Patterns for Quantum Software Development

Fabian Bühler, Johanna Barzen, Martin Beisel, Daniel Georg, Frank Leymann, and Karoline Wild
Institute of Architecture of Application Systems, University of Stuttgart
Universitätsstrasse 38, 70569 Stuttgart, Germany
email: {firstname.lastname}@iaas.uni-stuttgart.de

***Abstract*—Quantum algorithms have the potential to outperform classical algorithms for certain problems. However, implementing quantum algorithms in a reusable manner and integrating them into applications poses new challenges. To ensure reusability and integrability, quantum algorithm implementations must handle different problem sizes, be able to be processed by different quantum computers, and should also be able to be used and integrated by non-quantum experts. In classical software engineering a variety of best practices and design principles to achieve reusability of classical software components are well-known and documented as patterns. However, quantum software engineering currently lacks best practices for creating reusable implementations of quantum algorithms. To close this gap, this paper presents five patterns that describe proven solutions for modularization, integration, and translation of quantum algorithm implementations, further extending the existing quantum computing pattern language.**

Index Terms*—Quantum Computing; Pattern Language; Quantum Software Engineering; Quantum Computing Patterns.

I. Introduction

Quantum algorithms have the potential to outperform their classical counterparts by exploiting quantum mechanical phenomena such as entanglement. Most quantum algorithms are hybrid, comprising classical and quantum computations. This includes not only variational quantum algorithms (VQAs) [1], e.g., the Variational Quantum Eigensolver (VQE) and Quantum Approximate Optimization Algorithm (QAOA) [2], but also Shor's algorithm for prime factorization [3][4] and Grover search [5], which require classical pre- and post-processing steps and are therefore also hybrid [6].

Implementing a quantum algorithm is a complex task requiring expertise in the field of quantum computing and software engineering. Implementations consist of code that represents quantum circuits, and code defining the classical logic of the quantum algorithm. These algorithm implementations can then be integrated in hybrid quantum-classical applications [7] to solve specific problems. Thus, the reusability of implementations of quantum algorithms and their integration into applications, where their hybrid nature on the level of algorithms and applications poses additional challenges [8], are of great importance to quantum software engineering [9].

Reusability in the context of quantum algorithm implementations comprises multiple aspects: One aspect is the reusability of an implementation in different applications. The programming language used for the implementation has a large influence on its reusability, e.g., quantum algorithms implemented in a Python-based quantum programming language can easily be integrated directly into applications also implemented in Python. Another aspect is the reusability for different problem instances. Reusable implementations of quantum algorithms, e.g., to solve the maximum cut problem [10], should be able to process graphs of different sizes, which may affect the number of qubits required. Moreover, reusable algorithm implementations should be executable on different hardware, i.e., quantum computers of different vendors.

To achieve a high degree of reusability, classical software engineering provides many well-documented and well-known best practices for structuring the code of classical applications, such as modularization to achieve separation of concerns. To the best of our knowledge, similar best practices for implementing quantum algorithms and integrating them into applications are neither well-established nor well-documented in the emerging field of quantum software engineering.

An established method to document best practices for solving recurring problems are patterns [11]. They provide a structured way to capture design and architectural knowledge in a human-readable format. Patterns have been originally introduced by Alexander [12] in the domain of building architectures. Today they are widely used in different domains including software engineering, e.g., for the design of object-oriented applications [13], the integration of enterprise applications [14], or cloud computing [15]. Typically, patterns of a certain domain are organized in a pattern language. Patterns within a pattern language are interconnected, to facilitate the combination of related patterns and ease the understanding of similar problems and their solutions.

In the quantum computing domain, Leymann [16] introduced a pattern language, which has since been extended several times [2][17][18][19][20][21]. The pattern language contains patterns of different categories, e.g., patterns related to quantum operations or specific quantum algorithm classes. However, patterns documenting best practices to improve the reusability of quantum algorithm implementations are not yet part of the language. To close this gap, we extend the quantum computing pattern language by five new patterns that cover different aspects of reusability of such implementations.

The structure of this paper is as follows: Section II provides fundamentals on quantum software engineering and introduces the used pattern format and authoring process. Next, Section III explains the new patterns in detail, followed by a short discussion in Section IV. Then, related work is presented in Section V and Section VI provides a conclusion.

 ISBN: 978-1-68558-049-0

II. Background

This section introduces fundamentals of quantum software engineering. Additionally, the concept of patterns, the pattern format used in this paper, and the pattern authoring process are described in more detail.

A. Quantum Software Engineering

Quantum software engineering is an emerging field that aims to apply software engineering principles to the development of applications using *quantum algorithm* implementations [22]. In particular, the reusability of such implementations is a main goal of quantum software engineering [9]. One aspect of reusability is the ability to integrate quantum algorithm implementations into different applications. Typically, quantum algorithm implementations are integrated into *hybrid quantum-classical applications* [7], or *hybrid applications* for short, to solve problems that can only be solved efficiently using a quantum computer, while classical computers are used for general purpose computation and data storage.

Most quantum algorithms are hybrid algorithms, including variational algorithms as well as all algorithms that require classical computing for pre- and post-processing [6]. Thus, a quantum algorithm implementation is divided into a *quantum part* and a *classical part*. Although both classical and quantum parts can be implemented in similar text-based programming languages, developing quantum parts requires expert knowledge of quantum computing and its mathematical foundations in addition to software engineering knowledge.

Quantum computers use unique instruction sets. To make use of synergies with the existing patterns this paper focuses on gate-based quantum computing. For the gate-based quantum computing model, *quantum gates* represent the operations performed on qubits and together with the measurements to retrieve the results of the quantum computation, they form the *quantum circuit* that can be processed by a quantum computer. However, quantum computers of different vendors support different gates natively, which can lead to vendor lock-in when implementing quantum algorithms.

The inputs of a quantum algorithm, e.g., the problem instance, must be encoded into the quantum circuit. Therefore, reusable quantum algorithm implementations require a dynamic generation of quantum circuits based on the given input.

B. Pattern Format and Authoring Method

Patterns provide proven solutions for recurring problems in a structured, human-readable manner. Alexander et al. [12] originally introduced the concept of patterns in the domain of building architectures. This concept has then been adopted by the information technology domain. For documenting patterns in this domain, Coplien [11] provides guidelines for writing software patterns. In this work, the already established pattern format of the quantum computing pattern language that follows these guidelines for documenting patterns is used.

