When Abstraction Breaks Physics: Rethinking Modular Design in Quantum Software

Jianjun Zhao
Kyushu University, Japan
zhao@ait.kyushu-u.ac.jp

***Abstract*—Abstraction is a fundamental principle in classical software engineering, which enables modularity, reusability, and scalability. However, quantum programs adhere to fundamentally different semantics, such as unitarity, entanglement, the no-cloning theorem, and the destructive nature of measurement, which introduce challenges to the safe use of classical abstraction mechanisms. This paper identifies a fundamental conflict in quantum software engineering: *abstraction practices that are syntactically valid may violate the physical constraints of quantum computation*. We present three classes of failure cases where naive abstraction breaks quantum semantics and propose a set of design principles for physically sound abstraction mechanisms. We further propose research directions, including quantum-specific type systems, effect annotations, and contract-based module design. Our goal is to initiate a systematic rethinking of abstraction in quantum programming, based on quantum semantics and considering engineering scalability.**

***Index Terms*—Quantum software engineering, Abstraction, Modularity, Quantum programming, Quantum semantics**

I. Introduction

Abstraction is a fundamental principle in classical software engineering [1]. By hiding low-level implementation details and exposing high-level interfaces, abstraction enables modularity, reusability, and scalability, which are key elements for building complex systems [2], [3]. In classical programs, abstraction mechanisms such as functions, classes, and modules operate under assumptions of determinism, full observability, and compositional semantics [4], [5]. However, in the domain of quantum software engineering (QSE) [6], these assumptions do not hold.

Quantum programs are governed by fundamentally different physical principles, including unitarity, entanglement, the no-cloning theorem, and the destructive nature of measurement [7]. These properties introduce key challenges to the safe use of classical abstraction mechanisms. For example, encapsulating a subroutine composed of quantum gates may unintentionally violate global unitarity. Similarly, abstracting away the entangled states across modules may disrupt quantum correlations, leading to incorrect or non-executable behavior. Measurement operations, which collapse quantum states irreversibly, cannot be abstracted like classical observations without semantic misrepresentation.

Despite growing interest in high-level quantum programming languages and modular development frameworks [8]–[10], the foundational conflict between abstraction and quantum physical constraints has received little attention. Most existing languages focus on syntax or high-level constructs, but leave unresolved the question of whether such abstractions remain physically valid and semantically meaningful. In this paper, we argue that the concept of abstraction in quantum programs must be rethought under the lens of quantum semantics. We identify key failure cases where naive abstraction leads to physically invalid or semantically misleading behavior. Based on these observations, we propose a set of design principles for *physically sound abstraction mechanisms* that preserve unitarity, respect the boundaries of the entanglement, and account for the effects of the measurement. Our goal is to initiate a systematic rethinking of abstraction in QSE, laying the foundation for future language designs, toolchains, and modular development approaches.

II. Why Classical Abstraction Fails in Quantum Programs

Abstraction in classical software is enabled by assumptions that align with the semantics of deterministic computation. When a function is abstracted, its internal control flow and data manipulations are hidden, but its behavior remains predictable, reproducible, and compositionally safe. These abstractions facilitate reasoning, modularization, and code reuse, which form the basis of modern software engineering [4]. However, when these classical abstraction mechanisms are applied directly to quantum programs, they often fail due to fundamental mismatches between the semantics of classical computation and quantum mechanics. Table I summarizes the key differences between classical and quantum abstractions. These differences show that classical abstractions implicitly rely on properties such as observability, reusability, and determinism, which are not valid in quantum computation. For example, abstracting a quantum subroutine without considering its entanglement with external qubits can result in incorrect program behavior. Similarly, encapsulating measurement operations without making their side effects explicit violates the semantics of quantum state evolution. Moreover, classical abstraction assumes that modules can be composed without a global context. However, in quantum systems, whether a unitary transformation is valid often depends on the state of the entire system entangled, which makes local reasoning in general invalid.

In summary, the fundamental principles that form the basis of abstraction in classical software, such as safe composition, internal observability, and data duplication, are incompatible

TABLE I
CONTRASTING CLASSICAL AND QUANTUM ABSTRACTION ASSUMPTIONS

Aspect	**Classical Programs**	**Quantum Programs**
State representation	Concrete values (e.g., integers, booleans)	Superpositions of basis states in a Hilbert space
Observability	Full inspection of intermediate states	Intermediate states are not observable due to collapse
Determinism	Execution is deterministic	Execution is inherently probabilistic
Compositionality	Modules can be safely composed	Composition may break unitarity or entanglement integrity
Copying of data	Data can be freely copied	Qubits cannot be cloned due to the no-cloning theorem
Measurement effects	Observations are non-destructive	Measurement irreversibly collapses the quantum state

with the fundamental nature of quantum programs. These conflicts suggest that a physically sound notion of abstraction for quantum software must be based on the semantics of quantum mechanics, rather than classical computation.

