Enhancing Quantum Deep Q-Learning with Aspect-Oriented Programming: Cross-Cutting Optimization

EUSTACHE MUTEBA A.[1], NIKOS E. MASTORAKIS[2]
English Language Faculty of Engineering
Technical University of Sofia & INTERBIT
Obelya 1, Block 118
1387, Sofia, Bulgaria
BULGARIA

Abstract: - This paper proposes an enhancement to the Contract-based Quantum Deep Q-Learning (QDQL) model through the integration of Aspect-Oriented Programming (AOP), a paradigm that enables the clean separation of contract enforcement logic from the core learning agent. In this approach, aspects act as modular interceptors that transparently apply contracts (i.e., domain rules or constraints) during the agent's decision-making process. To facilitate the structured and scalable integration of these enforcement mechanisms within the Quantum Deep Q-Learning architecture, the use of design patterns is introduced as a formal method for defining both the structural organization and behavioral interactions of system components. As a practical use case, the approach is applied to adaptive oncology treatment recommendation, where Aspect-Oriented Programming AOP provides a principled and modular means of enforcing critical clinical constraints, such as compliance with medical protocols, ethical standards, and patient-specific conditions, without tightly coupling them to the core learning algorithm.

Key-Words: - Aspect-Oriented Programming, Cross-Cutting Concerns, design patterns Quantum Deep Q-Learning, Hybrid Systems, Oncology.

Received: June 19, 2025. Revised: July 28, 2025. Accepted: August 19, 2025. Published: September 23, 2025.

1 Introduction

In recent years, quantum computing has attracted growing interest within the scientific community, particularly among computer science researchers. Promising major advances in solving complex problems, such as factoring large numbers, optimizing, and simulating physical or medical systems, this new approach is disrupting traditional computing paradigms [1], [2], [3].

Also, in problem solving using computers, the choice of programming paradigm is not neutral: it guides the way in which a problem is modeled, solved and optimized [4], [5]. Mastering several paradigms allows one to choose the approach most suited to the situation, which is essential for designing effective, robust, and maintainable solutions [6].

That is why the aspect-oriented programming (AOP) paradigm was introduced to address some of the limitations of the object-oriented programming (OOP) paradigm, particularly in the management of cross-cutting concerns [7], [8].

Our recent work on Contractual Quantum Deep Q-Learning [9] highlights the need for formal contracts between quantum and classical components. Contracts are integrated (i.e. explicit specifications / constraints) into the QDQL system, so that classical and quantum modules behave according to agreed rules (e.g., the action selection must satisfy certain safety or fairness constraints, etc.).

The issue of contracts had already been addressed in [10] in the context of software agents. These contracts define operational, performance, or structural constraints that must hold for the system to behave correctly, safely, and efficiently.

Following our previous study and recognizing that contracts intersect with various concerns, such as safety, ethics, performance, and resource constraints, across multiple stages of the agent lifecycle, Aspect-Oriented Programming (AOP) emerges as an ideal approach for implementing a contract-based Quantum Deep Q-Learning (QDQL) system. By treating contracts as cross-cutting concerns, AOP enables their seamless integration into the QDQL agent, enhancing modularity, maintainability, and reusability.

The paper proposes an enhancement to the Contract-based Quantum Deep Q-Learning (QDQL) model through the integration of Aspect-Oriented Programming (AOP), a paradigm that enables the clean separation of contract enforcement logic from the core learning agent. In this approach, aspects act as modular interceptors that transparently apply contracts (i.e., domain rules or constraints) during the agent's decision-making process.

Aspect-Oriented Programming principles can normally be applied to the formal Quantum Deep Q-Learning Contract by modularizing each contract specification into cross-cutting concerns (aspects) that can be weaved into the Quantum Deep Q-Learning agent's runtime behavior.

To support the systematic integration of contract enforcement mechanisms into the Quantum Deep Q-Learning architecture, the use of design patterns is proposed as a means of formalizing the structural and behavioral composition of the system.

