**Rationale of my implementation**

**Project structure**

I structured this project in 3 packages, besides the test. Most important one contains everything related with instruments and his management. Here the important is the class InstrumentModuleManager that contains all instances of our instruments and some methods that should be invoked from outside to manage and use them. It also contains a different sub package, containing everything related with modules. Finally there is a third package called commands that represents the different operations that we can apply to the modules that belong to the previously created instruments.

**Patterns and SOLID principles used**

**Command design pattern**

For apply operations to different modules that makes the instrument, I propose a implementation using Command design pattern. The idea of have a different class for each different behavior allows us to fulfill the different solids principles and makes our system extensible and easy to maintain. It also abstracts the behavior from the entities and allows us to apply one behavior to more than one entity in runtime execution.

**Singleton pattern**

This pattern restricts the instantiation of a class. It must provide global access point to get the instance of the class. In our code example its applied to ensure we just have one instance of the class InstrumentModuleManager

**Open/Close principle**

Instrument contains a list of commands and a list of modules where this commands will be applied. To add new functionalities to existing instrument, we just need to insert a new command or module in the instrument. Our previous code will be isolated and not affected, and as all new commands fulfill our contract, they are compatible with existing code.

**Single responsibility**

Our command will just do one operation over a given element.

**LISKOV replacement principle**

In my proposal is any extension of an existing class, so we are not using this principle.

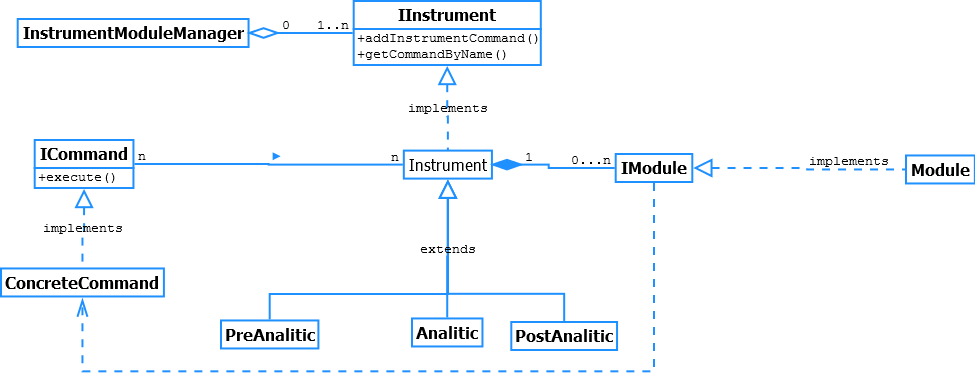
**Interface Segregation Principle**

Other SOLID principle used was the interface segregation principle. As you can see in the code skeleton, all the objects that manage class instances have just interfaces. These interfaces just contain the real needed public methods and all other things are encapsulated in the classes that implements this interfaces or in abstract classes for common things.

**Dependency inversion principle**

Dependencies (in our case modules) are passed in the concrete commands by injection. Best point to see this in my code is when we instantiate the commands, were we are passing the modules where they will apply parameter in the constructor. It allows us to mock these objects for testing purposes and to isolate operations of the definition of the modules.

**UML Class Diagram**

****