Classic Design Patterns applied for Hybrid Information Systems

The main goal of these folders is trying to adapt some of the classic structural design patterns to sample dummy hybrid systems. These systems are basically classic host code (written in Python) that needs some results provided by quantum algorithms (written in Q#). Remark that they are dummy examples just to illustrate the main idea of integrating the classic patterns in hybrid information systems.

Structural Patterns

Proxy Pattern

For the *proxy pattern* we have an example where we have a Client class that does a Request to a proxy. This proxy then checks whether there exists a RealSubject class (which calls the Quantum Algorithm). If not, it creates a RealSubject instance and calls the Request method. Finally, the RealSubject instance does the Quantum Algorithm.
Note that both RealSubject and Proxy are children classes from the abstract class Subject, since both deal with the subject (which in our case is the Request for the quantum result).

Regarding the Quantum Algorithm, it simply creates a superposition and returns a measurement.

Check figure 1 from Appendix for the UML diagram.

Facade Pattern

For the *facade pattern* the example consists of a system that can call to some Quantum Algorithms by means of *Subsystems* (SubsystemX, SubsystemY, SubsystemZ, SubsystemH). Each subsystem has a method calling the real Quantum Algorithm. Here the application of the facade pattern takes place in a class that collects some “related” methods together offering them grouped in two combinations (DoPauli and DoXnH methods in the Facade class).

There are four Quantum Algorithms for this example, but they are quite simple. We have one algorithm applying the Pauli X gate to a qubit and returning a measurement. The same for The Pauli Y, Pauli Z and the Hadamard gate.

Check figure 2 from Appendix for the UML diagram.

Flyweight Pattern

For the *flyweight pattern* we have an example where a FlyweightFactory is the one in charge of managing the different Flyweights instances. Each flyweight calls the real Quantum

Algorithm in the method Operation. When the Factory receives a request for a Flyweight it checks whether it exists in its flyweights registry (indexed by a key). If not, it creates one and adds it to the registry with the corresponding key.
Note that the ConcreteFlyweight inherits from an abstract class Flyweight. The main purpose of this is opening the possibility for adding more types of Flyweights giving other kinds of "service".

Finally, the Quantum Algorithm again consists just in creating a superposition and returning a measurement.

Check figure 3 from Appendix for the UML diagram.

Decorator Pattern

For the *decorator pattern* the example consists of a system that can execute three Quantum Algorithms (OpX, OpY and OpZ). The point here for the decorator pattern is that, individually, each component (both the ConcreteComponent or any of the two ConcreteDecorators) has a single operation. But we want to add to a ConcreteDecoratorX instance the functionality of doing the Pauli Y and Z Quantum Gates wrapped by the ConcreteDecoratorY and ConcreteComponent respectively. To do so, we use first add to an instance of the ConcreteDecoratorY (concrete_deco_Y) the functionality of the ConcreteComponent, and then add the concrete_deco_Y to the ConcreteDecoratorX instance (concrete_deco_X).

Regarding the Quantum Algorithms implementation of OpX, OpY and OpZ they are just creating a qubit, appling the Pauli X, Y and Z gates respectively to the created qubit.

Check figure 4 from Appendix for the UML diagram.

Composite Pattern

For the *composite pattern* let's suppose that we have a system that has a hierarchy of operations. In each operation we call a Quantum Operation. The desired behavior for the system is to be able to execute all the operations in the hierarchy, being the Entanglement operation for the Composite nodes and the OpX the operation for the Leaf nodes.
For this approach the Composite pattern allows us to define an abstract class Component that represents any node in the tree. Then, we can have as in any tree, Leafs (without children) and Composite (which is in fact a subtree with more components).
Note that we can build the tree as we need by means of the methods Remove and Add and then execute the operational hierarchy designed by means of calling Operation on the desired initial node (the root or any other component).

The Quantum Algorithms are OpX and Entanglement. The OpX just creates a qubit, applies the Pauli X gate and returns a measurement on the qubit. And the Entanglement method creates two qubits and entangles them for returning a measurement of both qubits.

