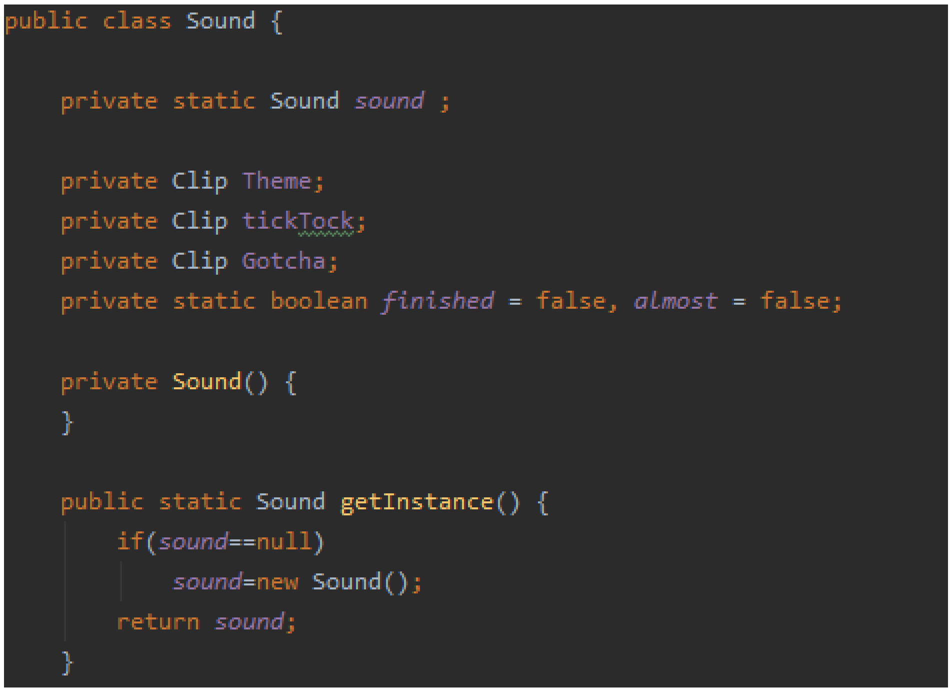
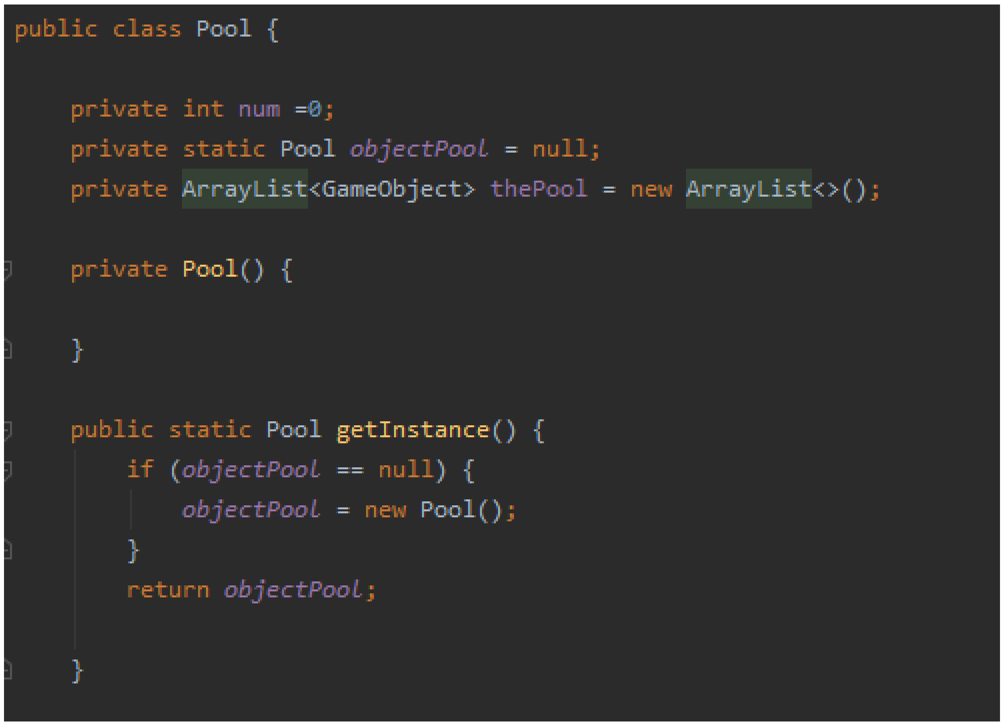
Design Patterns

1- Singleton:

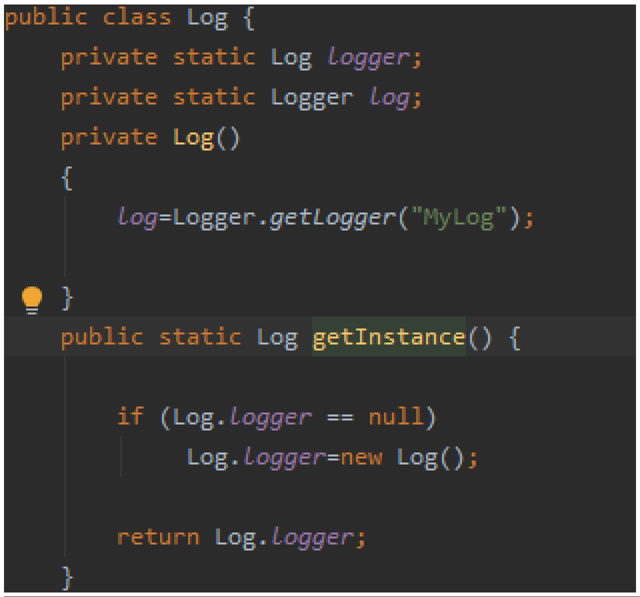
It is used when you need one instance only of this class to be made, so when we ask for an instance we get the same object of this class.