Finally, we explore the application of these patterns in adaptive oncology treatment recommendation, as a concrete use case. In this clinical domain, AOP offers a principled and modular mechanism to enforce critical constraints, including adherence to medical protocols, ethical guidelines, and patient-specific factors, without entangling the core learning algorithm with domain-specific logic.

2 Problem Formulation

The development of hybrid quantum-classical systems, such as Quantum Deep Q-Learning (QDQL) agents, introduces architectural and methodological challenges related to system correctness, safety, and maintainability. This section identifies the core problems that arise when designing contract-based QDQL systems and justifies the need for Aspect-Oriented Programming (AOP) and design patterns as a means to address them.

2.1 Contractual Requirements in QDQL Systems

Quantum Deep Q-Learning agents integrate classical decision-making components with quantum modules that perform sampling, optimization, or value estimation [11], [12]. In such systems, formal contracts are required to regulate interactions between quantum and classical parts. These contracts can specify constraints on behavior (e.g., fairness in action selection), performance (e.g., bounded response times), or safety (e.g., avoiding dangerous states) [9].

However, encoding these contracts directly into the learning logic results in scattered and tangled code, violating principles of separation of concerns and increasing the difficulty of verifying and maintaining system correctness over time [1].

2.2 Cross-Cutting Nature of Contracts

Contracts in QDQL systems are inherently cross-cutting concerns: they span multiple phases of the agent lifecycle (e.g., policy learning, action selection, feedback interpretation) and influence multiple components (e.g., environment interface, classical controller, quantum evaluator) [13], [14].

2.3 Limitations of Existing Integration Approaches

Previous attempts to integrate contract logic into agent-based systems have relied on imperative or declarative annotations, rule-based engines, or middleware components [14].

While these approaches offer some level of abstraction, they often lack:

- Modularity: Contract logic is duplicated across multiple modules.
- Reusability: There is no mechanism to package and reuse contract behaviors.
- Transparency: Contract enforcement is not consistently visible in the system architecture.
- Scalability: As the number of contracts increases, so does the complexity of integration.

In the specific context of QDQL systems, where quantum computations introduce non-deterministic and probabilistic behavior, these limitations become even more pronounced [12], [1].

2.4 Opportunity for Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) is well-suited to addressing cross-cutting concerns by allowing developers to define aspects, modular units that encapsulate behavior affecting multiple classes, and to weave them into the system at well-defined join points [15], [16].

AOP enables:

- Clear separation of contract logic from core learning algorithms.
- Non-invasive enforcement of constraints at runtime.
- Greater modularity and reusability through reusable aspect modules.

Given the diversity of contracts (e.g., performance, ethics, safety), their integration via AOP could provide a unified and extensible way to manage them in QDQL systems.

2.5 Need for Design Patterns in AOP-Based QDQL

While AOP provides the underlying mechanism for modular contract enforcement, there is currently no formalized methodology or design pattern for applying AOP to quantum-classical learning architectures.

Without such patterns, developers lack guidance on how to:

- Structure aspects that correspond to different types of contracts.
- Identify appropriate join points in QDQL agents.
- Integrate AOP seamlessly with quantum computation workflows.

Design patterns could fill this gap by providing reusable templates for structuring contract aspects, composing them with agent behavior, and ensuring system-wide consistency [13]. In particular, patterns are needed to formalize the interaction between quantum evaluators and classical policy components, where contract enforcement often resides.

3 Hybrid Paradigm for Cross-Cutting Optimization

To address the limitations identified in the previous section, we propose a hybrid architectural paradigm that combines Aspect-Oriented Programming (AOP) with Quantum Deep Q-Learning (QDQL) in a modular and contract-aware fashion. This paradigm supports cross-cutting optimization by treating contract specifications, such as safety, fairness, or performance constraints, as first-class modular concerns that can be defined, managed, and woven into the agent's behavior at runtime.