Check figure 5 from Appendix for the UML diagram.

Behavioral Patterns

State Pattern

For the *state pattern* the example consists of a system that needs certain quantum operations depending on a state, that can be either for performing a measurement over a superposition or over an entanglement. The state switches from one to the other after each call to the quantum operation. To do this, we can see the application of the State pattern. To do the operations we have to call the Operation method from the Context class. This executes the operation on the state that is currently active. We have two possible states that inherit from the State abstract class including the operations needed for any child of this class (a setter for specifying which context instance is the one in charge of doing the transition after each operation and the operation itself). The two children classes for the State abstract class are ConcreteStateSuperposition and ConcreteStateEntanglement.

In order to test the correct behavior of the system, we use the for loop with 4 iterations.

Regarding the Quantum Algorithms, we just have Superposition, which creates a qubit, puts it in superposition and measures it, and Entanglement, which creates a pair of qubits, entangles them and returns a measurement of both of them.

Check figure 6 from Appendix for the UML diagram.

Observer Pattern

For the *observer pattern*, let's suppose that we want to update a couple of instances (ConcreteObserver) each time a heavy Quantum Algorithm is performed. For this we have the Subject abstract class, from which the ConcreteSubject class inherits some methods for managing the observers, and itself it adds the functionality for calling the real Quantum Algorithm and a getter method for accessing the state.
The state of this class is just the number of Quantum Operations performed.
The Observer class is an abstract class from which the ConcreteObserver inherits a method for managing the updates from the ConcreteSubject class, and also adds a setter method for specifying the ConcreteSubject instance is interested in being updated.

Regarding the Quantum Algorithm, QuantumAlg just creates a couple of qubits, it entangles them and returns a measurement of both. But, in order to simulate the time consuming aspect, I added a delay of 3 seconds after calling the Quantum Algorithm.

Check figure 7 from Appendix for the UML diagram.