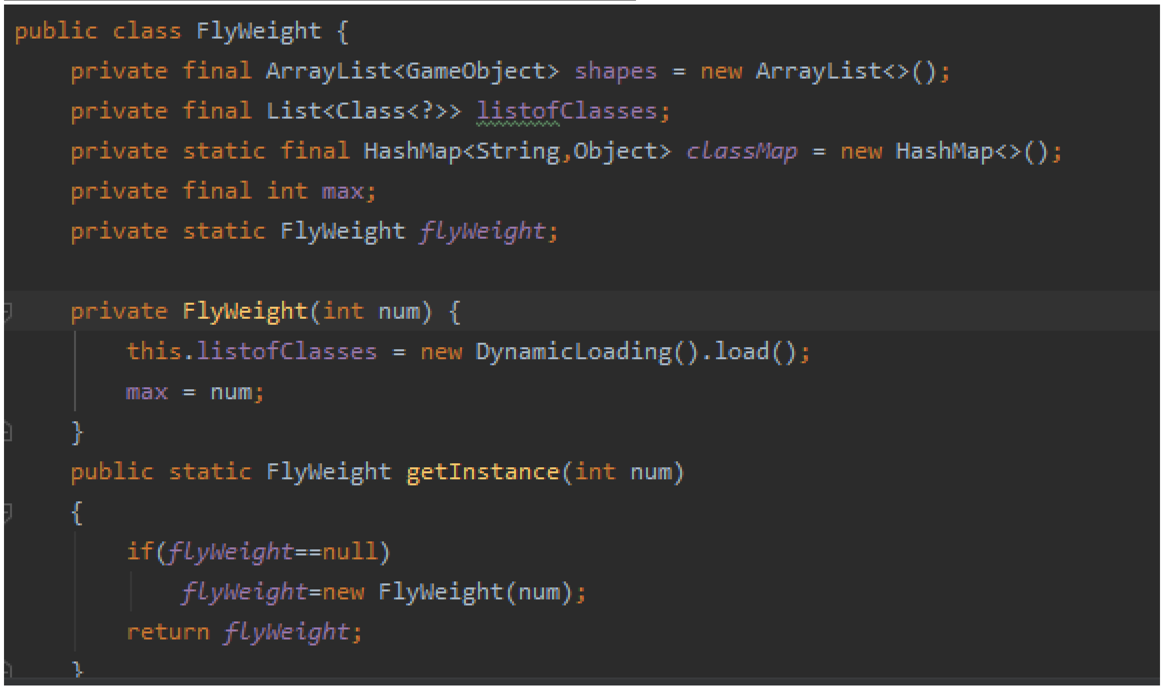
```java
public class Sound {

    private static Sound sound ;

    private Clip Theme;
    private Clip tickTock;
    private Clip Gotcha;
    private static boolean finished = false, almost = false;

    private Sound() {
    }

    public static Sound getInstance() {
        if(sound==null)
            sound=new Sound();
        return sound;
    }
```



```java
public class Pool {

    private int num =0;
    private static Pool objectPool = null;
    private ArrayList<GameObject> thePool = new ArrayList<>();

    private Pool() {

    }

    public static Pool getInstance() {
        if (objectPool == null) {
            objectPool = new Pool();
        }
        return objectPool;

    }
```



```
public class Log {
    private static Log logger;
    private static Logger log;
    private Log()
    {
        log=Logger.getLogger("MyLog");

    }
    public static Log getInstance() {

        if (Log.logger == null)
            Log.logger=new Log();

        return Log.logger;
    }
```

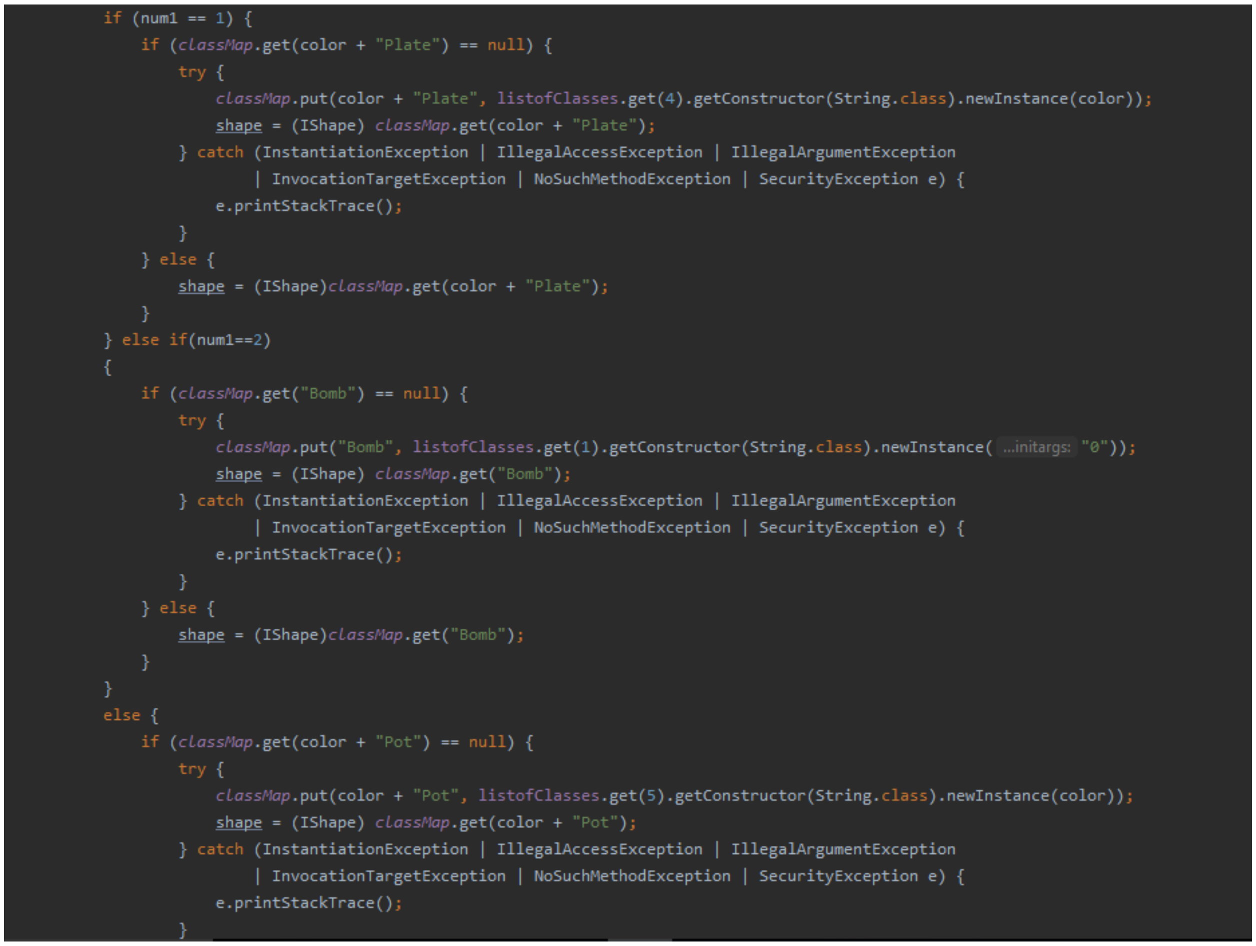


```
public class FlyWeight {
    private final ArrayList<GameObject> shapes = new ArrayList<>();
    private final List<Class<?>> listofClasses;
    private static final HashMap<String,Object> classMap = new HashMap<>();
    private final int max;
    private static FlyWeight flyWeight;

    private FlyWeight(int num) {
        this.listofClasses = new DynamicLoading().load();
        max = num;
    }
    public static FlyWeight getInstance(int num)
    {
        if(flyWeight==null)
            flyWeight=new FlyWeight(num);
        return flyWeight;
    }
```

2- Factory:

It is used when we deal with a reference from the interface and we have many options to instantiate this option to (many classes that implement the same interface).

