DESIGN patterns

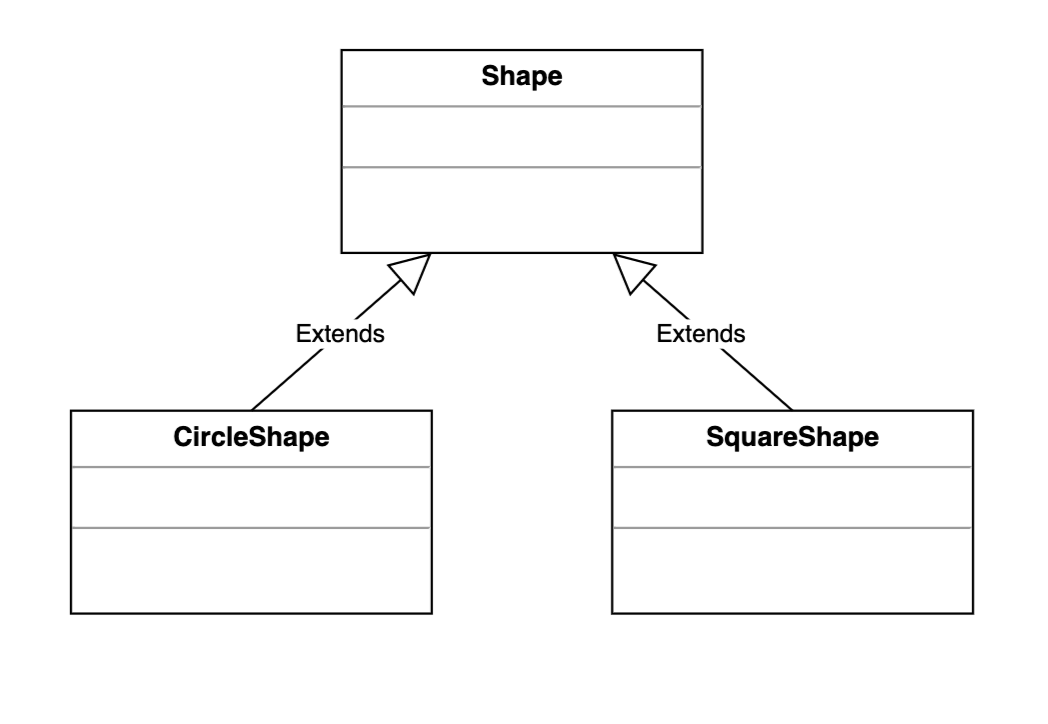
## Hypothesizes

* X, Y coordinates of the center of the shape are positive integers.
* Relative X, relative Y values are integers.
* Dimension is a positive float that represents the radius for the circle, and the side for the square.
* The factor for scaling is a positive integer.
* The action redo is not possible until an undo action is executed.
* If a new action is done after an undo, the stack of undo and redo actions is reinitialized.

## *Model*

In order to create different shapes that have similar behavior, an abstract super class is created that contains the logic shared between all the subclasses. The class is created as abstract to prevent creating an instance with no defined type.

Two subclasses are created in this case, one represents a circle and another one represents a square. The model is as follows:

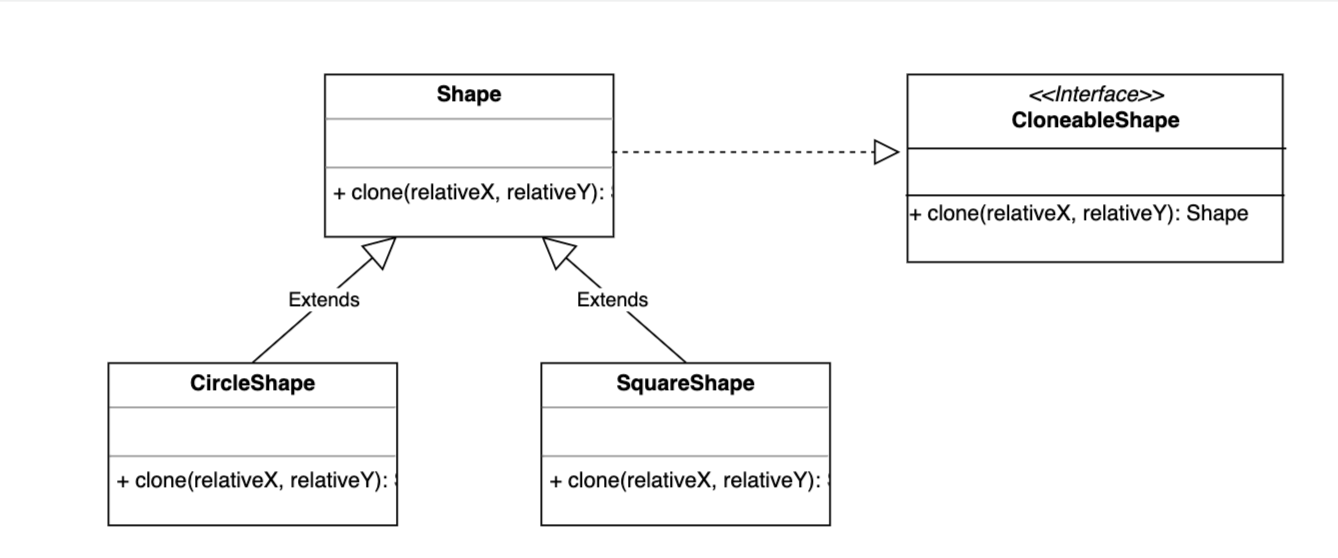


## *Cloning*

The cloning process presents three problematics:

### Cloning the object

To meet this requirement the design pattern Prototype is used. And since the cloning comes with a displacement from the original object (relative X, relative Y), a new interface is created and implemented by the Shape hierarchy to perform both cloning of the object and its displacement.



### ClONING A CLONE

Cleaning a clone shape has to bound the new clone to the original shape. To do so, in the clone method a check is done to see if the original shape is a clone or not:

// Process the observers in relation with the fact that the shape is a clone or not

**if**(!**this**.isClone()) {

clone.relativeX = relativeX;

clone.relativeY = relativeY;

clone.original = **this**;

**this**.addObserver(clone);

**this**.clonesList.add(clone.getId());

} **else** {

Shape origine = **this**.getOriginal();

// set the relative coordinates for clones created from another clones

clone.relativeX = **this**.relativeX + relativeX;

clone.relativeY = **this**.relativeY + relativeY;

clone.original =origine;

origine.addObserver(clone);

origine.clonesList.add(clone.getId());

}

### BOUND the CLONE TO THE ORIGINAL SHAPE

The problematic of sharing the modifications between the clones and the original object is resolved by using the Observer design pattern.

In our case the clone and the original object are instances of the same class. So, the class Shape extends from Observable, and implements Observer.

The origin shape represents the Observable object, after each of its changes that has to be transferred to its clones, two methods are called: setChanges() and notifyObservers().

The clones represent the observers, hence the implementation of the observer interface. While creating a clone, the cloned object is added as an observer to the original shape: **this**.addObserver(clone);

The observer interface contains an update method. This method determines the way the observer behaves when it receives a notification from the original object. In our case, in the update method, the shared attributes are updated with those of the original shape.

When one of the clone’s attributes is modified, the setter of this attribute delegates this modification to the setter of this attribute in the original shape.

**publicvoid** paint(ShapeColor color) {

**if** (!**this**.isClone()) {

**this**.color = color;

setChanged();

notifyObservers();