Each pattern has a descriptive *name* and a mnemonic *icon*. A short question briefly introduces the *problem* solved by the pattern. Next, the *context* in which the problem occurs and the *forces* acting in that context are described in detail. Forces are aspects of the problem context that require special focus. Understanding these forces is crucial as solutions often cannot resolve the forces but rather balance the forces with varying tradeoffs [11]. Then, the *solution* is presented along with a solution sketch, followed by a paragraph describing the *result* of applying that solution. The solution is presented in an abstract, technology-independent manner so that it can be applied in a broad context. Finally, real world occurrences of the pattern are listed in *known uses* and relationships to patterns solving similar problems or that are recommended to be used in combination are provided in *related patterns*.

To identify patterns for quantum software engineering solving recurring problems, we empirically inspected quantum algorithm implementations of well-established libraries like Qiskit [23] and Amazon Braket [24]. Additionally, we analyzed algorithm descriptions from scientific literature and quantum computing tutorials. Moreover, we evaluated the potential applications of established best practices of classical software engineering in the context of quantum computing.

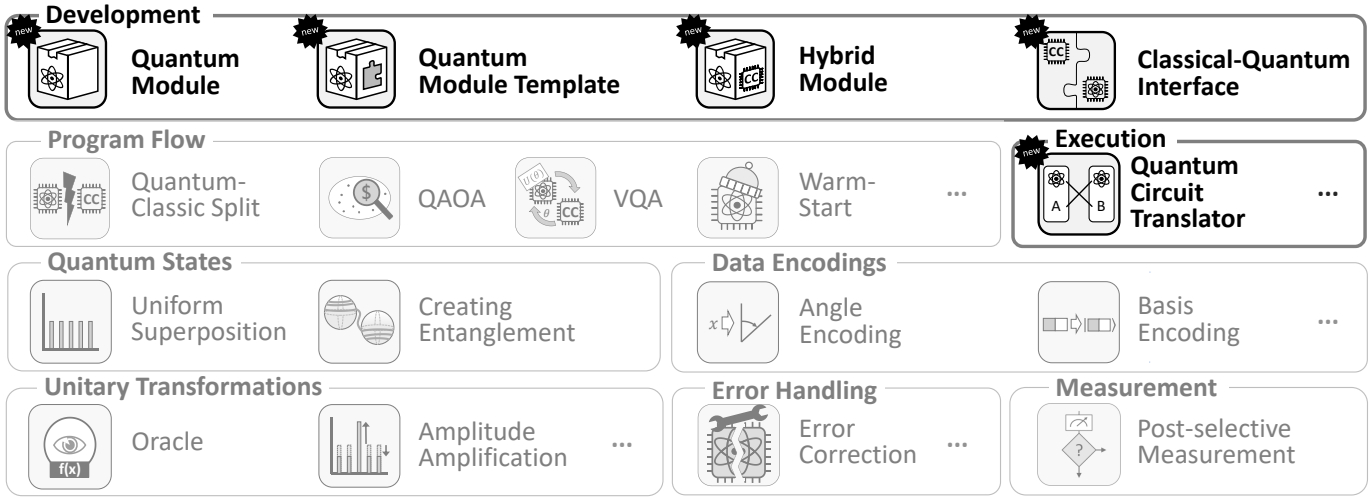


Figure 1. Overview of the quantum computing pattern language [16] including the new patterns.

 ISBN: 978-1-68558-049-0

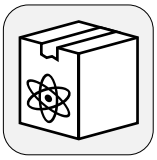
III. Patterns for Quantum Software Development

As a first step to capture best practices for quantum software development and execution, five new patterns are introduced in this section that extends the existing quantum computing pattern language [2][16][17][18][19][20][21]. Figure 1 shows an overview of the categories of the quantum computing pattern language, including the new *Development* and the extended *Execution* category. The existing patterns mostly focus on the *program flow*, which uses *unitary transformations* to manipulate *quantum states*, e.g., to create specific *data encodings* for an algorithm, and *measurements* to read out the results of a quantum computation. The new patterns focus on modularization, reusability, and the integration of quantum algorithm implementations into hybrid applications.

The two patterns Quantum Module and Quantum Module Template focus on encapsulating the implementation of the quantum part of a quantum algorithm as reusable modules. Quantum Modules can generate quantum circuits with a known structure and behavior, while Quantum Module Templates allow the integration of arbitrary behavior into the generated quantum circuit. In contrast, a Hybrid Module is used to package a complete quantum algorithm implementation containing the quantum as well as the classical parts, such as the continued fraction expansion of Shor's algorithm [3], of the quantum algorithm. Its main use case is distribution and deployment of a quantum algorithm implementation for integration into hybrid applications. The implementation of a Hybrid Module can again be modularized, e.g., using Quantum Modules or Quantum Module Templates. A Classical-Quantum Interface facilitates the use of quantum algorithms by non-quantum computing experts. Last, a Quantum Circuit Translator enables the execution of quantum circuits on quantum computers of different vendors by translating the circuits into a compatible format. In the following subsections these patterns are presented in detail.

A. Quantum Module

Problem: *How can the implementation of the quantum part of a quantum algorithm be packaged for reuse independent of concrete input values?*



Context: Each quantum algorithm is a hybrid algorithm, i.e., parts of the algorithm require quantum computers and other parts require classical computers for their execution. For the execution of the quantum part, a quantum circuit implementing the required operations is needed. However, quantum circuits are problem-specific and, thus, depend on various inputs, e.g., the problem instance or initial values for parameterized quantum gates, which are then optimized by a classical optimizer. Therefore, the implementation of the quantum part of a quantum algorithm must be input-agnostic in order to be reusable.

Forces: Quantum circuits to be processed by a quantum computer must already contain all appropriately encoded input values. A static implementation of a quantum circuit that does not allow the quantum circuit to be changed based on some input values cannot be reused to solve different problems. Thus, a reusable implementation of the quantum part of a quantum algorithm needs to be able to adapt the quantum circuit to different input values. The input values that need to be encoded in a quantum circuit are, first, the problem to be solved, e.g., an implementation of Shor's algorithm [3] would require as input the number to be factored into primes, and second, parameters used for optimization or machine learning, e.g., for QAOA [2].

Moreover, implementing the quantum part of a quantum algorithm requires in depth knowledge of quantum computing and the underlying mathematical concepts. Thus, quantum computing experts are required in the development teams. However, other parts of the algorithm that only require classical computation, e.g., classical optimizers, may not require quantum computing knowledge at all and can be implemented by different teams without a quantum computing expert.