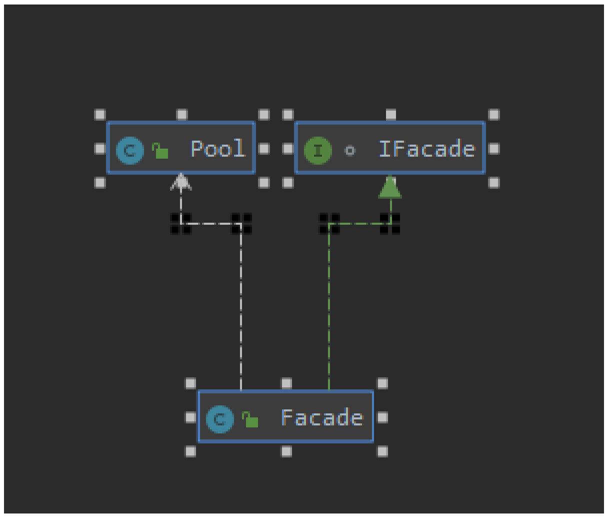
We deal with that interface till a point that we decide what object of this interface we need so we assign it to this reference. Such as in the strategy chooser when the gamer chooses the game mode or the image chooser when the load image process starts.



```java
        if (num1 == 1) {
            if (classMap.get(color + "Plate") == null) {
                try {
                    classMap.put(color + "Plate", listofClasses.get(4).getConstructor(String.class).newInstance(color));
                    shape = (IShape) classMap.get(color + "Plate");
                } catch (InstantiationException | IllegalAccessException | IllegalArgumentException
                        | InvocationTargetException | NoSuchMethodException | SecurityException e) {
                    e.printStackTrace();
                }
            } else {
                shape = (IShape)classMap.get(color + "Plate");
            }
        } else if(num1==2)
        {
            if (classMap.get("Bomb") == null) {
                try {
                    classMap.put("Bomb", listofClasses.get(1).getConstructor(String.class).newInstance( ...initargs: "0"));
                    shape = (IShape) classMap.get("Bomb");
                } catch (InstantiationException | IllegalAccessException | IllegalArgumentException
                        | InvocationTargetException | NoSuchMethodException | SecurityException e) {
                    e.printStackTrace();
                }
            } else {
                shape = (IShape)classMap.get("Bomb");
            }
        }
        else {
            if (classMap.get(color + "Pot") == null) {
                try {
                    classMap.put(color + "Pot", listofClasses.get(5).getConstructor(String.class).newInstance(color));
                    shape = (IShape) classMap.get(color + "Pot");
                } catch (InstantiationException | IllegalAccessException | IllegalArgumentException
                        | InvocationTargetException | NoSuchMethodException | SecurityException e) {
                    e.printStackTrace();
                }
```

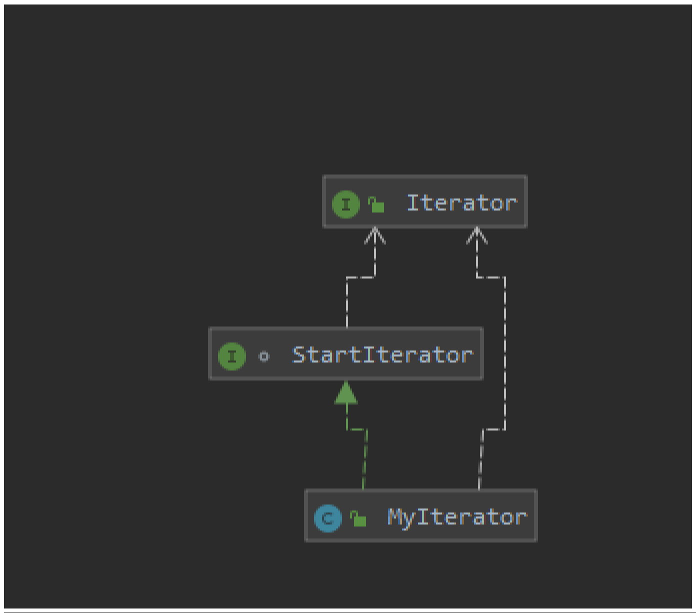
3- Pool:

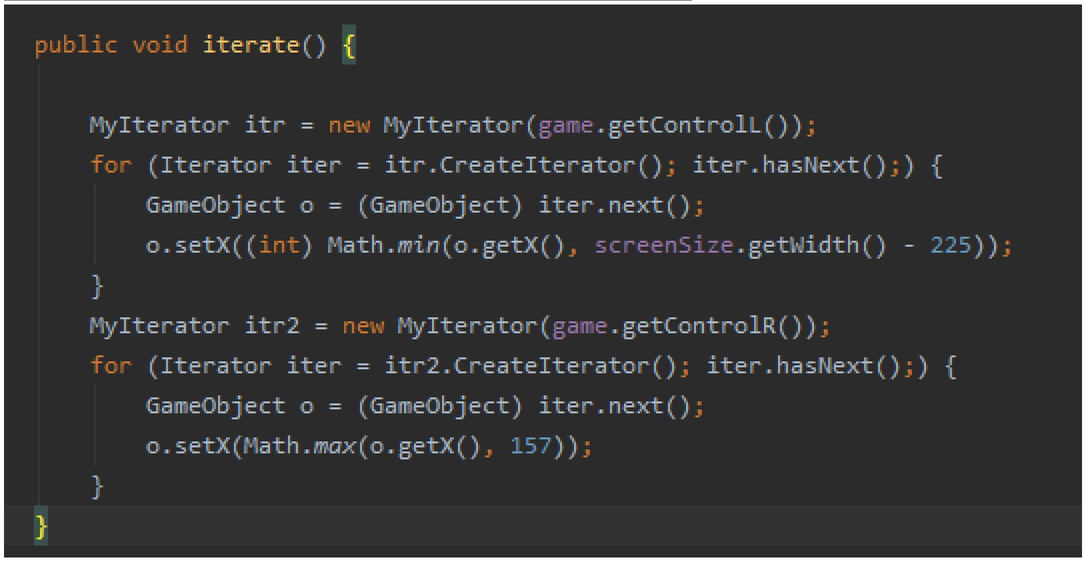
Its purpose is to limit the use of the memory and don't waste it, so when we use an object we remove it from the pool as it's no longer available for the others to use it, when we finish using the object we return it back to the pool. The pool here is the moving objects, when the gamer collects three consecutive shapes with the same color or drop one, it will be returned to the pool so it can be reused.