3.1 Formalization of Contracts

Each contract is defined as a tuple:

$$C = \langle S, A, P, E \rangle \qquad (1)$$

Where:

- S: The set of system states to which the contract applies;
- A: The set of actions or behaviors the contract constrains or monitors;
- P: The predicates or rules to be satisfied (e.g., safety, fairness, resource bounds);
- E: The enforcement logic, including responses to violations (e.g., abort, modify, log).

3.2 Modularizing Cross-Cutting Contracts as Aspects

The modularization of contracts uses standard AOP constructs adapted to the QDQL environment:

1° Pointcuts: Define the join points in the QDQL lifecycle where contract enforcement is needed.

Typical join points include:

- *before(action_selection)*: Apply constraints before an action is chosen.
- *after(reward_update)*: Validate learning after reward propagation.
- *around(quantum_sampling)*: Wrap quantum subroutine execution to validate inputs/outputs.

2°Advice: Defines the behavior to be executed at the matched pointcut:

- *before advice*: Pre-conditions such as input validation, safety checks.
- *after advice*: Logging, state verification, performance monitoring.
- *around advice*: Policy enforcement, contract modification, fault recovery.

3° Aspects: Encapsulate pointcuts and advice into a reusable, composable module representing a specific contract.

3.3 Design Patterns for Contract Weaving

Design patterns are reusable solutions to common design problems occurring within a given context in software development. Originating from the architectural principles established by Alexander et al. [17]. Design patterns provide a proven, abstracted vocabulary for modeling recurring software design challenges and enable the modularization of complex interactions between the agent, its learning components, and the contract aspects.

In quantum-classical hybrid systems such as QDQL, design patterns can serve a critical role in managing the complexity arising from the integration of classical reinforcement learning pipelines with quantum circuit components [18].

Suggested patterns are presented in the following.

a) Core QDQL Agent

Pattern Name: QDQL Agent
Intent: Generates actions based on learned policy
Pseudo Code:

```
// Core QDQL agent
class QDQLAgent:
function core_qdql(state):
   # compute action
   return chosen_action
```

b) Aspect

Pattern Name: Aspect
Intent: Decorator/interceptor wrapping core agent, applies contracts dynamically
Pseudo Code:

```
// Aspect decorator: wraps core_qdql_function
class Aspect decorator:
function aspect(contract):
   function decorator(core_qdql_function):
      function wrapper(state):
         action = core_qdql_function(patient_state)
         contracted_action = contract(action, state)
         return contracted_action
```

```
        return wrapper
    return decorator
```

c) Contract

Pattern Name: Contract
Intent: Pure functions encoding domain rules, returning modified or validated actions
Pseudo Code:

```
//Contract: receives action and state, returns modified action
class Contract
function contract(action, state):
    if violates_rule(action, state):
        return modify_action(action, state)
    else:
  return action
```

d) Multiple Contracts Composed in Chains

Pattern Name: Contract Chain Composition
Intent: Allow multiple contracts to be applied sequentially to the QDQL agent's actions, enabling complex validations/modifications by composing simple contracts.
Pseudo Code:

```
// --- Contract Chain ---
// Applies multiple contracts sequentially
class ContractChain:
    contracts  // list of contracts

    method apply(action, state):
        current_action = action
        for contract in contracts:
            modified_action = contract(current_action, state)
            if modified_action is None:
                // Contract rejects the action
                return None
            current_action = modified_action
        return current_action
```

e) Dynamic Contract Activation

Pattern Name: Conditional Contract Activation
Intent: Enable runtime decision on which contracts should be applied based on the environment, agent state, or external conditions.
Pseudo Code:

```
// --- Dynamic Contract Activation ---
// Contracts have predicates to check if they are active
class DynamicAspect:
    contract_pool  // list of (predicate, contract) tuples

    method intercept(action, state):
        applicable_contracts = []
        for (predicate, contract) in contract_pool:
            if predicate(state) == True:
                applicable_contracts.append(contract)

        if applicable_contracts is empty:
            return action  // no contracts active

        chain = ContractChain(applicable_contracts)
        return chain.apply(action, state)
```