Solution: Separate the implementation of the quantum part of the quantum algorithm into one or more Quantum Modules. These modules contain the code that generates quantum circuits based on input values provided to the module. Quantum modules can also be used to reduce the number of code duplicates by implementing common parts of a quantum circuit as a reusable quantum module.

The solution sketch in Figure 2 depicts that a Quantum Module receives input values and uses generative code to construct quantum circuits depending on these input values. This ensures the reusability of the Quantum Module, as the implementation can create quantum circuits for different problem sizes as well as parameters.

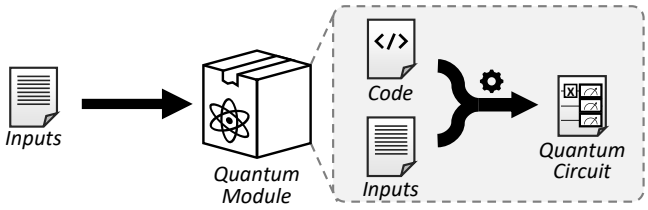


Figure 2. Solution sketch: Quantum Module.

Result: A quantum algorithm implementation is partitioned into (i) Quantum Modules containing the implementations of the quantum part, and (ii) additional classical code required for the control flow and other classical computations of the quantum algorithm. The quantum modules are independent of the concrete input values, which increases their reusability for different quantum algorithm implementations.

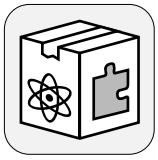
The separation of code that generates quantum circuits into quantum modules can thus also be reflected in the organizational structure of the development teams. Only the teams working on the quantum modules need quantum computing experts, while other teams mainly need experts in classical software engineering.

 ISBN: 978-1-68558-049-0

Known Uses: QUANTUM MODULES can already be found in several libraries for building quantum circuits. In Amazon Braket [24], the Grover algorithm [5] is offered as a module with functions to build the oracle and execute the Grover search. Another example of a QUANTUM MODULE for creating oracles is the *PhaseOracle* in Qiskit [23]. Generic parts used in multiple quantum algorithms, such as the quantum fourier transformation used in Shor's algorithm [3] are available in Amazon Braket [24] and Qiskit [23]. The quantum phase estimation, which is also part of Shor's algorithm can be constructed in Qiskit [23] with the *PhaseEstimation* module.

Related Patterns: A QUANTUM MODULE generating specific quantum circuits for a quantum algorithm can be used inside a HYBRID MODULE that contains the implementation of the overall quantum algorithm with its quantum and classical parts. Quantum circuits generated by a QUANTUM MODULE can be integrated into a QUANTUM MODULE TEMPLATE to create a complete quantum circuit, if the QUANTUM MODULE only generates a part of a quantum circuit. The boundary of a QUANTUM MODULE is directly corresponding to the QUANTUM-CLASSIC SPLIT [16]. The QUANTUM-CLASSIC SPLIT pattern states, that there is necessarily a separation – a split – between code executed on classical computers and code executed on quantum computers. Thus, the QUANTUM MODULE pattern is related to this pattern.

B. Quantum Module Template



Problem: *How can the implementation of the quantum part of a quantum algorithm be packaged for reuse when some of the behavior is determined later?*

Context: Some quantum algorithms can be implemented in a reusable manner, but their behavior may be partially modified depending on the problem to which the algorithm is applied. For example, the Grover search algorithm [5] contains an unspecified oracle. The information required for defining the concrete behavior of this oracle may not be available until a later point in time. Similar cases are algorithms like QAOA [2], which do not specify a concrete ansatz to use. Thus, implementations of the quantum part of such algorithms, where the unspecified behavior can be integrated later, are required.

Forces: Quantum algorithms may intentionally leave parts of the behavior of the quantum part unspecified until a later point in time. For example, the Grover search [5] uses an unspecified placeholder gate, as the specific function that marks the correct values cannot be known before it has been decided what to search for. In the case of QAOA, the choice of a suitable ansatz depends on information that is only available at runtime. However, for the algorithms to be executed, the missing behavior must be integrated before the execution of the quantum circuits on a quantum computer. Note, that similar situations can arise if the development of a quantum algorithm is split between different teams.

Integrating quantum behavior into an existing circuit requires a specification of the requirements an implementation has to fulfill to be integrated and function correctly. This includes the specification of the input qubits available, possible ancilla qubits, on which qubits and in what form the output is expected, and any other requirements or restrictions, e.g., on the creation of entanglement between quantum bits. Some of the restrictions, e.g., the number of available ancilla qubits, may additionally depend on the quantum computer used for execution, as a quantum computer with more qubits can allocate more ancilla qubits if the number of qubits used in the circuit is otherwise constant.

Solution: Implement the generic behavior of the quantum part of a quantum algorithm in a QUANTUM MODULE TEMPLATE. This module accepts inputs, that define the unspecified behavior to be integrated into the final quantum circuit. The behavior can either be specified as a quantum circuit or as a QUANTUM MODULE that generates the required quantum circuit. This circuit then gets integrated by the QUANTUM MODULE TEMPLATE into the main quantum circuit that represents the generic behavior.

To ensure that the *behavior* input, in form of a quantum circuit, can be integrated to correctly perform the operations it contains, the QUANTUM MODULE TEMPLATE must include specifications in the documentation that can be used to build a compatible quantum circuit, as outlined in the pattern forces. This specification is mainly a contract that needs to be fulfilled by the quantum circuit serving as input for the template. Similar contracts, e.g., plugin contracts [25], are also used in classical software engineering.

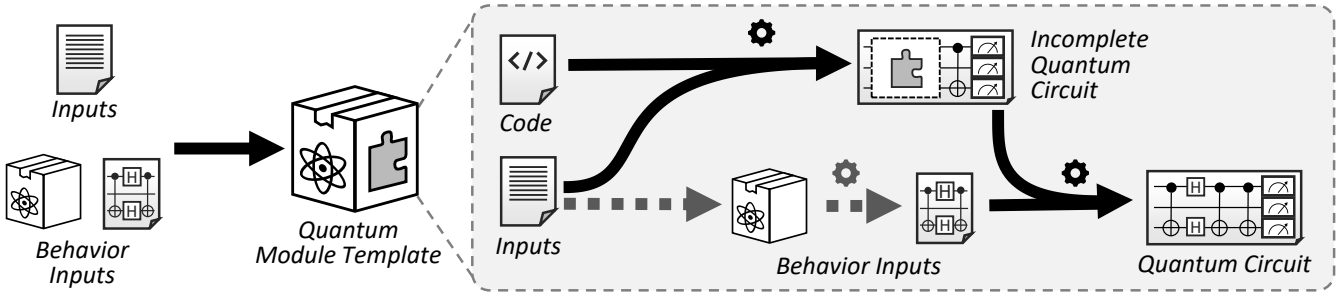


Figure 3. Solution sketch: QUANTUM MODULE TEMPLATE.