Appendix

Figure 1

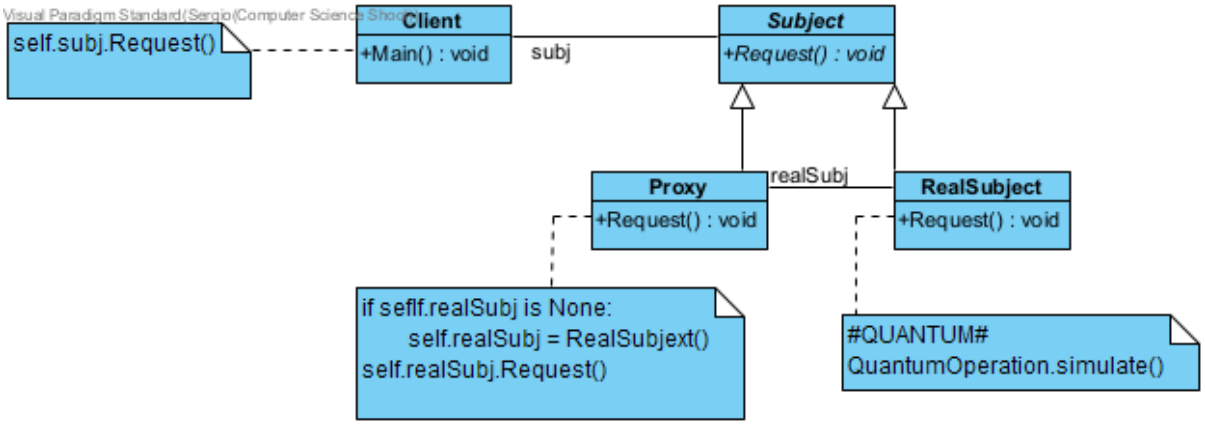


Figure 2

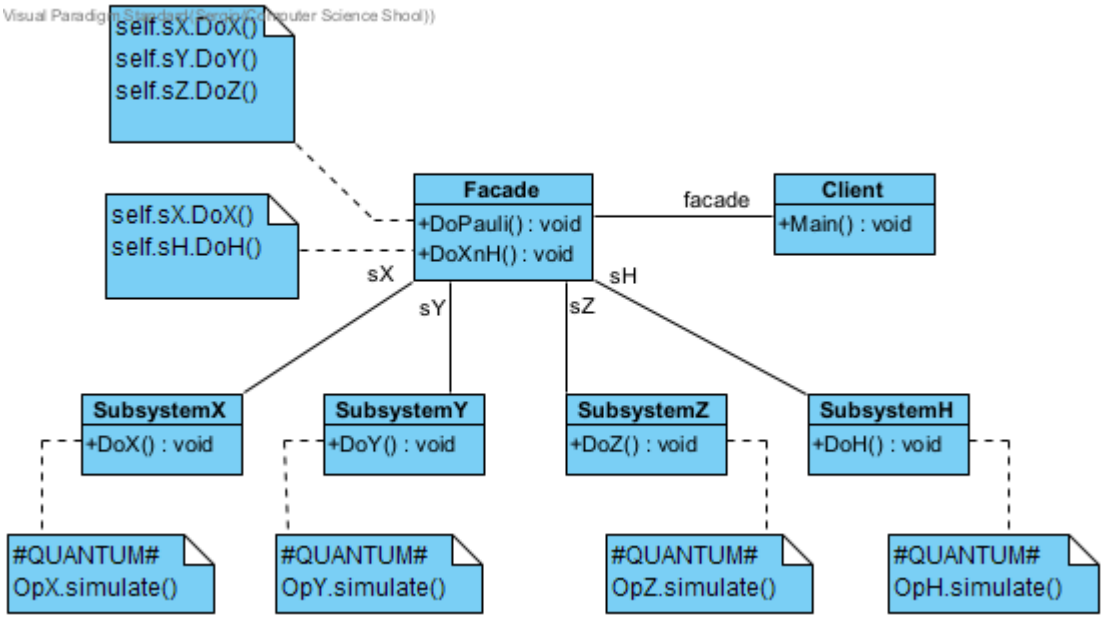


Figure 3

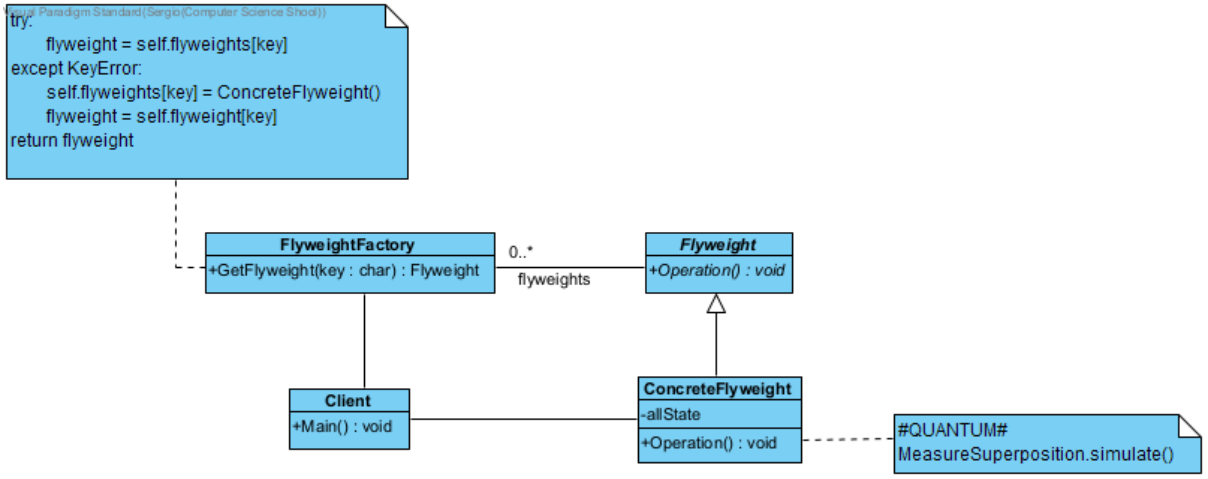


Figure 4

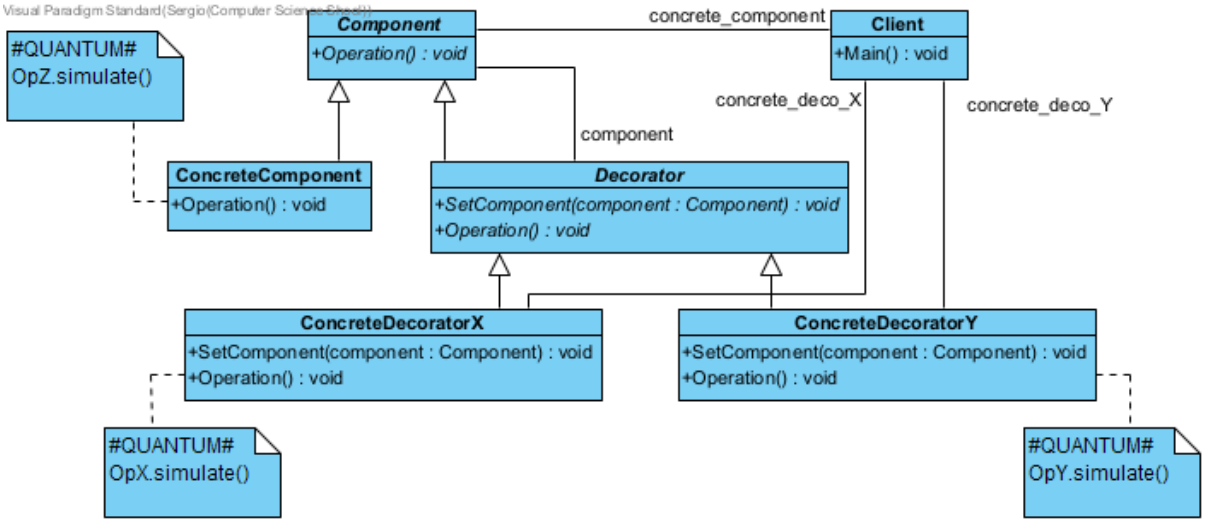


Figure 5