} **else** {

**this**.getOriginal().paint(color);

}

}

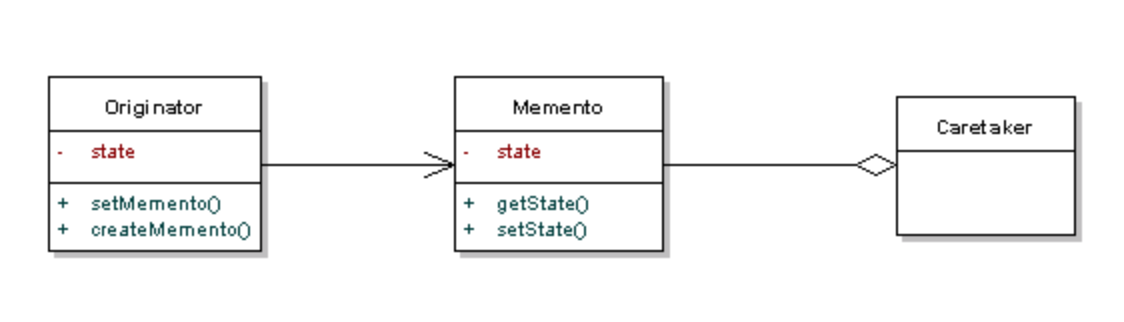
## *UNDO / REDO*

To implement the undo/redo method two design patterns had to be used

### MEMENTO

This design pattern externalizes an object's internal state so that it can be restored later.