Figure 3 sketches the essential building blocks of a QUANTUM MODULE TEMPLATE. The template requires two kinds of inputs: (i) the input values representing the problem to be solved as well as parameters affecting the circuit generation, as used in the QUANTUM MODULE, and (ii) behavior inputs partially specifying the behavior of the algorithm, provided in the form of a quantum circuit or a QUANTUM MODULE. Much like the QUANTUM MODULE, the QUANTUM MODULE TEMPLATE uses the input values to generate a quantum circuit, which is still incomplete as it does not include the behavior from the behavior inputs yet. If the behavior inputs are provided in the form of a quantum module, this module is used to generate a quantum circuit from the inputs. Finally, the quantum circuit is integrated into the incomplete main circuit. However, implementations of the template are not limited to the exemplary steps shown here.

Result: The generic behavior of the quantum part of a quantum algorithm is implemented as QUANTUM MODULE TEMPLATE that requires behavior inputs to generate an executable complete quantum circuit. The behavior inputs specify the parts of the algorithm's behavior that cannot be known in advance. Thereby, their influence on the resulting quantum circuit can be significantly higher than with a QUANTUM MODULE. The behavior inputs must be compatible with the required input definitions of the template.

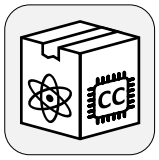
The integration of the behavior inputs can be done at design time if the behavior is provided as QUANTUM MODULE, since the QUANTUM MODULE generates the circuit based on the input values. Templates can be nested inside other templates to compose quantum circuits from QUANTUM MODULES implementing higher level circuit functions. This facilitates the replacement of a part of a quantum circuit if that part should be generated by a new QUANTUM MODULE implementing an improved algorithm, e.g., a more efficient state preparation.

Known Uses: Various quantum algorithms, e.g., the algorithm of Deutsch [26] or the Grover search [5], use an unspecified unitary gate as placeholder. Implementations of the generic behavior of these algorithms are available in Amazon Braket [24] and Qiskit [23]. These algorithms need oracle circuits to replace the placeholder gate, which are described in the ORACLE pattern [16]. The oracle replacement is described in [27] in an *Oracle Expansion Task* for workflows using the *Quantum Modeling Extension*. Generic parts of QAOA [2], such as state preparation and the mixer operator are implemented in Amazon Braket [24] and Qiskit [23] and can be used by providing a quantum circuit encoding the cost function.

Related Patterns: A QUANTUM MODULE TEMPLATE is a special kind of QUANTUM MODULE that additionally accepts behavior inputs, which are integrated into the generated quantum circuit. QUANTUM MODULE TEMPLATES can be used to integrate, e.g., ORACLES [16] and STATE PREPARATION [16] circuits into an executable quantum circuit allowing circuits to be built from smaller modules.

C. Hybrid Module

Problem: *How can the implementation of a quantum algorithm requiring both classical and quantum computations be packaged so that it can be integrated into applications?*



Context: Quantum algorithms often require classical computation for pre- and post-processing of the quantum computation results [6]. This means that almost all quantum algorithms are hybrid. Thus, any implementation of a quantum algorithm has to contain both the quantum and the classical parts for the algorithm to be functional.

Forces: Quantum algorithms typically require a classical computer for some parts of their computation. This means that they can have multiple quantum and classical parts. For example, VQAs, such as VQE and QAOA, alternate between quantum and classical computations [1][2]. Both, the quantum and the classical part, are required for the algorithm to work correctly. This also includes the control flow of the algorithm, which is included in the classical part of the algorithm.

Integrating a quantum algorithm into an application requires the implementation of the entire algorithm. A dedicated interface is required to enable the integration into applications. Deploying the algorithm to a hybrid runtime, which can execute both the quantum and the classical part of the algorithm, even requires both parts to be deployed together.

Solution: Package the entire quantum algorithm, i.e., both the quantum parts and the classical parts, as a HYBRID MODULE. This module can be composed of smaller modules, e.g., QUANTUM MODULES. It also contains the control flow logic to orchestrate the quantum and classical computation. The HYBRID MODULE should provide an interface that facilitates its integration into applications. This interface should mainly accept the required problem-specific input values, i.e., the problem that should be processed by the algorithm. Moreover, the interface of a HYBRID MODULE can also allow behavior inputs to the classical as well as quantum computation, similar to the QUANTUM MODULE TEMPLATES.

An exemplary sketch of a HYBRID MODULE is shown in Figure 4. It includes the control flow logic and implementations of classical and quantum parts with a loop between quantum and classical computation. The implementation of such a hybrid module can consist of multiple smaller modules, e.g., the three classical and one quantum computation steps shown can each be implemented in a separate module.

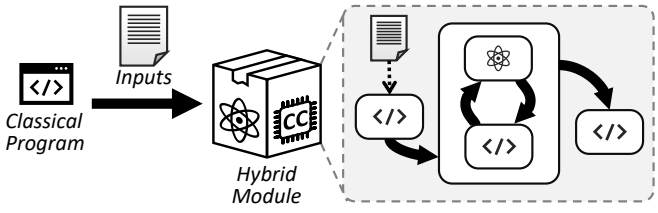


Figure 4. Solution sketch: HYBRID MODULE.

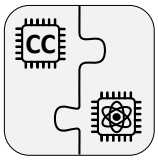
 ISBN: 978-1-68558-049-0

Result: The entire quantum algorithm implementation is packaged as a HYBRID MODULE. It contains both the quantum and the classical parts, as well as the control flow logic. HYBRID MODULES can be used to deploy the algorithm as a standalone service, e.g., in a hybrid runtime environment that can execute both the classical and the quantum part [28]. Furthermore, a HYBRID MODULE can be distributed as a library that implements the quantum algorithm and can be integrated into classical applications. It provides an interface for the application to use. To facilitate the integration of a HYBRID MODULE by problem-domain experts, a CLASSICAL-QUANTUM INTERFACE can be used as the modules' interface.
Known Uses: One concrete example are implementations of Shor's algorithm [3] which computes the prime factors of the input number. The period-finding calculated on the quantum computer and the classical post-processing performing the continued fraction expansion is packaged as a HYBRID MODULE in Amazon Braket [24], Qiskit [23] and Q# [29].

