

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.

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.

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.

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.

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, applying the Pauli X, Y and Z gates respectively to the created qubit.