4- Iterator:

It is used to iterate on any 'Array List' used in the project



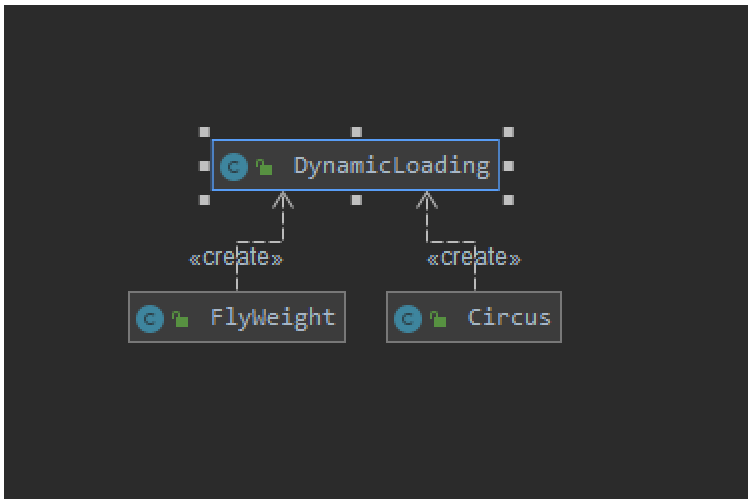


```
public void iterate() {

    MyIterator itr = new MyIterator(game.getControlL());
    for (Iterator iter = itr.CreateIterator(); iter.hasNext();) {
        GameObject o = (GameObject) iter.next();
        o.setX((int) Math.min(o.getX(), screenSize.getWidth() - 225));
    }
    MyIterator itr2 = new MyIterator(game.getControlR());
    for (Iterator iter = itr2.CreateIterator(); iter.hasNext();) {
        GameObject o = (GameObject) iter.next();
        o.setX(Math.max(o.getX(), 157));
    }
}
```

5- Dynamic Linkage:

- Load all image objects of the game dynamically before the game begins

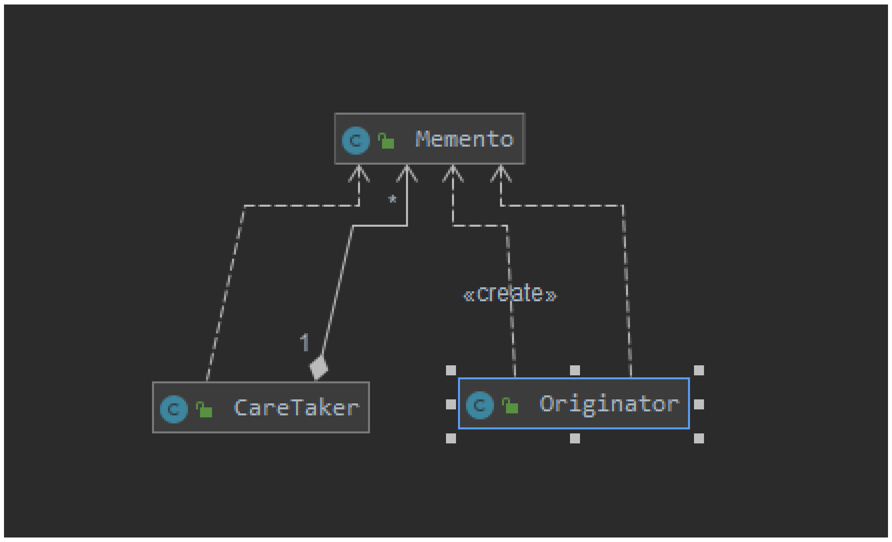
- Load all image from resources from jar file



6- Snapshot:

- Save the state of both the left and right stack with every insertion, and the last inserted shape in the stack

- Remove the last three states if there are three consecutive shapes have the same color

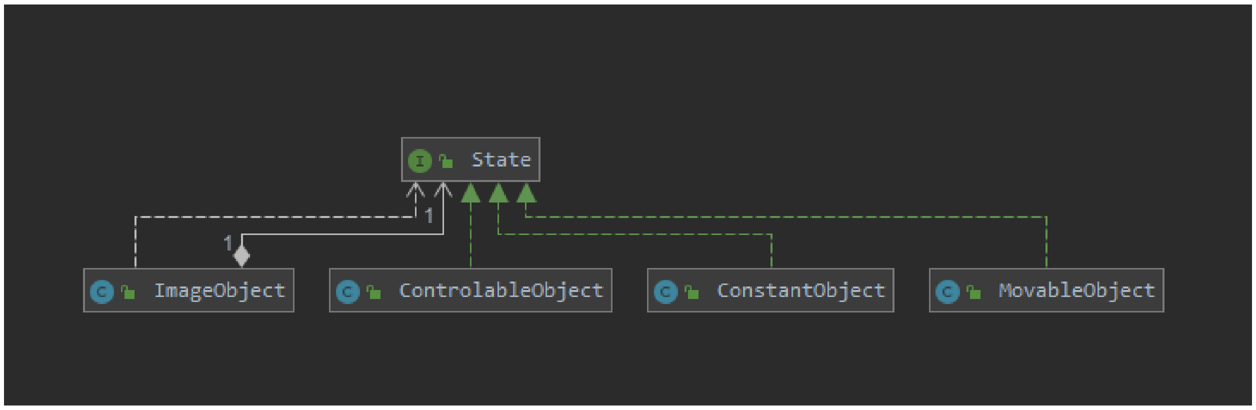


7- State:

It is used to determine the state of the image: constant, moving or controlled. It is also used to change the shape from state to another:
- when the gamer captures it: from moving to controlled