Other examples of HYBRID MODULES are implementations of VQAs [1], e.g., QAOA and VQE implementations for the Qiskit Runtime contain the full quantum algorithm implementation [23]. Beisel et al. [30] showcase a service ecosystem enabling a workflow-based composition of HYBRID MODULES for VQAs.
Related Patterns: The quantum part of the algorithm implementation inside a HYBRID MODULE can be organized into QUANTUM MODULES and QUANTUM MODULE TEMPLATES. To facilitate their integration into applications by problem-domain experts without quantum computing knowledge, the HYBRID MODULE can expose a problem domain-specific CLASSICAL-QUANTUM INTERFACE.

D. Classical-Quantum Interface

Problem: *How can a quantum algorithm implementation be used by developers without quantum computing knowledge?*



Context: Using a quantum algorithm implementation often requires in depth quantum computing knowledge. For example, the Grover search algorithm requires that the user provides a quantum circuit for the missing oracle [5]. Other algorithms, like QAOA, require choosing an ansatz, which also requires quantum computing knowledge [1][2]. However, software developers who want to integrate a quantum algorithm implementation into an application have a deep understanding of the problem domain rather than deep knowledge of quantum computing.
Forces: To integrate a quantum algorithm implementation into an application, a compatible interface is required. A HYBRID MODULE already provides an interface enabling its integration into applications, however, using this interface may still require considerable quantum computing knowledge. For example, it may require the problem instance to be provided in the form of a behavior input to the quantum part of an algorithm, or it may have parameters that otherwise influence the quantum part, e.g., by enabling certain error mitigation methods. The effects of the changes, e.g., on resource requirements or runtime, are difficult to estimate without knowledge of quantum computers. Thus, to facilitate the integration of quantum algorithms by problem-domain experts without quantum computing knowledge, such an interface is not sufficient.
Solution: Use a CLASSICAL-QUANTUM INTERFACE that hides the quantum implementation details. Inputs can be provided to the interface in formats specific to the problem domain. These problem domain-specific inputs are internally converted into inputs in the formats required by the implementation of the quantum part.

The documentation of interface inputs that affect the quantum part requires special consideration, since understanding their impact on algorithm execution is important information when integrating the quantum algorithm implementation. Thus, the impact of these inputs on the algorithm should be documented in a comprehensible and easily understandable manner by the interface developer. For example, a parameter that increases the accuracy of the result, but also increases the number of gates in the generated circuits, which can result in increased errors with current quantum computers, could be documented as follows: *"Increasing this parameter can increase the accuracy of the result. However, it also increases the probability of computation errors accumulating, which can negate any improvement in accuracy."*

Figure 5 shows the interaction of a classical program with a quantum algorithm implemented as a HYBRID MODULE through a CLASSICAL-QUANTUM INTERFACE. It transforms the problem domain-specific input of the classical program into the inputs required by the quantum algorithm. This interface can also be integrated directly into the HYBRID MODULE.

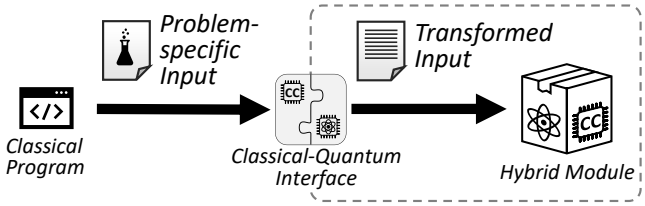


Figure 5. Solution sketch: CLASSICAL-QUANTUM INTERFACE.

Result: The quantum algorithm implementation can be utilized using a CLASSICAL-QUANTUM INTERFACE. Problem domain experts can make use of this quantum algorithm implementation through the CLASSICAL-QUANTUM INTERFACE created for their domain. The knowledge required to utilize the algorithm implementation is presented in the interface documentation, and the format of input parameters is familiar to problem-domain experts.
Known Uses: Domain-specific libraries for quantum computing are among the first having implemented this pattern. Examples for already implemented CLASSICAL-QUANTUM INTERFACES can be found in the chemistry domain in Qiskit, Amazon Braket, and Q# [31][32][33]. They offer transformation modules that map the electronic structure of molecules to qubits. Furthermore, Qiskit [23] provides a finance module

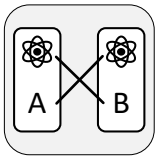
 ISBN: 978-1-68558-049-0

enabling portfolio optimization by implementing a transformer that takes a generic optimization problem as input and outputs a cost operator that can be used in a quantum algorithm. As many classical problems can be formulated as such an optimization problem, this can be used as a CLASSICAL-QUANTUM INTERFACE for different problem domains.

Related Patterns: The CLASSICAL-QUANTUM INTERFACE enables the integration of quantum algorithm implementations into applications. It can be used as an interface for a quantum algorithm implemented as a HYBRID MODULE. This interface provides a bridge between the different programming paradigms separated by the QUANTUM-CLASSIC SPLIT [16]. It is a special kind of FACADE [13] for quantum algorithms that not only hides the complexity of the algorithm, but also translates between the quantum computing domain and the problem domain.

E. Quantum Circuit Translator

Problem: *How can a quantum circuit be executed by different quantum computers with different instruction sets?*



Context: Quantum circuits can be implemented in different programming languages and with different quantum gates. However, quantum computers typically only support specific circuit formats and instruction sets, which hinders interoperability and leads to vendor lock-in [34]. Thus, executing a quantum circuit on different quantum computers often requires a translation of the quantum circuit.

Forces: There are a multitude of quantum programming languages available for implementing quantum algorithms [7]. A quantum circuit may be implemented in a programming language that is incompatible with the targeted quantum computer. The circuit needs to be re-implemented in a compatible quantum programming language and instruction set. However, a manual re-implementation is error-prone, time-consuming, and requires expertise in quantum computing, and, hence, is not feasible for real-world problem sizes. Therefore, an automatic translation, transforming unsupported gates into gates natively supported by the quantum computer, is required.

Solution: Use a translator to convert the quantum circuit into the target language and transpile the circuit to the target instruction set, i.e., replace unsupported gates with equivalent gates from the target instruction set.

