024. URL: https://www.quera.com/glossary/ansatz.

[47] QuEra. *Ground State*. 2024. URL: https://www.quera.com/glossary/ground-state.

[48] QuEra. *NISQ*. 2023. URL: https://www.quera.com/glossary/nisq.

[49] QuEra. *Quantum Advantage*. 2024. URL: https://www.quera.com/glossary/advantage.

[50] Reddit. *PlantUML turns text into diagrams*. 2019. URL: https://www.reddit.com/r/programming/comments/ebes8j/plantuml_turns_text_into_diagrams/.

[51] Henry Reich. *How Quantum Computers Break Encryption — Shor's Algorithm Explained*. 2019. URL: https://www.youtube.com/watch?v=lvTqbM5Dq4Q.

[52] Margaret Rouse. *What Does Modeling Language Mean?* 2016. URL: https://www.techopedia.com/definition/20810/modeling-language.

[53] SciPy. *minimize*. 2024. URL: https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html.

[54] Martina Seidl et al. *UML @ Classroom: An Introduction to Object-Oriented Modeling*. Cham: Springer, 2014.

[55] M.A. Serrano, R. Pérez-Castillo, and M. Piattini. *Quantum Software Engineering*. Springer International Publishing, 2022. ISBN: 978-3-031-05323-8. DOI: 10.1007/978-3-031-05324-5.

[56] Peter W. Shor. "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer". In: *SIAM Journal on Computing* 26.5 (Oct. 1997), pp. 1484–1509. DOI: 10.1137/s0097539795293172. URL: http://dx.doi.org/10.1137/S0097539795293172.

[57] Eliza Taylor. *Advantages and Disadvantages of Quantum Computing: A Complete Guide*. 2024. URL: https://www.theknowledgeacademy.com/blog/advantages-and-disadvantages-of-quantum-computing/.

[58] uml-diagrams.org. *Session*. 2024. URL: https://www.uml-diagrams.org/frame.html.

[59] uml-diagrams.org. *UML Composition*. 2021. URL: https://www.uml-diagrams.org/composition.html.

[60] Wikipedia. *Ansatz*. 2024. URL: https://en.wikipedia.org/wiki/Ansatz.

[61] Wikipedia. *PlantUML*. 2024. URL: https://en.wikipedia.org/wiki/PlantUML.

[62] Wikipedia. *Simula*. 2024. URL: https://en.wikipedia.org/wiki/Simula.

[63] Wikipedia. *Variational quantum eigensolver*. 2024. URL: https://en.wikipedia.org/wiki/Variational_quantum_eigensolver.

7 Appendix

7.1 IBM Quantum Learning VQE Code

Below is the example code provided by IBM Quantum Learning to implement the VQE algorithm in Qiskit[29].

```
# General imports
import numpy as np

# Pre-defined ansatz circuit and operator class for Hamiltonian
from qiskit.circuit.library import EfficientSU2
from qiskit.quantum_info import SparsePauliOp

# SciPy minimizer routine
from scipy.optimize import minimize

# Plotting functions
import matplotlib.pyplot as plt

# Runtime imports
from qiskit_ibm_runtime import QiskitRuntimeService, Session
from qiskit_ibm_runtime import EstimatorV2 as Estimator

# To run on hardware, select the backend with the fewest number of jobs
    in the queue
service = QiskitRuntimeService(channel="ibm_quantum")
backend = service.least_busy(operational=True, simulator=False)

# Define Hamiltonian
hamiltonian = SparsePauliOp.from_list(
    [("YZ", 0.3980), ("ZI", -0.3980), ("ZZ", -0.0113), ("XX", 0.1810)]
)

# Create ansatz
ansatz = EfficientSU2(hamiltonian.num_qubits)
num_params = ansatz.num_parameters

# Initalise pass manager
from qiskit.transpiler.preset_passmanagers import
    generate_preset_pass_manager

target = backend.target
pm = generate_preset_pass_manager(target=target, optimization_level=3)

# Transform ansatz
ansatz_isa = pm.run(ansatz)

# Transform Hamiltonian
```

```
hamiltonian_isa = hamiltonian.apply_layout(layout=ansatz_isa.layout)

# Define Cost Function
def cost_func(params, ansatz, hamiltonian, estimator):
    """Return estimate of energy from estimator

    Parameters:
        params (ndarray): Array of ansatz parameters
        ansatz (QuantumCircuit): Parameterized ansatz circuit
        hamiltonian (SparsePauliOp): Operator representation of
            Hamiltonian
        estimator (EstimatorV2): Estimator primitive instance
        cost_history_dict: Dictionary for storing intermediate results

    Returns:
        float: Energy estimate
    """
    pub = (ansatz, [hamiltonian], [params])
    result = estimator.run(pubs=[pub]).result()
    energy = result[0].data.evs[0]

    cost_history_dict["iters"] += 1
    cost_history_dict["prev_vector"] = params
    cost_history_dict["cost_history"].append(energy)
    print(f"Iters. done: {cost_history_dict['iters']} [Current cost:
        {energy}]")

    return energy

# Initalise cost history dictionary
cost_history_dict = {
    "prev_vector": None,
    "iters": 0,
    "cost_history": [],
}

# Initalise parameters
x0 = 2 * np.pi * np.random.random(num_params)

# Run optimisation routine with COBYLA
with Session(backend=backend) as session:
    estimator = Estimator(mode=session)
    estimator.options.default_shots = 10000

    res = minimize(
        cost_func,
        x0,
        args=(ansatz_isa, hamiltonian_isa, estimator),
        method="cobyla",
    )
```

```
# Verify parameters
all(cost_history_dict["prev_vector"] == res.x)

# Verify iterations
cost_history_dict["iters"] == res.nfev

# Plot convergence
fig, ax = plt.subplots()
ax.plot(range(cost_history_dict["iters"]),
    cost_history_dict["cost_history"])
ax.set_xlabel("Iterations")
ax.set_ylabel("Cost")
plt.draw()
```