- when he collects 3 with the same color: from controlled to moving

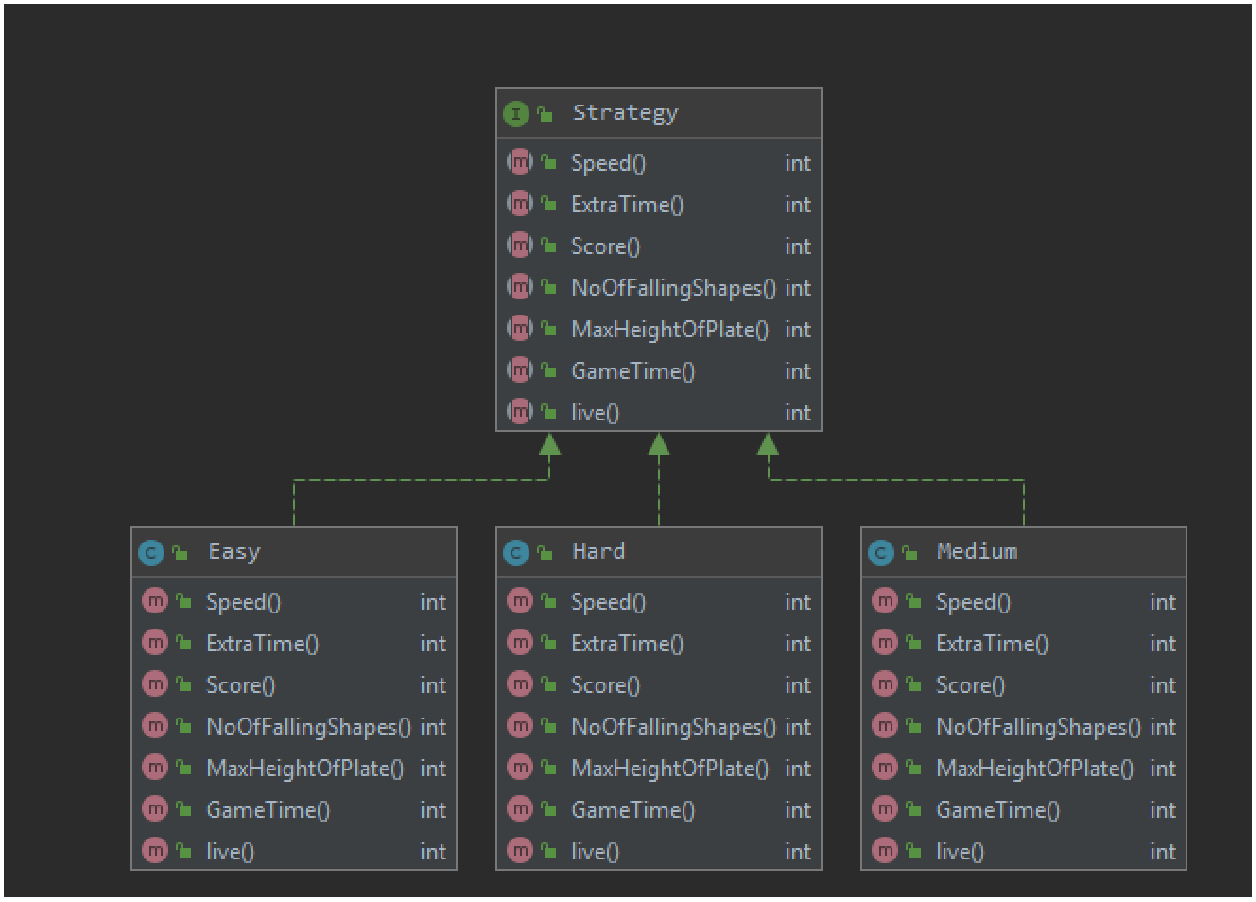
- when the shape falls: from controlled to moving



8- Strategy:

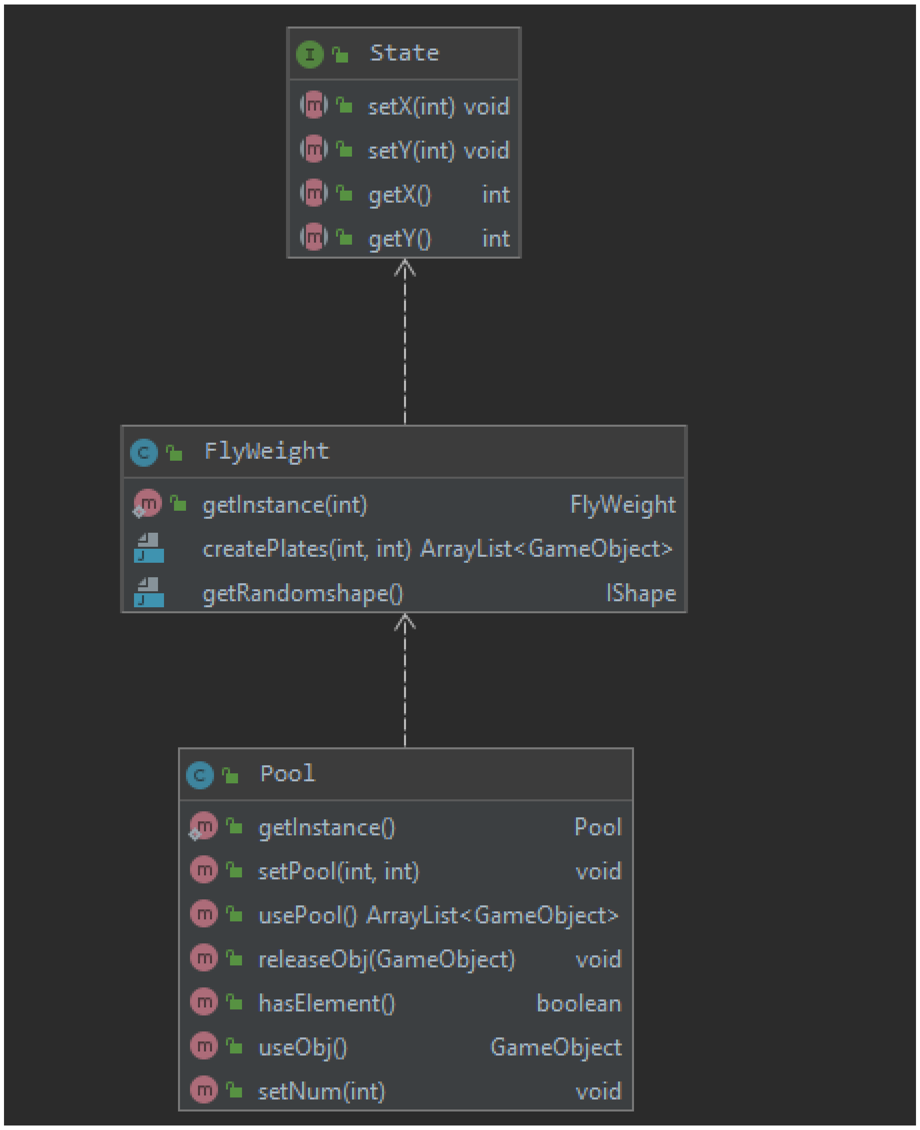
It determines for each level its properties:

The game speed, the control speed, the maximum time, the number of falling shapes, the score, the bonus time, the maximum height the plates can reach before the gamer loses



9- Flyweight:

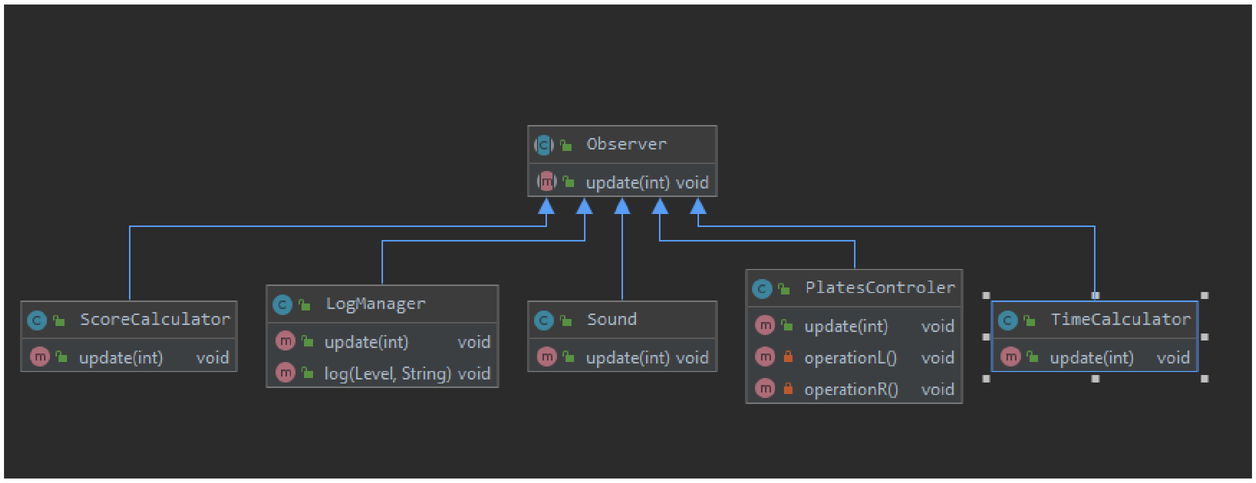
It randomizes the falling shapes' colors, and it is also used to reduce the number of loaded falling objects at the beginning of the game.



10- Observer:

- Used to notify time, score, sound and removing plates from the stack

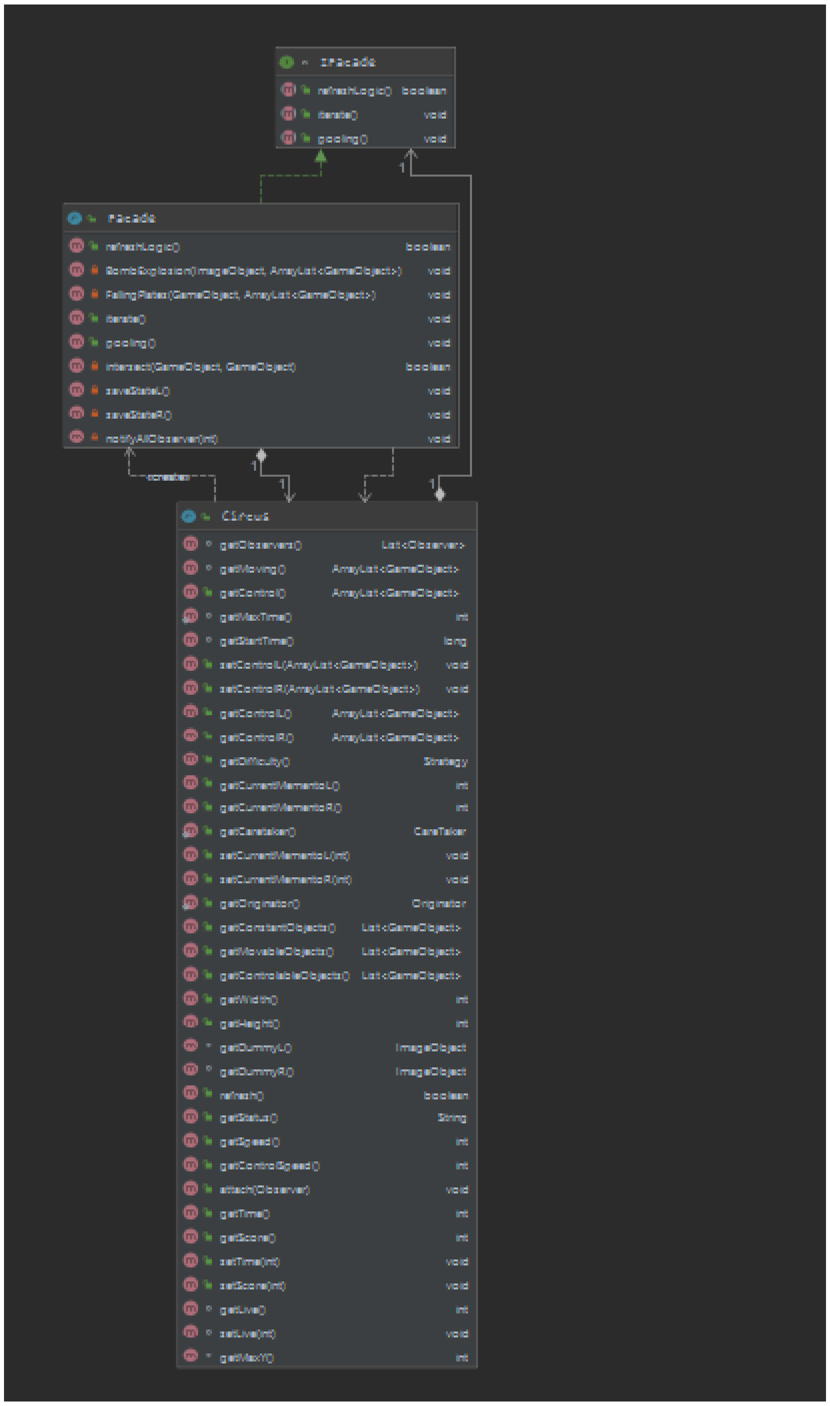
- This notify occurrence when the gamer collect three consecutive shapes with the same color