f) Conflict Resolution Strategies

Pattern Name: Contract Conflict Resolver
Intent: When multiple contracts modify an action in conflicting ways (e.g., one increases dosage, another decreases), a resolution strategy reconciles differences to ensure consistency.
Pseudo Code:

```
// --- Conflict Resolution Strategies ---
class PriorityContract:
    contract
    priority  // higher number = higher priority

class ConflictResolver:
    priority_contracts  // list of PriorityContract objects sorted by priority desc

    method resolve(action, state):
        for pc in priority_contracts:
            modified_action = pc.contract(action, state)
            if modified_action is not None:
                return modified_action
        return action

// --- Aspect with Conflict Resolution ---
class AspectWithConflictResolution:
    priority_contracts  // list of PriorityContract objects

    method intercept(action, state):
        resolver = ConflictResolver(priority_contracts)
   return resolver.resolve(action, state)
```

4 Case Study: Adaptive Oncology Treatment Recommendation

4.1 Context and Objective

In clinical decision-making, selecting an optimal treatment often involves balancing multiple, potentially conflicting objectives, such as cost, toxicity, effectiveness, and patient quality of life (QoL).

This case study demonstrates the application of runtime weaving using aspect-oriented programming (AOP) to dynamically inject multi-objective contract logic into a core treatment selection algorithm. The goal is to show how cross-cutting concerns, in this case, ethical and clinical policy constraints, can be applied selectively and transparently at runtime, enhancing adaptability and explainability.

4.2 Experimental Scenario

Three synthetic patient profiles were defined in table 1 to simulate common clinical scenarios. These patients vary in toxicity index, cancer stage, and available budget, affecting how the system chooses treatments based on runtime conditions.

Table 1: Patient profiles

Patient	Age	Toxicity Index	Cancer Stage	Budget (€)	Previous Treatments
P1	60	0.3	II	€ 7,00	None
P2	70	0.7	III	€ 3,50	Chemotherapy
P3	75	0.9	IV	€ 15,00	Radiotherapy, Immunotherapy

4.3 System Overview

The system is implemented in Python and is composed of four primary components:

1° Core Decision Logic (*core_qdql*): A selector that chooses the treatment based on cost. In this case, it picks the most expensive treatment (just as a placeholder behavior).

2° Aspect (*multi_objective_aspect*): A runtime-injected decorator that overrides the core logic using a weighted multi-criteria scoring system. It evaluates each treatment based on:

- Cost minimization
- Toxicity minimization
- Effectiveness maximization
- QoL preservation

3° Condition Function (*high_toxicity_condition*): Governs whether the aspect is applied. The contract aspect is only woven into the execution flow if the patient's toxicity index exceeds a configurable threshold.

4° Weaving *(@conditional_aspect)*: The conditional aspect decorator is used to perform runtime weaving, seamlessly directing the flow toward either the aspect-enhanced or original logic.

This creates a runtime-weaved decision function:

- It checks the condition
- If true → weaves the aspect (contract logic) into the call
- If false → runs the core logic only