III. Failure Cases of Naive Abstraction

While abstraction is a desirable engineering mechanism, applying it naively in quantum programs can result in physically invalid or semantically incorrect behavior. This section presents three representative failure cases observed in common quantum programming practices, each illustrating a fundamental conflict between abstraction and quantum semantics.

1) Violation of Unitarity: In classical programming, subroutines can encapsulate arbitrary computations. However, in quantum computing, unless measurement is involved, every operation must be unitary [7]. Encapsulating a non-unitary sequence, for example, conditionally applying gates based on classical logic, without proper control or ancillary handling, can result in an invalid transformation.

Listing 1. Encapsulated function may break unitarity (conceptual example)

```
def my_subroutine(circuit, q):
    if some_classical_condition():
        circuit.x(q[0])
        circuit.h(q[1])
```

In Listing 1, the subroutine `my_subroutine` encapsulates a classically controlled quantum operation. While syntactically valid in Python, such patterns risk violating global unitarity when classical control is evaluated outside the context of the quantum circuit. This breaks the assumption that all quantum subcircuits must be expressed as unitary transformations, unless they explicitly involve measurement or reset.

2) Entanglement Boundary Violation: Quantum entanglement leads to non-local dependencies between qubits. Abstracting a subcircuit that operates on an entangled subset of qubits without explicitly modeling the entanglement boundary can result in hidden dependencies or unintended side effects when the module is reused or rearranged [11].

Listing 2. Abstracted entanglement operation reused across contexts

```
def entangle_pair(circuit, q0, q1):
    circuit.h(q0)
    circuit.cx(q0, q1)

# Used in global context
entangle_pair(circ, q[0], q[1])
entangle_pair(circ, q[2], q[3])
```

In Listing 2, the abstraction `entangle_pair` is reused without awareness of the global entanglement structure. If other parts of the program also entangle `q1` or `q2` with other qubits, reusing this module may interfere with entanglement across modules and break the assumption that modules operate independently.

3) Violation of Measurement Semantics: Measurement in quantum computing is non-reversible and collapses quantum states [7]. Encapsulating measurements inside an abstract module may hide their global impact, especially when used within larger circuits or hybrid quantum-classical loops.

Listing 3. Measurement hidden inside abstraction

```
def measure_qubit(circuit, q, c, idx):
    circuit.h(q[idx])
    circuit.measure(q[idx], c[idx])
```

In Listing 3, the function `measure_qubit` performs a Hadamard transform followed by measurement on a single qubit. The use of encapsulation hides the non-reversible nature of measurement from the calling context, potentially invalidating the assumptions of later circuit segments. From the outside, the measurement side-effect is not evident, which violates compositional transparency.

In summary, these failure cases demonstrate that abstraction in quantum software must not only preserve syntactic validity but also maintain physical correctness. In the next section, we identify core design principles for abstraction mechanisms that are compatible with quantum semantics.

IV. Design Principles for Physically Sound Abstractions

To handle the conflicts discussed in Section III, we propose several design principles for abstraction mechanisms in quantum software. These principles aim to ensure that abstractions preserve the physical semantics of quantum computation and remain valid across modular compositions. Unlike classical abstraction guidelines that primarily target structural correctness and interface clarity, these principles are based on quantum physics and semantics.

1) Preservation of Unitarity: Any abstracted quantum subroutine must either (i) be representable as a unitary transformation over the involved qubits, or (ii) explicitly declare the presence of non-unitary effects such as measurement or reset. This requirement ensures that the composition of abstracted modules does not implicitly violate the core constraint of quantum evolution.

Implication: Language-level abstractions (e.g., circuit functions, macros) should enforce or annotate unitarity, and compilers should provide warnings if composed abstractions break global unitarity without explicit intent.

2) Entanglement Boundary Awareness: Abstractions must not hide entanglement relationships between qubits. When a module operates on qubits that are (or may be) entangled with external qubits, such dependences [11] should be made explicit, either through type annotations, interface contracts, or analysis tools [12].