The solution sketch in Figure 6 shows the application of a QUANTUM CIRCUIT TRANSLATOR that translates a quantum circuit between two programming languages and instruction sets. The SWAP gate connecting the outer qubit wires in the left quantum circuit has been decomposed into three C-NOT gates in the right target quantum circuit.

Result: A QUANTUM CIRCUIT TRANSLATOR is able to automatically translate a quantum circuit into a target format, enabling components with different circuit formats and instruction sets to use the same circuit. A QUANTUM CIRCUIT TRANSLATOR increases the reusability of QUANTUM MODULES, as it enables their use with different quantum computers. However, the translated circuits do not need to be executed directly, but can instead be used as inputs for a QUANTUM MODULE TEMPLATE. Therefore, a QUANTUM CIRCUIT TRANSLATOR enables the composition of quantum algorithms based on modules implemented in different programming languages. Thus, a QUANTUM CIRCUIT TRANSLATOR can be used to increase the interoperability of QUANTUM MODULES.

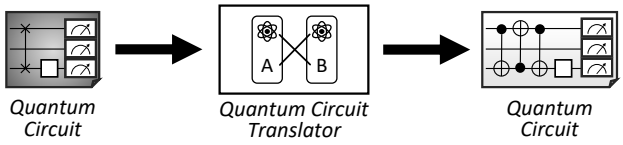


Figure 6. Solution sketch: QUANTUM CIRCUIT TRANSLATOR.

Known Uses: A widely used format for defining quantum circuits is OpenQASM [35], an open quantum assembly language. It can be imported and exported by many quantum software development kits (SDKs) such as Amazon Braket [24], Qiskit [23] and Cirq [36]. For estimating whether a quantum circuit can be executed, the *NISQ Analyzer* [34] needs the transpiled circuit for the respective quantum device. It includes multiple circuit translators. For the Python SDK Pennylane [37] there is a plugin enabling the support for IBM quantum computers without additional libraries. Explicit translation is supported by pytket [38] from and to Cirq [36]. Qconvert [39] can convert from pyQuil or OpenQASM to several other formats by using their web tool.

Related Patterns: The QUANTUM CIRCUIT TRANSLATOR pattern is related to the MESSAGE TRANSLATOR pattern from the enterprise integration pattern language [14]. With a CANONICAL DATA MODEL [14] quantum circuits of any language can be translated into any other language using at most two translators for each language. A circuit translator can be used to translate circuits generated by a QUANTUM MODULE implemented in one programming language before using them with a QUANTUM MODULE TEMPLATE implemented in a different programming language.

IV. DISCUSSION

The validity of a software engineering pattern strongly depends on the number of real-world uses of that pattern [11]. Each of the newly introduced patterns has a number of known real-world uses documented in the known uses section.

Except for quantum computing libraries aiming for a larger user base such as Qiskit, most quantum software development is currently ad-hoc, e.g., for a one-time experimental algorithm implementation without applying best practices from classical software engineering. The prevalence of ad-hoc development can be partly explained by the fact that only today's largest quantum computers have surpassed the amount of qubits that can be simulated on a classical computer [28]. Additionally, these qubits are still noisy, which further limits their potential applications [6]. Therefore, most quantum algorithms cannot

 ISBN: 978-1-68558-049-0

show their quantum advantage for relevant problem sizes on today's quantum computers. Thus, current implementations of quantum algorithms are often single-use, e.g., for a proof-of-concept, as the limited hardware available today can only process small problems. Therefore, the majority of examples of the QUANTUM MODULE and HYBRID MODULE patterns have been found in the larger quantum computing libraries. As stated above, implementing quantum algorithms requires a deep understanding of quantum computing, its mathematical foundations, and software engineering, which is a rare combination of skills. Thus, many implementations are created by physicists without a software engineering background.

Quantum algorithms are expected to be an essential part of many applications in various domains once they can solve problems of relevant size. Since applications integrating quantum algorithms do not depend on the algorithms' implementation details, the HYBRID MODULE pattern that can hide all this complexity inside the module will be useful here.

Splitting the implementation of an algorithm into multiple modules is an established technique used to reduce the complexity of an implementation. To refine this established design principle with quantum computing-specific requirements, we introduced the module patterns to the growing quantum software engineering discipline. The two patterns, QUANTUM MODULE and QUANTUM MODULE TEMPLATE, can be used to modularize quantum algorithm implementations. Modularization can be used on multiple levels. For example, QUANTUM MODULES can be used to build a HYBRID MODULE.

To ensure the interoperability of modules, they must expose an interface that can be used by other modules. A well-defined interface improves reusability and hides complexity, such that quantum algorithm implementations can be used without deep quantum computing knowledge. The CLASSICAL-QUANTUM INTERFACE pattern is crucial for creating algorithm implementations that ease the integration into existing applications.

The last pattern, the QUANTUM CIRCUIT TRANSLATOR, is mainly used for executing quantum circuits on different quantum computers. It is required as hardware vendors have not agreed upon a standard format for representing quantum circuits. OpenQASM [35] is at the moment the most promising candidate for such a format. However, even if all existing quantum computers can interpret OpenQASM, we will most likely still have many quantum programming languages with different properties. A QUANTUM CIRCUIT TRANSLATOR that translates quantum circuits between two such languages can also be used during the development of quantum algorithms. With such a translator it becomes possible to use QUANTUM MODULES implemented in other programming languages.

V. RELATED WORK

The patterns introduced in this work extend the existing quantum computing pattern language originally introduced by Leymann [16] which is continuously growing [2][17][18][19][20][21]. There are also other publications defining terms and summarizing concepts in the quantum computing domain [40][41]. However, these concepts are not documented as patterns in the sense of the definition provided by Alexander et al. [12] to guide developers in implementing quantum algorithms. Furthermore, Huang et al. [42] describe methods for validating quantum programs using anti-patterns.

Similar approaches documented for the field of quantum computing are also established in other areas of information technology. The QUANTUM CIRCUIT TRANSLATOR is based on the same concept as the MESSAGE TRANSLATOR presented in the enterprise integration patterns by Hohpe and Woolf [14]. It enables the communication between systems using different message formats. Other related enterprise integration patterns, such as the CANONICAL DATA MODEL and the NORMALIZER, can also be adapted to the quantum computing domain.

Leymann and Barzen [43] present the Pattern Atlas, a publicly available [44] tool to facilitate the visualization of connections between patterns within a pattern language as well as between different pattern languages. Moreover, it enables the creation of *Pattern Views* [45], i.e., a collection combining individual patterns and connections from different languages.