4.4 Results

The decision-making process for each patient is visualized through both textual and graphical outputs. When the aspect is applied, all treatments are scored in real-time, and the action with the highest aggregate score (expressed as a percentage) is selected.

```
Patient 1: Age 60, Toxicity Index 0.3
[Conditional Aspect] Condition NOT met, using core logic.
→ Final recommended treatment:  surgery
```

```
Patient 2: Age 70, Toxicity Index 0.7
[Conditional Aspect] Condition met, applying contract aspect.

[Aspect] Entering multi-objective contract aspect
[Aspect] Core QDQL suggested initial action: 'surgery'
[Contract] Evaluating actions with weights: {'cost': 0.25, 'toxicity': 0.25, 'effectiveness': 0.4, 'quality_of_life': 0.1}
  [Contract] Action 'chemotherapy ': Score = 36.5%  (Cost: 6000, Toxicity: 1, Effectiveness: 0.6, QoL: 0.00)
  [Contract] Action 'radiotherapy ': Score = 63.2%  (Cost: 3000, Toxicity: 0.3, Effectiveness: 0.5, QoL: 0.70)
  [Contract] Action 'immunotherapy': Score = 60.7% (Cost: 10000, Toxicity: 0.3, Effectiveness: 0.8, QoL: 0.70)
  [Contract] Action 'surgery      ': Score = 36.0% (Cost: 12000, Toxicity: 1, Effectiveness: 0.9, QoL: 0.00)

[Aspect] Best action selected: 'radiotherapy' (63.2%)
[Aspect] Exiting multi-objective contract aspect

→ Final recommended treatment: radiotherapy
```

```
Patient 3: Age 75, Toxicity Index 0.9
[Conditional Aspect] Condition met, applying contract aspect.

[Aspect] Entering multi-objective contract aspect
[Aspect] Core QDQL suggested initial action: 'surgery'
[Contract] Evaluating actions with weights: {'cost': 0.25, 'toxicity': 0.25, 'effectiveness': 0.4, 'quality_of_life': 0.1}
  [Contract] Action 'chemotherapy ': Score = 36.5%  (Cost: 6000, Toxicity: 1, Effectiveness: 0.6, QoL: 0.00)
  [Contract] Action 'radiotherapy ': Score = 63.2%  (Cost: 3000, Toxicity: 0.3, Effectiveness: 0.5, QoL: 0.70)
  [Contract] Action 'immunotherapy': Score = 60.7% (Cost: 10000, Toxicity: 0.3, Effectiveness: 0.8, QoL: 0.70)
  [Contract] Action 'surgery      ': Score = 36.0%  (Cost: 12000, Toxicity: 1, Effectiveness: 0.9, QoL: 0.00)

[Aspect] Best action selected: 'radiotherapy' (63.2%)
[Aspect] Exiting multi-objective contract aspect

→ Final recommended treatment: radiotherapy
```

4.5 Runtime Weaving Visualization

The figure 1 is a diagram that clarifies the runtime composition of logic, showing where and how contracts are applied, improving transparency and traceability.

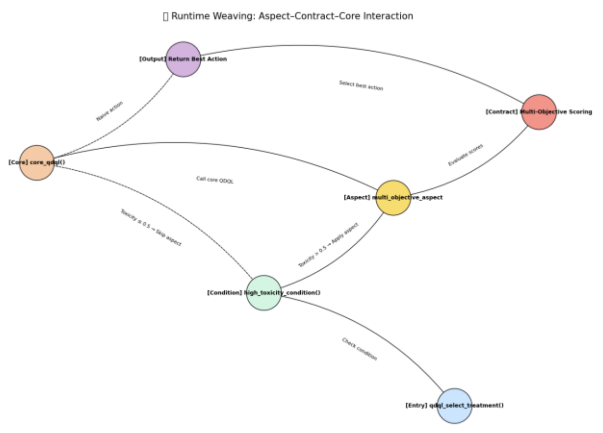
Runtime Weaving: Aspect–Contract–Core Interaction
[Output] Return Best Action
Select best action
[Contract] Multi-Objective Scoring
Naive action
[Core] core_qdql()
Evaluate scores
Call core QDQL
[Aspect] multi_objective_aspect
Toxicity ≤ 0.5 → Skip aspect
Toxicity > 0.5 → Apply aspect
[Condition] high_toxicity_condition()
Check condition
[Entry] qdql_select_treatment()

Fig. 1. Runtime weaving

5 Conclusion

This work demonstrates how Aspect-Oriented Programming (AOP) can be effectively employed to modularize and dynamically apply contract enforcement in the context of Quantum Deep Q-Learning (QDQL).

Through the integration of well-established design patterns, the architectural composition of the system becomes both systematic and extensible, supporting the principled insertion of aspects as modular decision influencers.

The case study in adaptive oncology treatment recommendation validates this methodology, showing how critical clinical and ethical guidelines can be enforced transparently, depending on real-time patient contexts. By decoupling domain-specific policies from the core learning agent, the proposed approach enables runtime weaving of constraints, such as cost-efficiency, toxicity management, and treatment effectiveness, without sacrificing the agent's generality or adaptability.