This design pattern involves three objects responsibilities

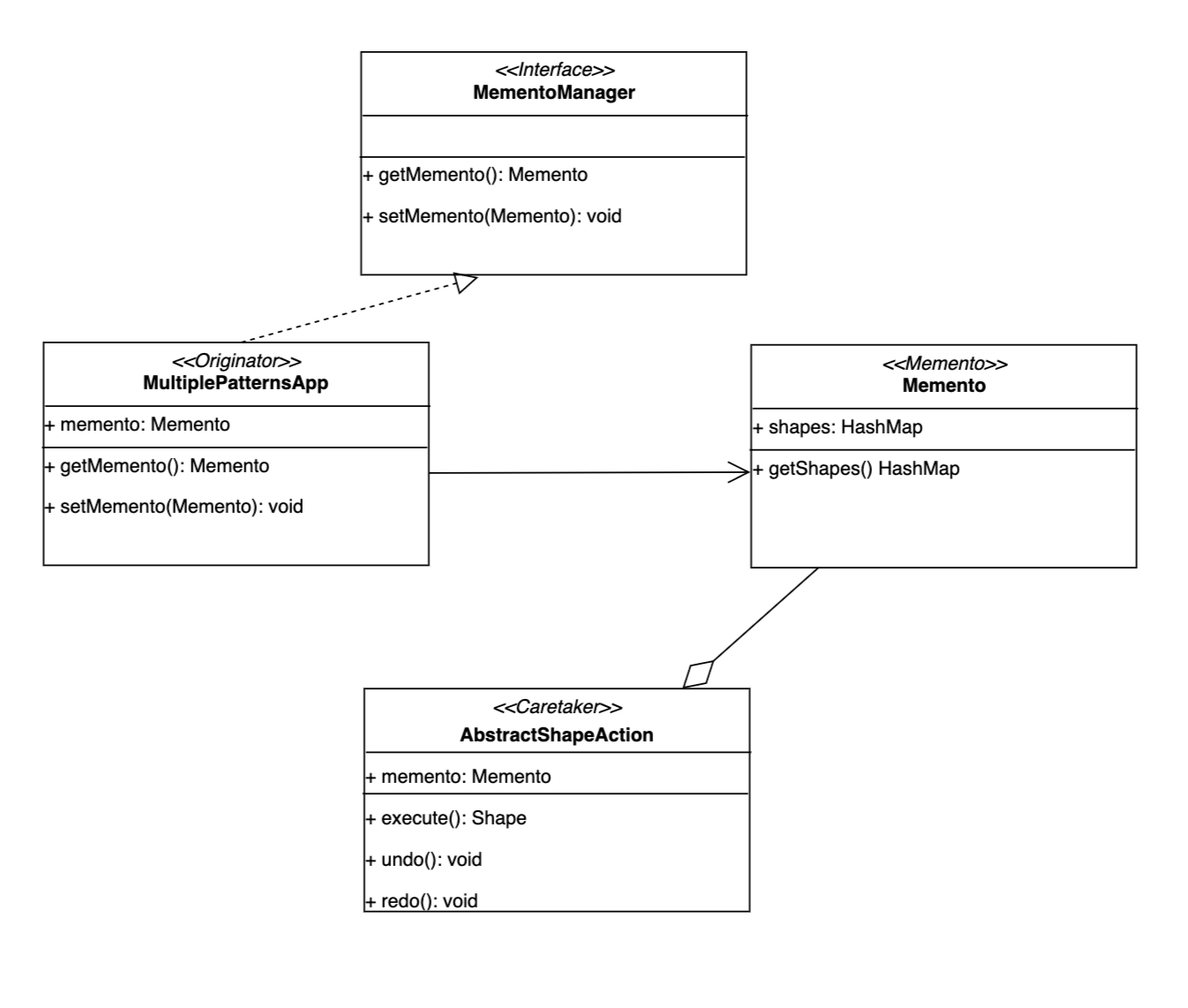


The originator: It is the class which state has to be saved, this class knows how to save itself.

The caretaker: It is the class that deals with when, and why, the Originator needs to save or restore itself.

The memento: It is the class that holds the information about the Originator's state and cannot be modified by the Caretaker.

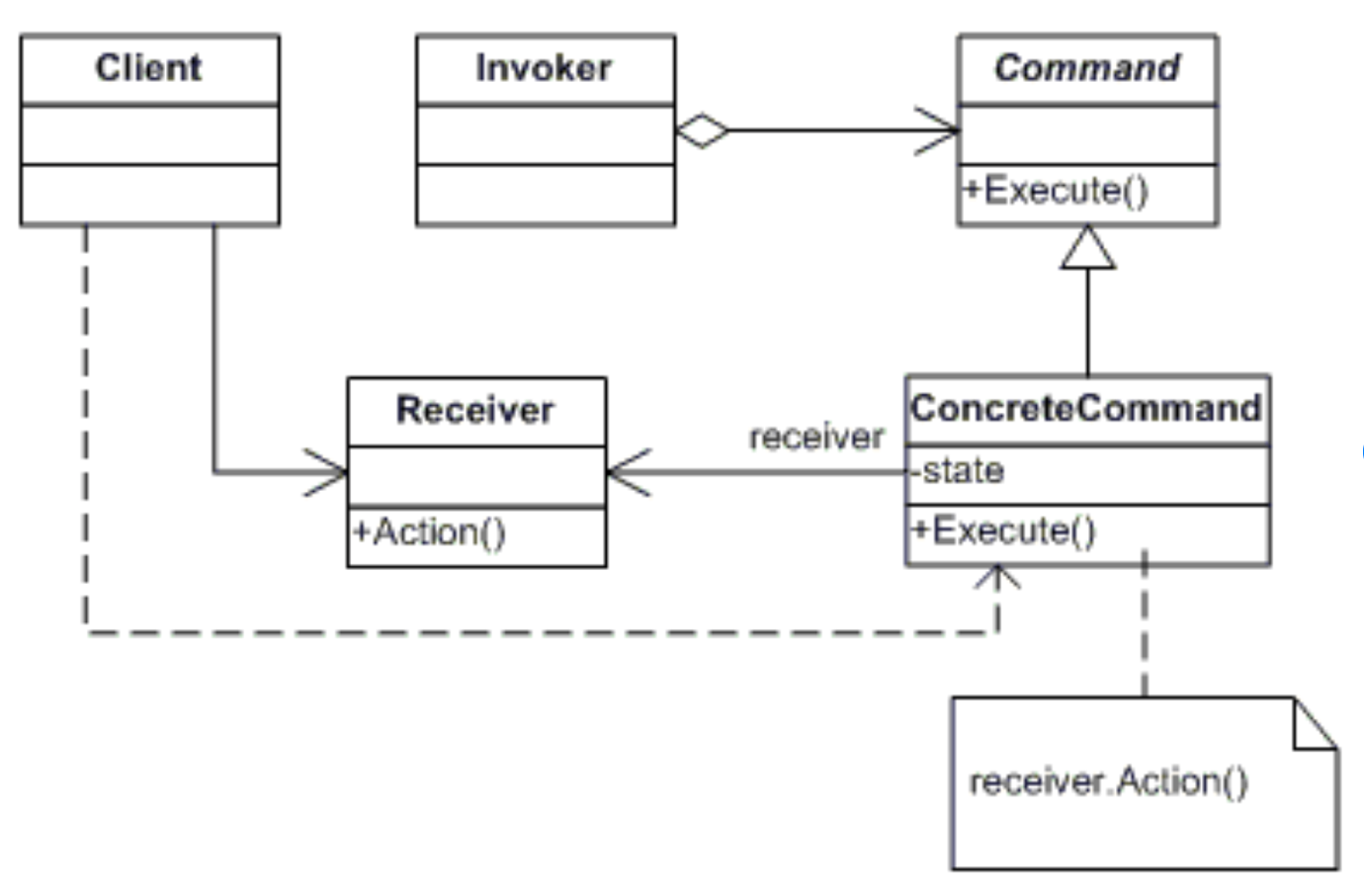
In our case the diagram is as follows:



The MultiplePatternsApp can save its state in the method setMemento, this state is stored in the memento object. The AbstractShapeAction class stores the state of the shapes created before each action in the memento object and can undo or redo it whenever the user requests it.

### COMMAND

This design pattern is used when we want to decouple objects that produce the commands from their consumers. It is composed of four responsibilities:



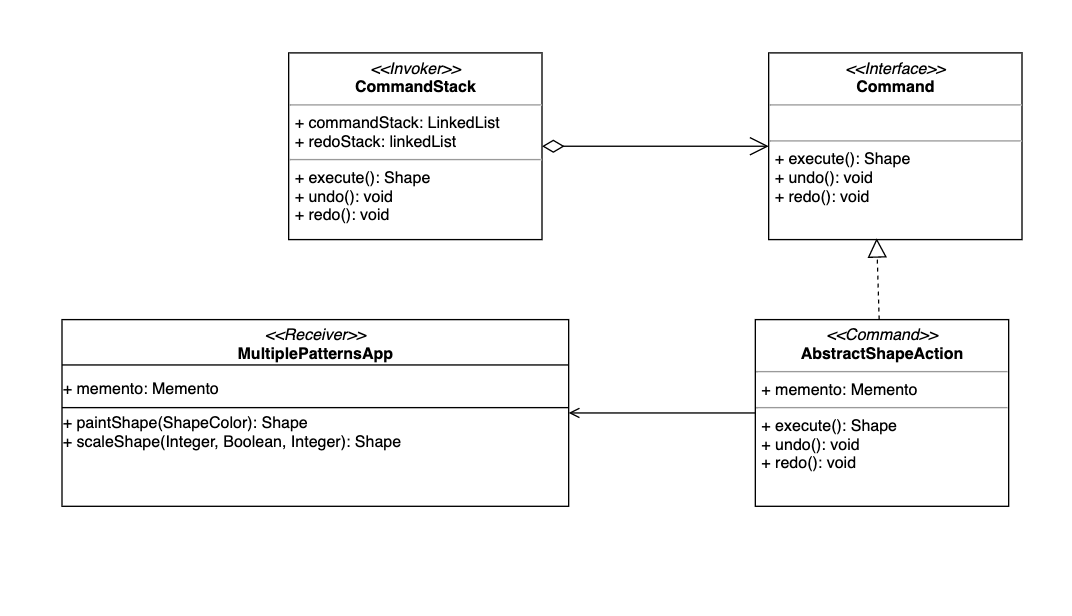
***Command***: Its role is to store all the information required for executing an action.

***Receiver***: Its role is to execute the actions.

***Invoker***: Its role is to call the actions.

***Client***: Its role is to control the command execution process

In this project the hierarchy is as follows:



### COMBINING THE TWO

While an action is invoked by the caretaker, a memento of the list of shapes is stored in the memento before execution and saved in the commandStack. The stack LinkedList used in this case because of the need of pulling and putting the elements in the top of the list. The undo and redo methods manipulate the mementos stored in the commandStack and redoStack.