11- Facade:

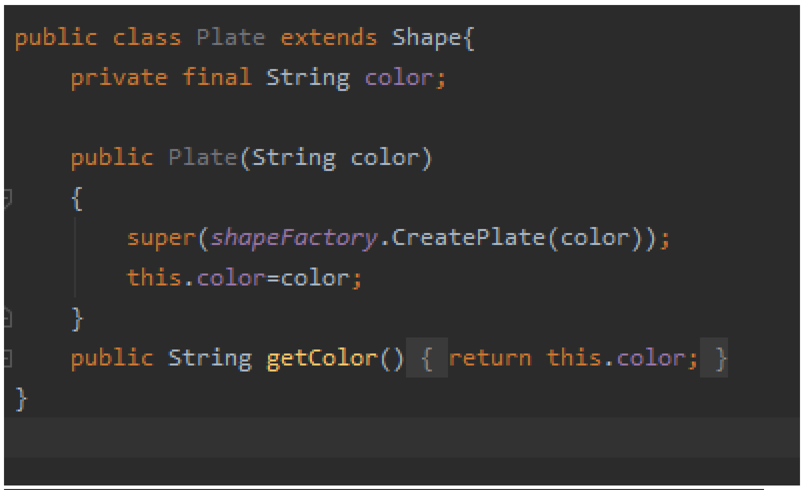
- All the game is connected in this class through the refresh method of the world interface

- Handling creating plates when taken

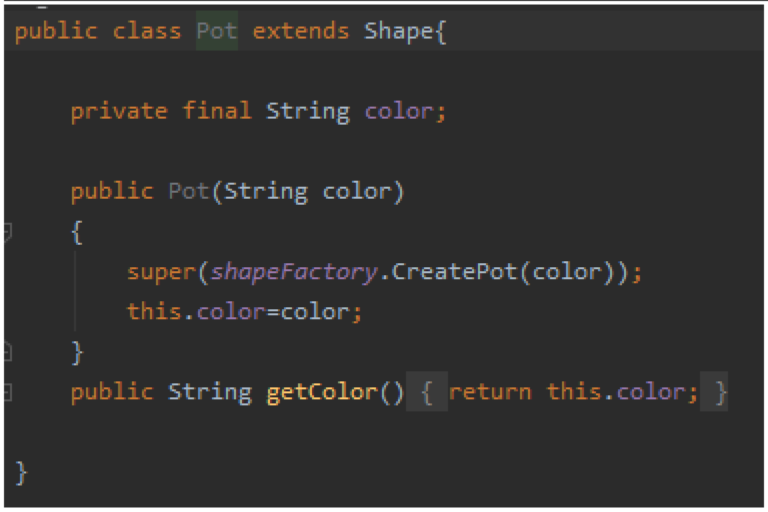

IFacade
refreshLogic() boolean
iterate() void
pooling() void
1
Facade
refreshLogic() boolean
BombExplosion(ImageObject, ArrayList<GameObject>) void
FallingPlates(GameObject, ArrayList<GameObject>) void
iterate() void
pooling() void
intersect(GameObject, GameObject) boolean
saveStateL() void
saveStateR() void
notifyAllObserver(int) void
«create»
1
1
1
Circus
getObservers() List<Observer>
getMoving() ArrayList<GameObject>
getControl() ArrayList<GameObject>
getMaxTime() int
getStartTime() long
setControlL(ArrayList<GameObject>) void
setControlR(ArrayList<GameObject>) void
getControlL() ArrayList<GameObject>
getControlR() ArrayList<GameObject>
getDifficulty() Strategy
getCurrentMementoL() int
getCurrentMementoR() int
getCaretaker() CareTaker
setCurrentMementoL(int) void
setCurrentMementoR(int) void
getOriginator() Originator
getConstantObjects() List<GameObject>
getMovableObjects() List<GameObject>
getControlableObjects() List<GameObject>
getWidth() int
getHeight() int
getDummyL() ImageObject
getDummyR() ImageObject
refresh() boolean
getStatus() String
getSpeed() int
getControlSpeed() int
attach(Observer) void
getTime() int
getScore() int
setTime(int) void
setScore(int) void
getLive() int
setLive(int) void
getMaxY() int

12- Decorator:

To make colored shapes.



```java
public class Plate extends Shape{
    private final String color;

    public Plate(String color)
    {
        super(shapeFactory.CreatePlate(color));
        this.color=color;
    }
    public String getColor() { return this.color; }
}
```



```java
public class Pot extends Shape{

    private final String color;

    public Pot(String color)
    {
        super(shapeFactory.CreatePot(color));
        this.color=color;
    }
    public String getColor() { return this.color; }

}
```

13- MVC:

We applied MVC in our design.

14- Command:

- It is used to create the main menu to choose the desired difficulty

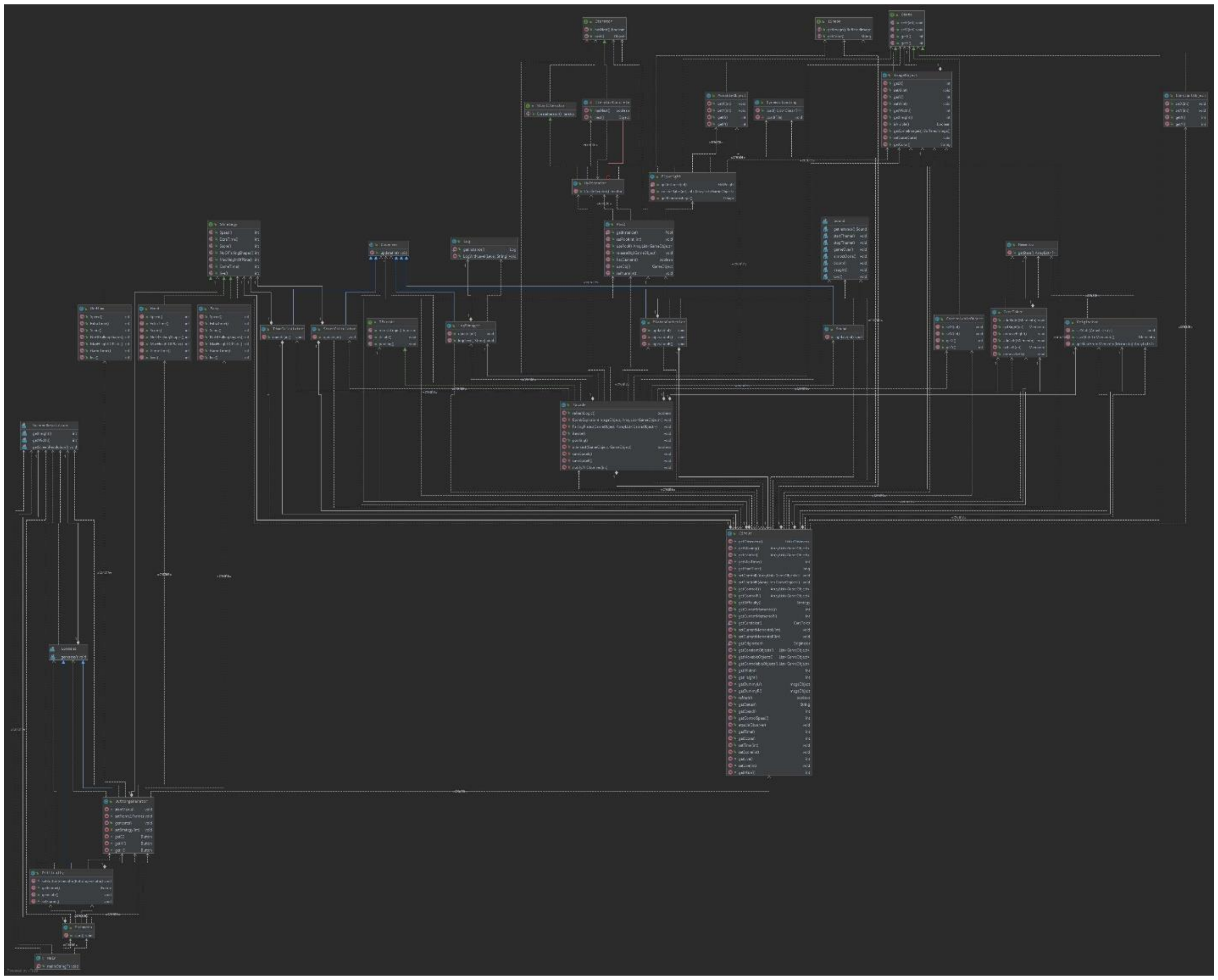
- The main class creates an instance of Screen Resolution to get it then creates an instance of Main Menu. Then the call methods instantiate 3 classes that extend the Command Abstract Class (abstract to fill the constructor)

- The 2 subclasses are:

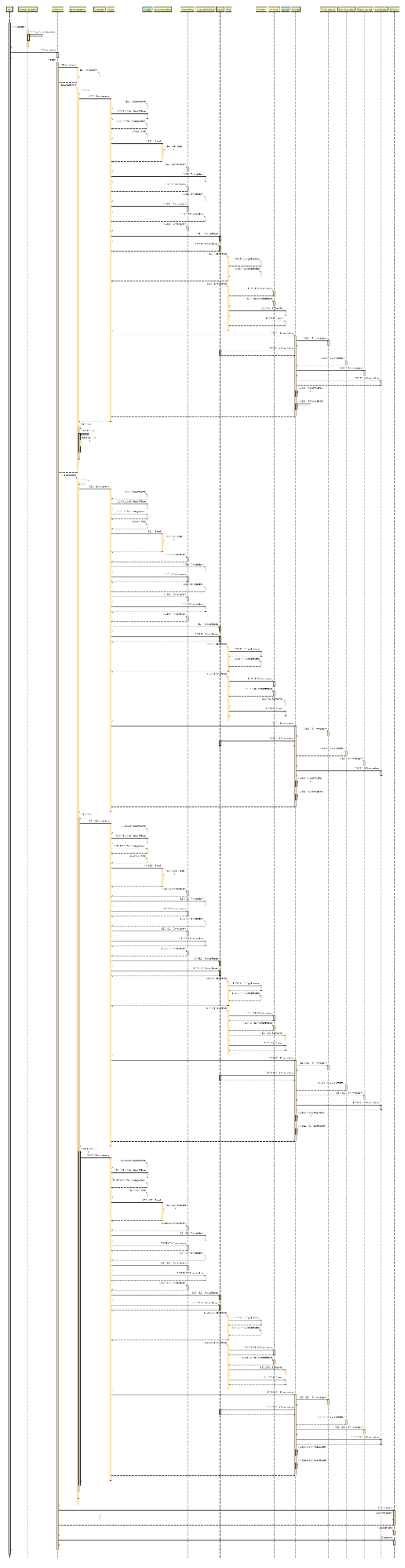
Button generator: creates the buttons holding the difficulties

Game Difficulty Box: which is basically the Main Menu

The game UML diagram:



Sequence diagram:



Code design:

We have made a Circus class that implements the world interface, this class creates an interface which is used to access all other design patterns naming a few: Pool, FlyWeight…. The strategy design pattern (which includes the difficulties) contains game time, score and speed of falling objects and their numbers.

We used the “Dynamic Linkage” in order to load the photos (plates, pots and bombs).

The music starts and the created objects are all inside the pool . When refreshing Snapshot design pattern was used to check the last 3 plates and if they have the same color they are sent to the pool and the observers are called : updating score, sound. The state design pattern decides whether this object is controllable like clown or caught plates , or constant like background or movable like the falling shapes.

Design decisions:

<u>Catching the shapes:</u>

The clown initially has right and left plates from which the user can decide the area of intersection so he can adjust the clown to catch the shapes, the falling shape center must be inside this interval ,, but as the clown catching the shapes things won't be straight some shapes will go little bit left or right but still inside this interval, so another condition is added for the catch process to happen, besides being in the intersection of the first plate, the falling shape has to intersect with the last plate too .
This leads us to the last possibility, that the falling shape intersect with the last shape but at the same time it doesn't intersect with the intersection area of the first 2 plates at this point the falling shape doesn't intersect but it causes the last shape to fall with it , the fallen plate will be returned to the moving shapes again but the user loses a point from this .
This last situation happens when the plates go in the same direction right of a shape that was right or left of a shape that was left so that the user has to try to center all the shapes.
The user starts with a limit of falling shapes decided by the mode, when the user catches a shape its reduced from the limit of shapes available when the user collects any 3 successive shapes of the same color he gains score **and the 3 shapes are added again to the pool of shapes available** ,,,
The user has another **feature** to drop the shapes from the clown.