Implication: Toolchains should support entanglement tracking across modules. Programmers should be aware that copying or reusing a subcircuit may unintentionally propagate or break entanglement structures.

3) Measurement Transparency: Measurement is a destructive operation that fundamentally alters the quantum state [7]. Any abstraction that performs measurement must make this fact explicit in its interface or usage contract. Implicit measurement within an encapsulated routine may break composability and violate the programmer's assumptions about the continuity of quantum state evolution.

Implication: Quantum domain-specific languages (DSLs) should distinguish between pure and measurement-containing modules, and runtime systems should prohibit implicit reuse of measurement abstractions in contexts that assume unitary evolution.

4) Interface-Driven Classical Interaction: In hybrid quantum-classical programs, the classical control flow must be handled via well-defined interfaces rather than opaque side effects. Abstracting classical logic together with quantum operations can obscure program semantics and break the deferred execution models used by many backends.

Implication: Classical logic should be modularized separately, and abstractions should clearly separate quantum operations from classical conditionals or callbacks that may affect circuit generation.

5) Physical Resource Compatibility: Abstract modules should not assume arbitrary qubit connectivity or reuse without regard to underlying hardware constraints. Since physical qubit layout and coherence times affect execution fidelity, abstractions must be designed to either (i) remain hardware-agnostic and compilable, or (ii) declare hardware assumptions explicitly [13].

Implication: The extraction layers must support qubit allocation hints, routing awareness, or post-mapping validation to ensure compatibility of the real device.

In summary, these principles provide a foundation for designing future quantum programming languages and toolchains that support modularity while preserving physical correctness. Rather than mimicking classical abstraction practices, quantum software engineering must develop its own abstraction discipline aligned with quantum mechanics.

V. Rethinking Abstraction Mechanisms

The need for physically sound abstraction mechanisms may open a new frontier in quantum software engineering, whereas classical software relies on decades of well-established design patterns and modeling formalisms [1], [14], [15], quantum programming demands fundamentally different abstractions that are compatible with the laws of quantum mechanics. In this section, we present several research directions that could serve as a foundation for future quantum abstraction mechanisms.

1) Type Systems for Abstraction Semantics: One promising direction is the development of quantum-specific type systems that encode abstraction constraints [16]. For example, a type system could statically enforce whether a module preserves unitarity, modifies entanglement, or performs measurement. These types would serve not only as documentation, but also as compile-time guarantees for physically valid behavior.

Example 1. *A quantum subroutine* `apply_oracle` *may be annotated with a type like* `Unitary :: Qubit[n] -> Qubit[n]`*, indicating that it is a unitary transformation that preserves quantum coherence. Conversely, a function like* `measure_qubits` *may be typed as* `Measuring :: Qubit[n] -> Bit[n]`*, making it clear that it performs a destructive measurement and returns classical results.*

2) Effect Systems and Side-Effect Annotations: Quantum abstractions may benefit from effect annotations that track semantic side effects such as measurement, entanglement, and classical branching. These annotations can help developers reason about the behavior of modules without examining internal implementation and allow compilers or analyzers to catch violations of expected behavior.

Example 2. *A subroutine could be annotated with* `@measuring`*, indicating that it collapses part of the quantum state. Another might use* `@entangles(q0, q1)` *to indicate that it introduces entanglement between two qubits. These annotations would allow downstream tools to enforce constraints, such as forbidding the reuse of measured qubits in a coherent subcircuit.*

3) Contract-Based Module Design: Inspired by design-by-contract principles in classical software engineering [17], quantum modules could specify preconditions and postconditions based on quantum mechanics, such as constraints on unitarity, entanglement, or measurement. These contracts would capture semantic constraints regarding the quantum state before and after execution, allowing tools or simulators to validate the correctness of usage.

Example 3. *A modular subroutine implementing a quantum kernel may declare that a contract like* `requires: q[0] and q[1] are separable`*, meaning that it expects the two qubits to be unentangled before invocation. Another example might specify* `ensures: q[0] is disentangled after execution`*, guiding a safe modular composition in larger quantum programs.*

4) Abstraction-Aware Tool Support: The practicality of physically sound abstraction mechanisms depends on the corresponding tooling infrastructure. Tool support can provide feedback during development and debugging, especially when physical correctness is not visually obvious.

- **Static analyzers**: Tools that scan quantum programs for entanglement leakage, unintended measurements, or violations of declared abstraction types [12].
- **Visualization tools**: Interfaces that allow developers to inspect entanglement graphs and abstraction boundaries, improving their understanding of the program.
- **Verification frameworks**: Lightweight or formal tools to check whether a composed module preserves unitarity or whether all measurement effects are visible in abstraction contracts [18].