Sánches et al. [46] define the term *quantum module* and its properties to describe how to modularize the design of quantum circuits. The QUANTUM MODULE pattern defined in our work differs significantly, as it includes classical code to generate quantum circuits. However, the properties they identified for their quantum module definition must also be fulfilled by any behavior inputs to QUANTUM MODULE TEMPLATES.

Piattini et al. [9] outline the importance of quantum software engineering. They provide principles of quantum software engineering, e.g., that quantum software has a hybrid nature and suggest that reusable parts of quantum applications should be identified for creating libraries and to provide reference examples. The patterns presented in this work can be a part of the answer towards creating reusable quantum algorithm implementations. Beisel et al. [30] modularize recurring tasks of VQAs in various microservices and integrate them using workflows. Thereby, they follow established engineering concepts to provide VQAs as reusable, automatically executable workflows. Hence, their approach is an exemplary implementation of a HYBRID MODULE.

Georg et al. [21] describe the execution of quantum applications. The PRE-DEPLOYED EXECUTION and the PRIORITIZED EXECUTION pattern can be applied to execute a quantum application created by using the HYBRID MODULE pattern, which packages the algorithm together with classical dependencies inside a single module.

The quantum software lifecycle proposed by Weder et al. [22] describes the development process of quantum applications in ten phases. The patterns in this work document best practices for the implementation phases with more detailed instructions, e.g., for the *Hardware-independent Implementation* phase by providing a QUANTUM MODULE which has a generative classical part for building the quantum circuit, or for the *Quantum Circuit Enrichment* phase by using the QUANTUM MODULE and QUANTUM MODULE TEMPLATE to better split the responsibility for the implementation of oracles and state preparation.

 ISBN: 978-1-68558-049-0

VI. Conclusion and Future Work

Due to the novelty of quantum computing, there are so far few, if any, established principles for quantum software development. Implementing a quantum algorithm requires expert knowledge in quantum computing and software engineering, which makes it a multidisciplinary task. Currently, quantum applications are often implemented ad hoc by physicists without in-depth software engineering knowledge. Thus, this work extends the quantum computing pattern language by five new patterns to support quantum software engineers in their work by making relevant knowledge easily accessible and digestible. The four patterns in the development category, aiding developers in creating modularized and reusable implementations of quantum algorithms, incorporate knowledge from classical software engineering adapted to the quantum computing domain. The fifth pattern, Quantum Circuit Translator, is part of the execution category and enables quantum circuits to be executed on quantum computers of different vendors. Its main use case is related to the execution of quantum circuits, but it can also be used during development.

In the future, we will make the patterns available through the Pattern Atlas [43] repository integrated into the PlanQK platform [47]. This repository already contains all previously published patterns. By introducing the patterns to the public, we can receive valuable feedback in order to further refine the existing patterns. It allows a continuous re-evaluation of the patterns, including an analysis of the usability of the patterns. Especially in such a new and rapidly growing domain as quantum software engineering, any pattern language can be expected to evolve with the domain as new best practices emerge over time. Additionally, we plan to build a quantum computing solution language [48] that can provide concrete solutions to the problems presented in the pattern language.

ACKNOWLEDGEMENTS

This work was partially funded by the BMWK projects *SeQuenC* (01MQ22009B), *EniQmA* (01MQ22007B), and *PlanQK* (01MK20005N).

References

[1] M. Cerezo *et al.*, "Variational quantum algorithms," *Nature Reviews Physics*, vol. 3, no. 9, pp. 625–644, Aug. 2021.

[2] M. Weigold, J. Barzen, F. Leymann, and D. Vietz, "Patterns for Hybrid Quantum Algorithms," in *Proceedings of the 15th Symposium and Summer School on Service-Oriented Computing (SummerSOC 2021)*. Springer International Publishing, Sep. 2021, pp. 34–51.

[3] P. W. Shor, "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer," *SIAM Journal on Computing*, vol. 26, no. 5, pp. 1484–1509, oct 1997.

[4] J. Barzen and F. Leymann, "Continued Fractions and Probability Estimations in Shor's Algorithm: A Detailed and Self-Contained Treatise," *AppliedMath*, vol. 2, no. 3, pp. 393–432, Jul. 2022.

[5] L. K. Grover, "A fast quantum mechanical algorithm for database search," in *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, 1996, pp. 212–219.

[6] F. Leymann and J. Barzen, "The bitter truth about gate-based quantum algorithms in the NISQ era," *Quantum Science and Technology*, pp. 1–28, Sep. 2020.

[7] D. Vietz, J. Barzen, F. Leymann, and K. Wild, "On Decision Support for Quantum Application Developers: Categorization, Comparison, and Analysis of Existing Technologies," in *Computational Science – ICCS 2021*. Springer International Publishing, Jun. 2021, pp. 127–141.

[8] B. Weder, J. Barzen, F. Leymann, and M. Zimmermann, "Hybrid Quantum Applications Need Two Orchestrations in Superposition: A Software Architecture Perspective," in *Proceedings of the 18th IEEE International Conference on Web Services (ICWS 2021)*. IEEE, Sep. 2021, pp. 1–13.

[9] M. Piattini *et al.*, "The Talavera Manifesto for Quantum Software Engineering and Programming," Mar. 2020.

[10] E. Farhi, J. Goldstone, and S. Gutmann, "A Quantum Approximate Optimization Algorithm," Nov. 2014, arXiv:1411.4028.

[11] J. O. Coplien, *Software Patterns*. SIGS Books & Multimedia, 1996.

[12] C. Alexander, S. Ishikawa, and M. Silverstein, *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, Aug. 1977.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley, Oct. 1994.

[14] G. Hohpe and B. Woolf, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2004.

[15] C. Fehling, F. Leymann, R. Retter, W. Schupeck, and P. Arbitter, *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Springer International Publishing, Jan. 2014.

[16] F. Leymann, "Towards a Pattern Language for Quantum Algorithms," in *First International Workshop, QTOP 2019, Munich, Germany, March 18, 2019, Proceedings*. Springer International Publishing, Apr. 2019, pp. 218–230.

[17] M. Weigold, J. Barzen, F. Leymann, and M. Salm, "Data Encoding Patterns For Quantum Algorithms," in *Proceedings of the 27th Conference on Pattern Languages of Programs (PLoP '20)*. HILLSIDE, Oct. 2020, pp. 1–11.

