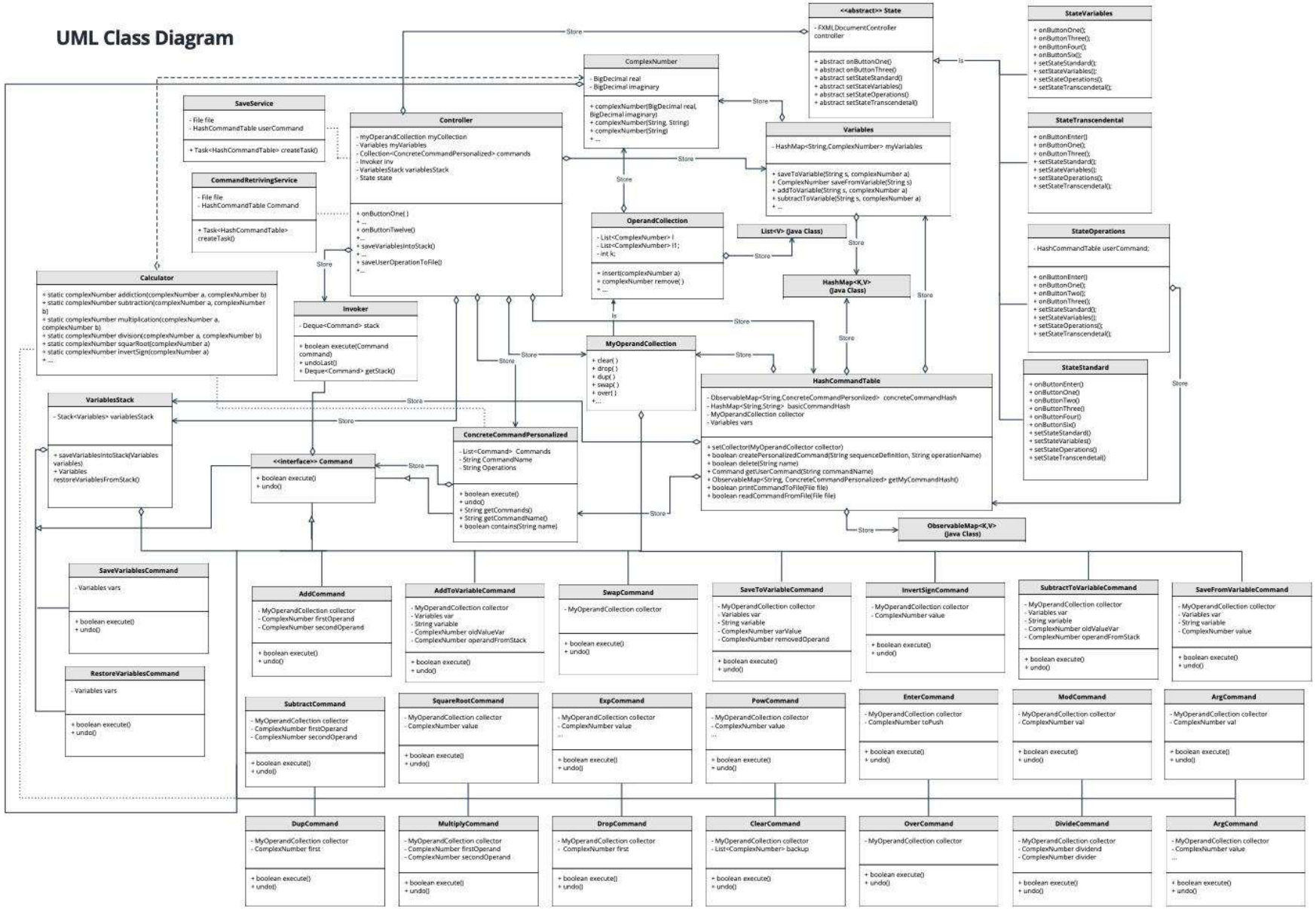
UML Class Diagram



This is UML Class Diagram of our system. During the design of the architecture, we tried to reach a lower level of coupling between the various units and a high level of cohesion in a single unit, in order to achieve the characteristics of completeness and consistency.
To achieve this goal the application of some design patterns helped us.
The patterns we decided to apply are:

- Command Pattern;
- State Pattern;
- Composite Pattern;

The **Command Pattern** is a behavioral design pattern that turns a request into a stand-alone object that contains all information about the request. The reason we decided to take advantage of this pattern was to respect the Principle of Separation of Concerns in order to break the app into layers.
In this way all the operations, instead of being a simple method, became objects containing all the information about the operations.
To apply this pattern have been identified:

- The Command Interface that declares the method of “execute” and of “undo”.
- The Concrete Commands, those associated with basic operations, and those associated with user defined operations.
- The Invoker or Sender that is responsible for initiating requests, so it has a reference to command objects. It triggers those commands instead of sending the request directly to the receiver.
- The Receiver that contains some business logic and that in some cases does itself the actual work, and the command only handle the detail of how a request is passed to the receiver.

By applying this pattern, we certainly got the benefit of respecting the Principle of Single Responsibility, decoupling classes that invoke operations from classes that perform these operations, and the possibility of introducing new commands into the app without modifying a lot of the existing code.
Obviously, the code has become more complex because a layer has been inserted between sender and receiver, and because this has led to a proliferation of classes.
In this way we were able to implement the undo mechanisms of all the operations performed by the Users.

The **State Pattern** is a behavioral design pattern closely related to the concept of a Finite-State Machine, so it is related with the idea that at any given moment there's a finite number of states which a program can be in, within any unique state, the program behaves differently and can be switched from one state to another instantaneously, and these transitions between different states are finite and predetermined.
We use this pattern in order to manage the different sections present on our graphical user interface, the components that are present or hidden in each of them and the different behaviors associated to each of them relating to the state in which the interface is.
By applying this pattern have been avoided complex “if statements” which would have burdened the Controller class.
To apply this pattern have been identified:

- The Context that contains a concrete state object and delegates to it all state-specific work and communicates with it via the State Interface.
- The State Interface that declares the state-specific methods that would have sense for all concrete states.
- The different Concrete States which contain a reference to context object, and which provide their own implementations for the state specific method.

Context and concrete states can set the next state of the context and perform the state transition by replacing the state object linked to the context.
Applying this pattern we respect the Single Responsibility Principle, organizing the code related to states into separate classes, we reached the possibility of introducing new states without changing existing state classes.

The **Composite Pattern** is a structural design pattern which we considered useful in our architecture even if not studied in depth in class. In fact, this pattern guarantees the possibility to compose objects into tree structures and then work with these structures as if they were individual objects
Applying this pattern was key to handling user-defined operations, composed by basic commands or by other user defined commands, and considering them as commands, calling on it the execute or the undo methods.
In this way we also solve the problem of perform the undo in cascade of already performed component commands when, executing a user defined operation, the remaining operands were not enough.
To apply this pattern have been identified:

- The Component Interface that corresponds to Command Interface, which describes operations common to both simple and complex operations.
- The Leaf that corresponds to basic command already implemented by Calculator
- The Container, that correspond to our ConcretePersonalizedCommand class, which has sub-elements, leaves or other containers. In this way, upon receiving a request a Container delegates the work to its sub elements, processes intermediate results and then returns the final result to client.

In this way new element types will be introduced into the app without breaking existing code.