More broadly, this approach offers a pathway to building explainable, auditable, and policy-compliant intelligent systems, especially in high-stakes domains like healthcare, where decision traceability and contract adherence are paramount. By enabling runtime adaptation of cross-cutting concerns, the fusion of AOP with QDQL lays the groundwork for more trustworthy and context-aware reinforcement learning applications.

References:

[1] Preskill J., Quantum computing in the NISQ era and beyond, *Quantum*, Vol. 2, 2018, pp. 79. Available online : https://doi.org/10.22331/q-2018-08-06-79

[2] Arute, F., Arya, K., Babbush, R. et al., Quantum supremacy using a programmable superconducting processor, *Nature* 574, 2019, pp. 505–510. Available online : https://doi.org/10.1038/s41586-019-1666-5

[3] Montanaro A., Quantum algorithms: An overview, *npj Quantum Information*, Vol. 2, Article 15023, 2016. Available Online: https://doi.org/10.1038/npjqi.2015.23

[4] Van Roy P., Haridi S., *Concepts, Techniques, and Models of Computer Programming*, Cambridge, MA: MIT Press, 2004.

[5] Ghezzi C., Jazayeri M., Mandrioli D., *Fundamentals of Software Engineering*, 2nd ed. Upper Saddle River, NJ: Prentice Hall, 2002.

[6] Krishnamurthi S., Teaching programming languages in a post-Linnaean age, *J. Funct. Program*, Vol. 14, no. 4, 2004, pp. 387–402. Available Online: https://doi.org/10.1017/S0956796803004992

[7] Kiczales G. et al., Aspect-oriented programming, *in Proc. Europ. Conf. Object-Oriented Program*, (ECOOP), Jyväskylä, Finland, Jun. 1997, pp. 220–242. Available Online: https://doi.org/10.1007/BFb0053381

[8] Filman R. E., Friedman D. P., Aspect-oriented programming is quantification and obliviousness, *Workshop on Advanced Separation of Concerns, OOPSLA*, Minneapolis, MN, USA, 2000.

[9] Muteba A. E., Mastorakis N. E., Contractual Quantum Deep Q-Learning, *WSEAS Transactions on Computers*, Vol. 24, 2025, pp. 142-147.

[10] Muteba A. E., *Modelling Software Agents for Medical Decision Support System: Agent Builder Framework*, Le livre en papier, 2024.

[11] Nielsen M. A., Chuang I. L., *Quantum Computation and Quantum Information*, Cambridge University Press, 2010.

[12] Rebentrost P., Mohseni M., Lloyd S., Quantum support vector machine for big data classification, *Phys. Rev. Lett.*, vol. 113, no. 13, 2014, pp. 130503.

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

[14] Jennings N., et al., Autonomous agents for contract negotiation and management, *International Journal of Autonomous Agents and Multi-Agent Systems*, vol. 9, no. 3 , 2004, pp. 235–263.

[15] Filman R. E. et al., *Aspect-Oriented Software Development*, Addison-Wesley, 2005.

[16] Elrad T., Filman R. E., Baderv, Aspect-Oriented Programming: Introduction, *Communications of the ACM*, vol. 44, no. 10, 2001, pp. 29–32.

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

[18] Khan M. A., Software design patterns and architecture patterns - A study explored, *in Proc. IEEE Int. Conf. on Comput. Intell. and Informatics (ICCI)*, 2022.

Contribution of individual authors to the creation of a scientific article (ghostwriting policy)

The authors equally contributed in the present research, at all stages from the formulation of the problem to the final findings and solution.

Sources of funding for research presented in a scientific article or scientific article itself

The article was produced without specific funding.

Creative Commons Attribution License 4.0 (Attribution 4.0 International , CC BY 4.0)

This article is published under the terms of the Creative Commons Attribution License 4.0
https://creativecommons.org/licenses/by/4.0/deed.en_US