By embedding physical awareness into language constructs and toolchains, these approaches would allow developers to use abstraction safely, promoting modularity and scalability without compromising correctness. As quantum programs grow in complexity, such mechanisms will be essential not only for reliability but also for collaboration, maintenance, and long-term evolution of quantum software systems.

VI. Related Work

Recent quantum programming languages and frameworks emphasize modularity and scalability. Qiskit [19] supports modular circuit construction through Python functions and the `QuantumCircuit` class, but it does not provide checks for unitarity, entanglement boundaries, or measurement side effects, leaving these to the programmer. Q# [10] introduces higher-level abstractions with adjoint and controlled operations, and separates classical from quantum data. However, it does not enforce rules for measurement visibility or entanglement safety, which can lead to misuse. Silq [8] improves safety through automatic resource management and ancilla clean-up, but does not track measurement or entanglement, focusing mainly on resource safety. Quipper [9] generates scalable circuits using functional programming and parameterized abstractions, but its abstractions are mainly syntactic and do not validate physical correctness. ProjectQ [20] and Cirq [21] provide circuit modularization but lack mechanisms to enforce quantum semantics. Zayas Gallardo et al. [22] proposed *Locus*, a programming abstraction for composing circuits with support for deferred instantiation and modular reuse. While their focus is on a concrete mechanism to improve compositionality in Qiskit, our work addresses the more fundamental issue of ensuring that abstraction mechanisms remain consistent with quantum semantics and physical constraints.

Several studies have explored modularity in quantum software, primarily through modeling rather than direct implementation. An early effort proposed a UML-based quantum software modeling language [23], which emphasized modularity. This was followed by several other modeling approaches [24]–[26]. However, these modeling languages do not adequately handle physical constraints such as unitarity and entanglement. A formal definition of quantum programming modules was proposed in [27], focusing on structural composition and encapsulation, but without considering quantum-specific concerns such as measurement collapse or entanglement preservation. Di Matteo [28] further emphasized the general importance of abstraction in quantum software, but her discussion was limited to high-level perspectives and tool support rather than addressing semantic consistency with quantum physics.

To the best of our knowledge, no existing quantum programming language or framework systematically addresses physically sound abstraction, i.e., preserving unitarity, maintaining entanglement locality, and making measurement effects explicit. Our work is the first to identify this as a key challenge in quantum software engineering. Rather than proposing new language features, we argue that abstraction itself should be rethought in light of quantum constraints. This complements ongoing efforts in quantum language design, compiler optimization, and verification by focusing on an overlooked aspect: the correctness of abstraction under physical constraints.

VII. Conclusion and Future Work

Abstraction is a fundamental principle in classical software engineering, but directly applying it to quantum programming faces key challenges. We argue that classical abstraction mechanisms based on determinism, observability, and compositionality are often incompatible with quantum physical semantics. We identify three types of failure cases where naive abstraction violates unitarity, fails to expose measurement, or disrupts entanglement. Based on these observations, we propose design principles for *physically sound abstraction*, guided by the constraints of quantum mechanics. These include preserving unitarity, exposing measurement effects, maintaining entanglement locality, and aligning abstractions with hardware and classical interfaces. For future work, we propose several directions for developing languages and tools that align with these principles. These include type systems that capture abstraction semantics, effect annotations for physical side effects, contract-based module design, and tool development for entanglement-aware validation and visualization.

As quantum software systems grow in size and complexity, the development of such abstraction mechanisms will be crucial to the reliability, reusability, and engineering scalability. We hope that this work could initiate a broader effort toward formalizing abstraction mechanisms that are both physically valid and practically useful in quantum software engineering.

References

[1] I. Sommerville, *Software Engineering, 9/E*. America: Pearson Education Inc, 2011.

[2] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.

[3] M. Shaw, "Prospects for an engineering discipline of software," *IEEE Software*, vol. 7, no. 6, pp. 15–24, 1990.

[4] C. Ghezzi, M. Jazayeri, and D. Mandrioli, *Fundamentals of Software Engineering*, 2nd ed. Prentice Hall, 2003.

[5] B. Meyer, *Object-oriented software construction*. Prentice hall Englewood Cliffs, 1997, vol. 2.

[6] J. Zhao, "Quantum software engineering: Landscapes and horizons," arXiv preprint arXiv:2007.07047, 2020.