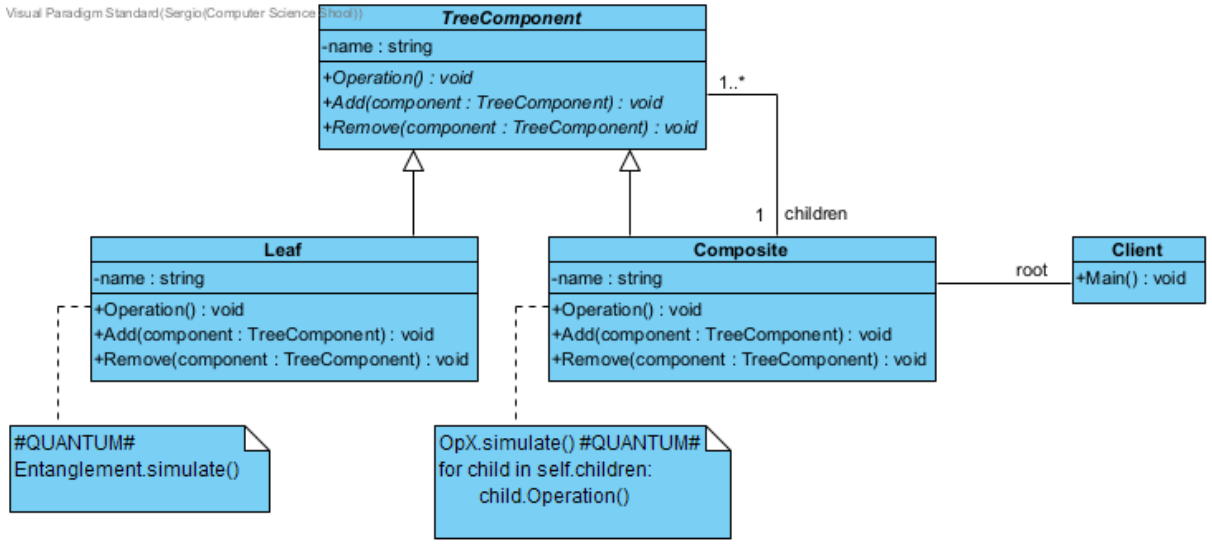


Figure 6

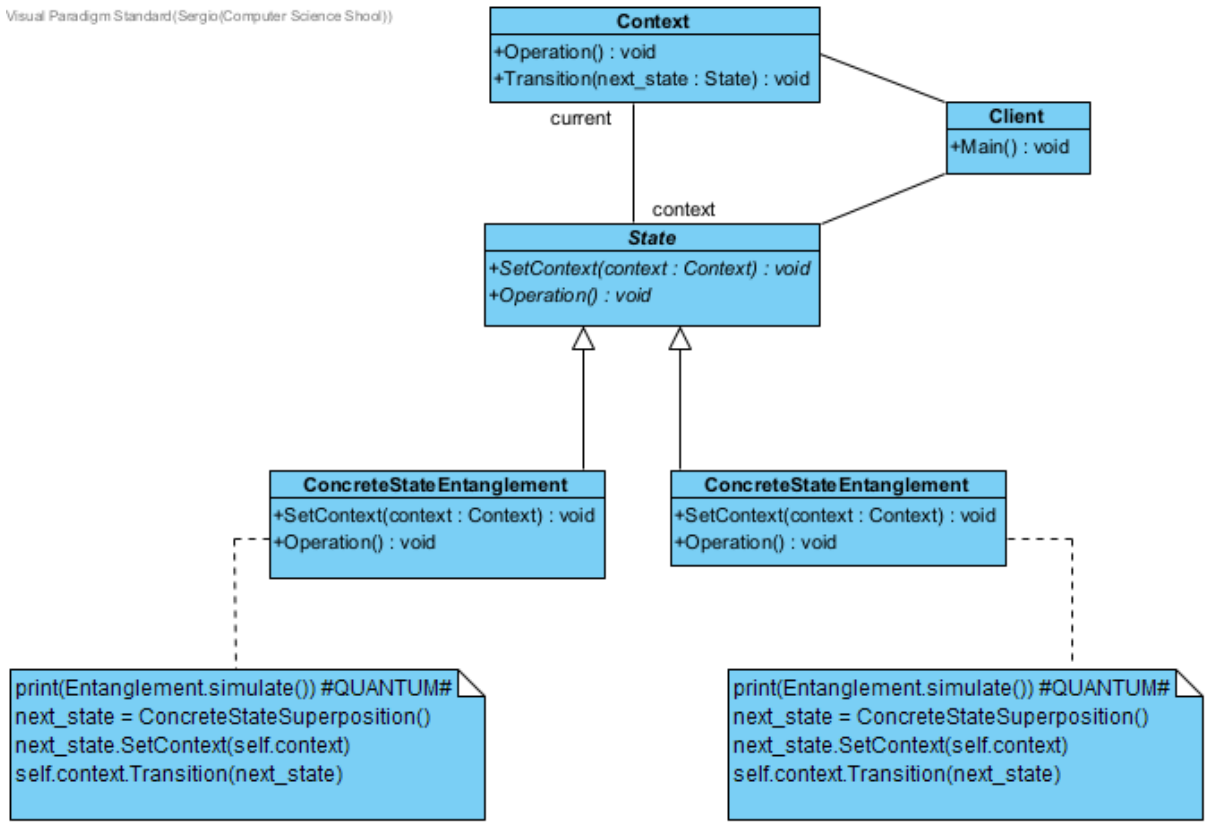
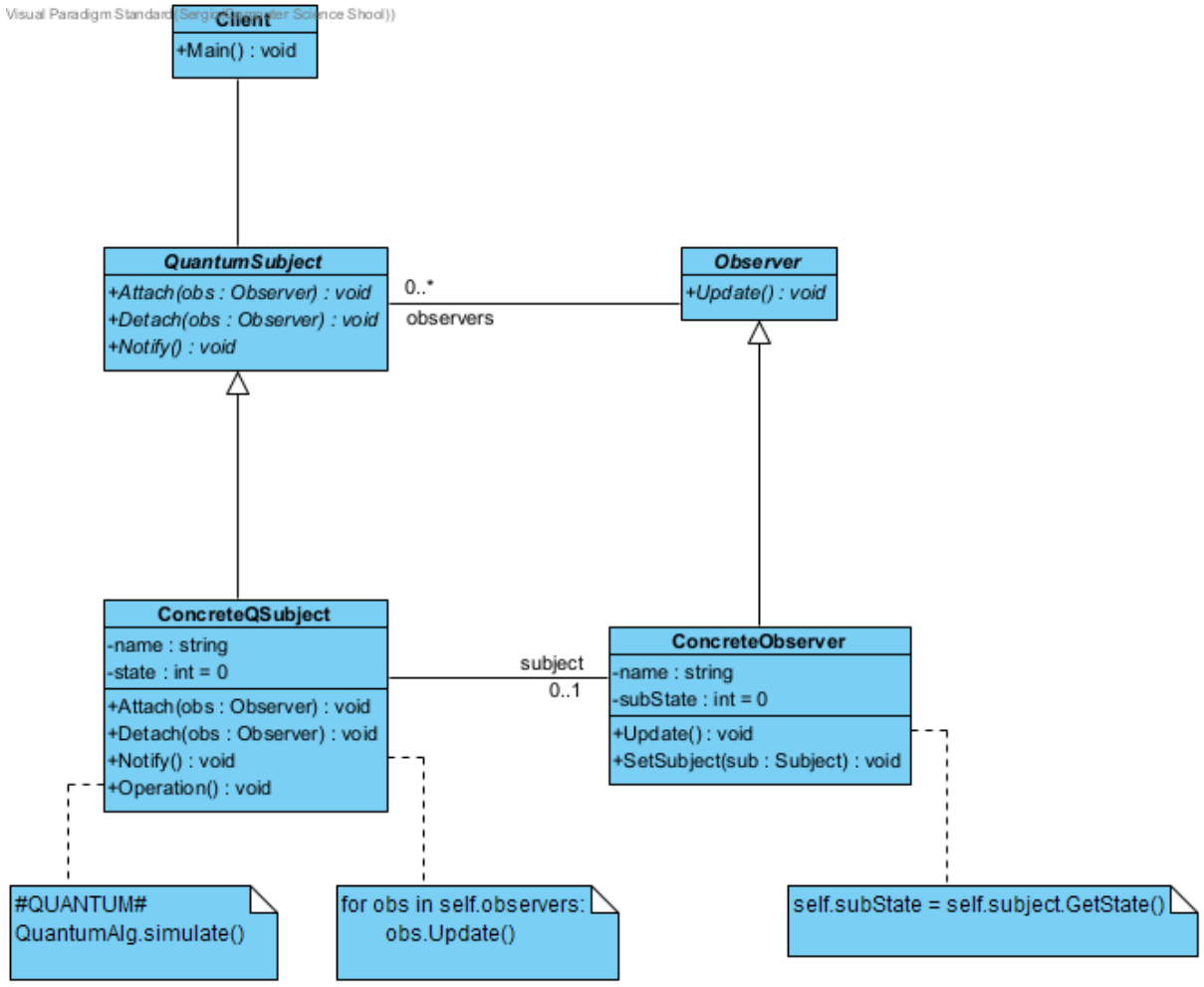


Figure 7


Client
+Main() : void
QuantumSubject
+Attach(obs : Observer) : void
+Detach(obs : Observer) : void
+Notify() : void
0..*
observers
Observer
+Update() : void
ConcreteQSubject
-name : string
-state : int = 0
+Attach(obs : Observer) : void
+Detach(obs : Observer) : void
+Notify() : void
+Operation() : void
subject
0..1
ConcreteObserver
-name : string
-subState : int = 0
+Update() : void
+SetSubject(sub : Subject) : void
#QUANTUM#
QuantumAlg.simulate()
for obs in self.observers:
obs.Update()
self.subState = self.subject.GetState()