[18] M. Weigold, J. Barzen, F. Leymann, and M. Salm, "Encoding patterns for quantum algorithms," *IET QuantumCommunication*, vol. 2, no. 4, pp. 141–152, Dec. 2021.

[19] M. Weigold, J. Barzen, F. Leymann, and M. Salm, "Expanding data encoding patterns for quantum algorithms," in *2021 IEEE 18th International Conference on Software Architecture Companion (ICSA-C)*. IEEE, Mar. 2021, pp. 95–101.

[20] M. Beisel *et al.*, "Patterns for Quantum Error Handling," in *Proceedings of the 14th International Conference on Pervasive Patterns and Applications (PATTERNS 2022)*. Xpert Publishing Services (XPS), Apr. 2022, pp. 22–30.

[21] D. Georg *et al.*, "Execution Patterns for Quantum Applications," in *Proceedings of the 18th International Conference on Software Technologies (ICSOFT 2023)*. SciTePress, 2023, accepted.

[22] B. Weder, J. Barzen, F. Leymann, M. Salm, and D. Vietz, "The Quantum Software Lifecycle," in *Proceedings of the 1st ACM SIGSOFT International Workshop on Architectures and Paradigms for Engineering Quantum Software (APEQS 2020)*. ACM, Nov. 2020, pp. 2–9.

[23] Qiskit, "Qiskit: An Open-source Framework for Quantum Computing," https://qiskit.org/documentation/, [accessed: 2023.05.25].

[24] AWS, "Amazon Braket Usage Overview," https://docs.aws.amazon.com/braket/latest/developerguide/braket-using.html, [accessed: 2023.05.25].

[25] K. Marquardt, "Patterns for Plug-ins," in *Proceedings of the Fourth European Conference on Pattern Languages of Programming and Computing*, 1999.

[26] D. Collins, K. W. Kim, and W. C. Holton, "Deutsch-Jozsa algorithm as a test of quantum computation," *Phys. Rev. A*, vol. 58, pp. R1633–R1636, Sep 1998.

[27] B. Weder, U. Breitenbücher, F. Leymann, and K. Wild, "Integrating Quantum Computing into Workflow Modeling and Execution," in *Proceedings of the 13th IEEE/ACM International Conference on Utility and Cloud Computing (UCC 2020)*. IEEE, Dec. 2020, pp. 279–291.

[28] H. Riel, "Quantum Computing Technology and Roadmap," in *ESSDERC 2022 - IEEE 52nd European Solid-State Device Research Conference (ESSDERC)*, 2022, pp. 25–30.

[29] Microsoft, "Microsoft Quantum Development Kit Overview," https://learn.microsoft.com/en-us/azure/quantum/overview-what-is-qsharp-and-qdk, [accessed: 2023.05.25].

[30] M. Beisel *et al.*, "Quokka: A Service Ecosystem for Workflow-Based Execution of Variational Quantum Algorithms," in *Service-Oriented Computing – ICSOC 2022 Workshops*. Springer, Mar. 2023,

 ISBN: 978-1-68558-049-0

Demonstration, pp. 369–373. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-031-26507-5_35
[31] Qiskit, "Qiskit Nature Overview," https://qiskit.org/documentation/nature/index.html, [accessed: 2023.05.25].
[32] AWS, "Amazon Braket Chemistry Example," https://github.com/aws/amazon-braket-examples/blob/main/examples/hybrid_quantum_algorithms/VQE_Chemistry/VQE_chemistry_braket.ipynb, [accessed: 2023.05.25].
[33] Microsoft, "Microsoft Quantum Development Kit: Quantum Chemistry Library," https://learn.microsoft.com/en-us/azure/quantum/user-guide/libraries/chemistry/, [accessed: 2023.05.25].
[34] M. Salm *et al.*, "The NISQ Analyzer: Automating the Selection of Quantum Computers for Quantum Algorithms," in *Proceedings of the 14th Symposium and Summer School on Service-Oriented Computing (SummerSOC 2020)*. Springer International Publishing, Dec. 2020, pp. 66–85.
[35] A. Cross *et al.*, "OpenQASM 3: A Broader and Deeper Quantum Assembly Language," *ACM Transactions on Quantum Computing*, vol. 3, no. 3, pp. 1–50, Sep. 2022.
[36] Google, "Cirq," https://quantumai.google/cirq, [accessed: 2023.05.25].
[37] V. Bergholm *et al.*, "PennyLane: Automatic differentiation of hybrid quantum-classical computations," 2018, arxiv:1811.04968.
[38] Cambridge Quantum, Quantum Software and Technologies, "Pytket Extensions," https://github.com/CQCL/pytket-cirq, [accessed: 2023.05.25].
[39] Quantastica, "Quantum programming language converter," https://github.com/quantastica/qconvert, [accessed: 2023.05.25].
[40] S. Perdrix, "Quantum patterns and types for entanglement and separability," *Electronic Notes in Theoretical Computer Science*, vol. 170, pp. 125–138, 2007.
[41] S. S. Gill *et al.*, "Quantum Computing: A Taxonomy, Systematic Review and Future Directions," 2020, arxiv:2010.15559.
[42] Y. Huang and M. Martonosi, "Statistical assertions for validating patterns and finding bugs in quantum programs," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 541–553.
[43] F. Leymann and J. Barzen, *Pattern Atlas*. Springer International Publishing, Apr. 2021, pp. 67–76.
[44] PlanQK, "PlanQK - Pattern Atlas," https://patterns.platform.planqk.de/pattern-languages, [accessed: 2023.05.25].
[45] M. Weigold *et al.*, "Pattern Views: Concept and Tooling of Interconnected Pattern Languages," in *Proceedings of the 14th Symposium and Summer School on Service-Oriented Computing (SummerSOC 2020)*. Springer International Publishing, Dec. 2020, pp. 86–103.
[46] P. Sánchez and D. Alonso, "On the Definition of Quantum Programming Modules," *Applied Sciences*, vol. 11, no. 13, 5843, 2021.
[47] PlanQK, "PlanQK - Platform and Ecosystem for Quantum Applications," https://platform.planqk.de/, [accessed: 2023.05.25].
[48] M. Falkenthal, J. Barzen, U. Breitenbücher, and F. Leymann, "Solution Languages: Easing Pattern Composition in Different Domains," *International Journal on Advances in Software*, vol. 10, no. 3, pp. 263–274, Dec. 2017.

 ISBN: 978-1-68558-049-0