[7] M. A. Nielsen and I. L. Chuang, *Quantum computation and quantum information*. Cambridge university press, 2010.

[8] B. Bichsel, T. Gehr, D. Drachsler-Cohen, and M. Vechev, "Silq: A high-level quantum language with safe uncomputation and intuitive semantics," in *Proc. PLDI 2020*, 2020, pp. 286–300.

[9] A. S. Green, P. L. Lumsdaine, N. J. Ross, P. Selinger, and B. Valiron, "Quipper: A scalable quantum programming language," *Proc. ACM Program. Lang.*, vol. 2, no. ICFP, pp. 103:1–103:29, 2018.

[10] K. M. Svore, A. Geller, M. Troyer, J. Azariah, C. Granade, B. Heim, V. Kliuchnikov, M. Mykhailova, A. Paz, M. Roetteler, D. S. Simon, and M. Soeken, "Q#: Enabling scalable quantum computing and development with a high-level DSL," in *Proc. Real World DSLs*, 2018.

[11] H. Yu and J. Zhao, "The quantum program dependence graph and its uses in quantum software development," in *Proceedings of the International Workshop on Quantum Software Engineering (Q-SE 2025)*. IEEE/ACM, 2025, to appear.

[12] S. Xia and J. Zhao, "Static entanglement analysis of quantum programs," in *2023 IEEE/ACM 4th International Workshop on Quantum Software Engineering (Q-SE)*. IEEE, 2023, pp. 42–49.

[13] Y. Shi, P. Gokhale, P. Murali, J. M. Baker, C. Duckering, Y. Ding, N. C. Brown, C. Chamberland, A. Javadi-Abhari, A. W. Cross *et al.*, "Resource-efficient quantum computing by breaking abstractions," *Proceedings of the IEEE*, vol. 108, no. 8, pp. 1353–1370, 2020.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH, 1995.

[15] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of computer programming*, vol. 8, no. 3, pp. 231–274, 1987.

[16] A. Paszke, G. Kaas, and M. Martonosi, "Qlinear: A linear type system for quantum programming with explicit measurement and entanglement," *arXiv preprint arXiv:2303.12094*, 2023. [Online]. Available: https://arxiv.org/abs/2303.12094

[17] B. Meyer, *Applying "Design by Contract"*. Prentice Hall, 1992.

[18] M. Alpay, R. Rand, and S. Zdancewic, "QWIRE: Reasoning about quantum circuits," in *Proceedings of Quantum Physics and Logic (QPL)*, 2020. [Online]. Available: https://qpl2020.org/slides/qwire.pdf

[19] Qiskit Development Team, "Qiskit: An open-source framework for quantum computing," https://qiskit.org, 2024.

[20] D. S. Steiger, T. Häner, and M. Troyer, "ProjectQ: An open source software framework for quantum computing," *Quantum*, vol. 2, p. 49, 2018.

[21] Q. A. team and collaborators, "Cirq," https://github.com/quantumlib/Cirq, 2022, accessed: 2024-06-11.

[22] J. Zayas Gallardo, F. Chicano, C. Canal, and J. M. Murillo, "Locus: A proposal for quantum software composition," in *Companion Proceedings of the 9th International Conference on the Art, Science, and Engineering of Programming (Programming 2025)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2025, pp. 17–1.

[23] C. A. Pérez-Delgado and H. G. Perez-Gonzalez, "Towards a quantum software modeling language," in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, 2020, pp. 442–444.

[24] X. Guo, S. Saito, and J. Zhao, "QuanUML: Towards a modeling language for model-driven quantum software development," in *2025 IEEE 49th Annual Computers, Software, and Applications Conference (COMPSAC)*, 2025, pp. 1344–1349.

[25] R. Pérez-Castillo and M. Piattini, "Design of classical-quantum systems with UML," *Computing*, vol. 104, no. 11, pp. 2375–2403, 2022.

[26] S. Ali and T. Yue, "Modeling quantum programs: challenges, initial results, and research directions," in *Proceedings of the 1st ACM SIGSOFT International Workshop on Architectures and Paradigms for Engineering Quantum Software*, 2020, pp. 14–21.

[27] P. Sánchez and D. Alonso, "On the definition of quantum programming modules," *Applied Sciences*, vol. 11, no. 13, p. 5843, 2021.

[28] O. Di Matteo, "The art of abstraction in quantum software," in *2025 IEEE/ACM International Workshop on Quantum Software Engineering (Q-SE)*. IEEE, 2025, pp. 25–26.