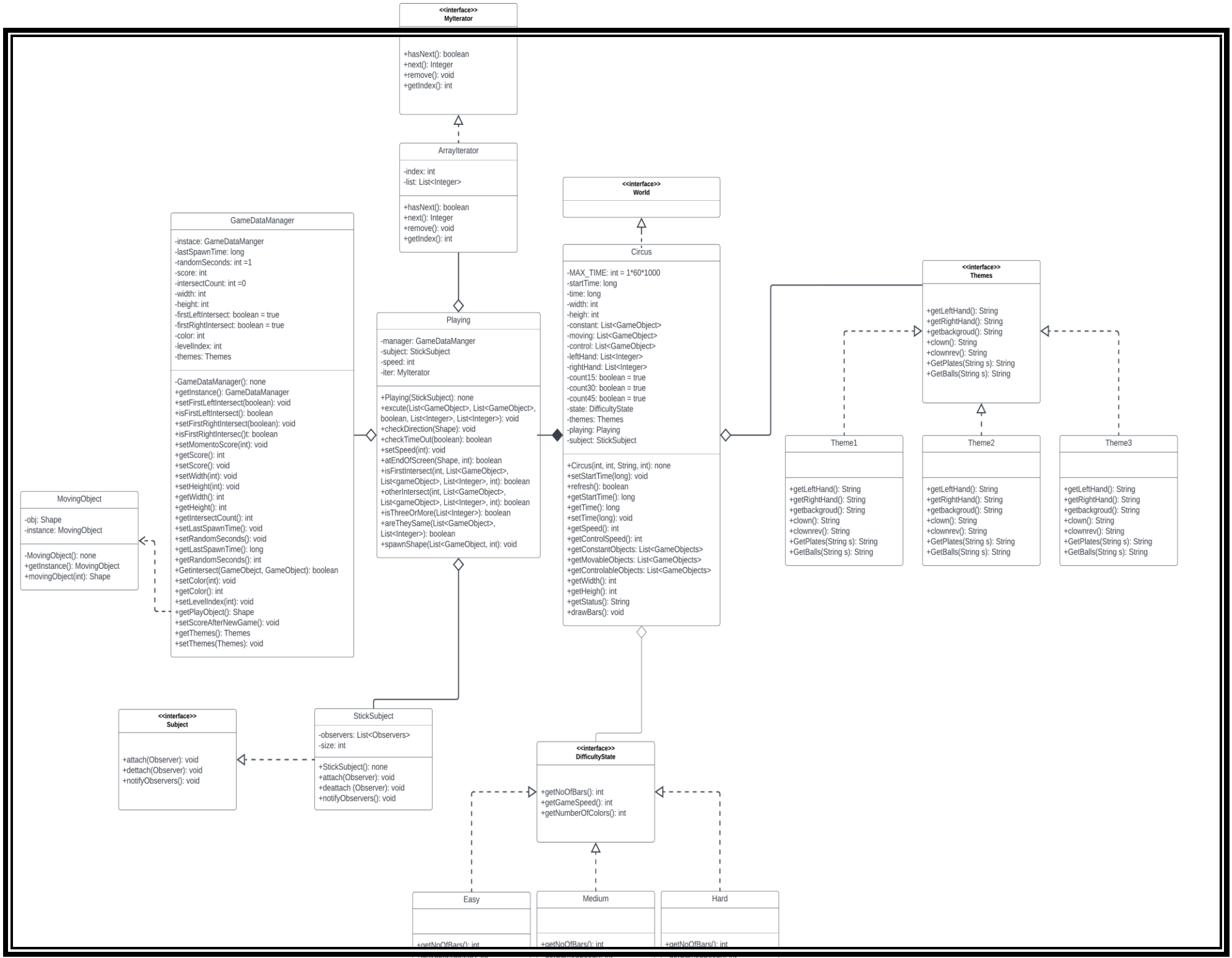
Circus Of Plates

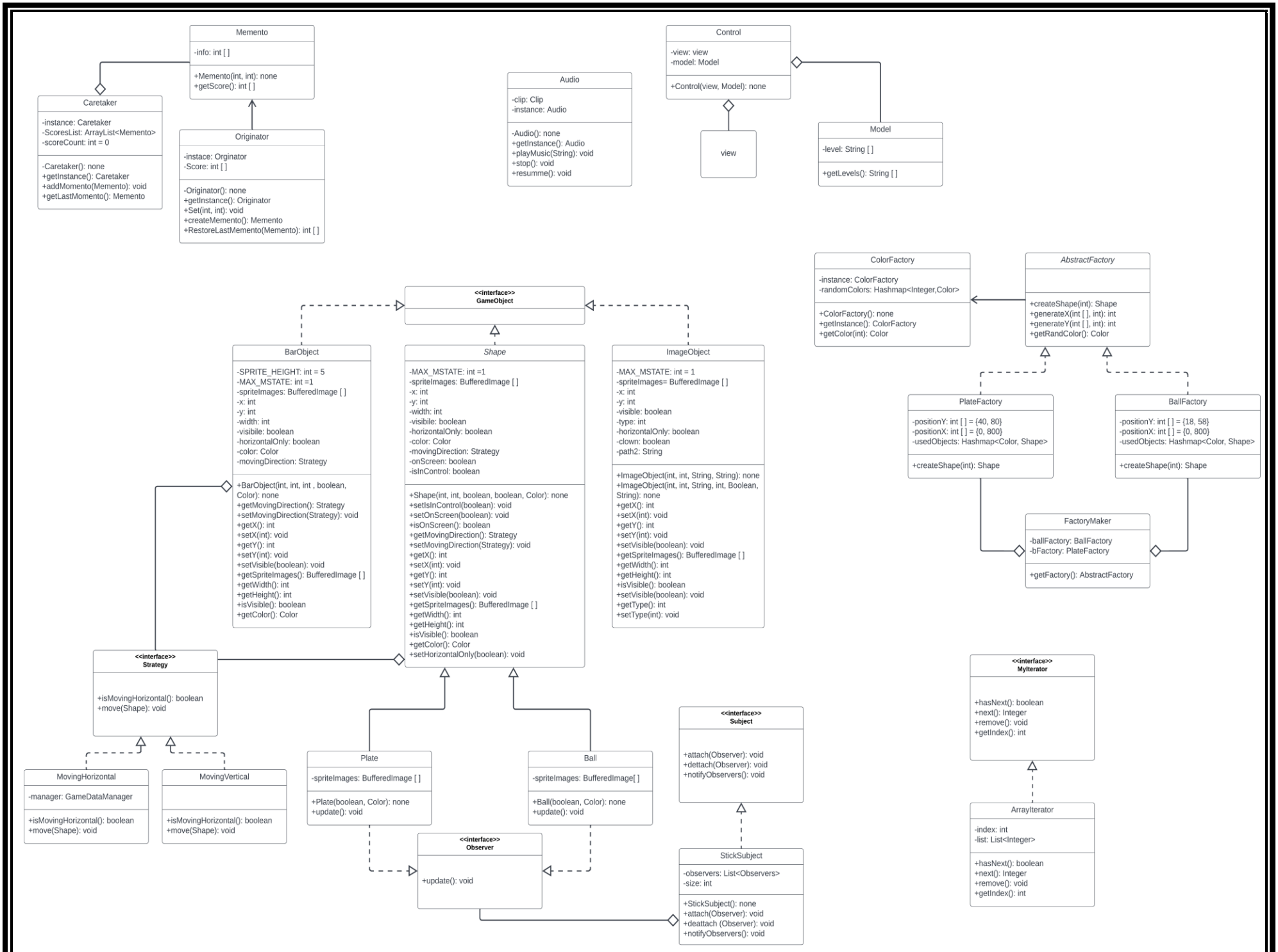
Names: *Youssef Mohamed El-Raggal* *7806*

Ahmed Tarek Ahmed *6646*

Mahmoud Haytham Mahmoud *7560*

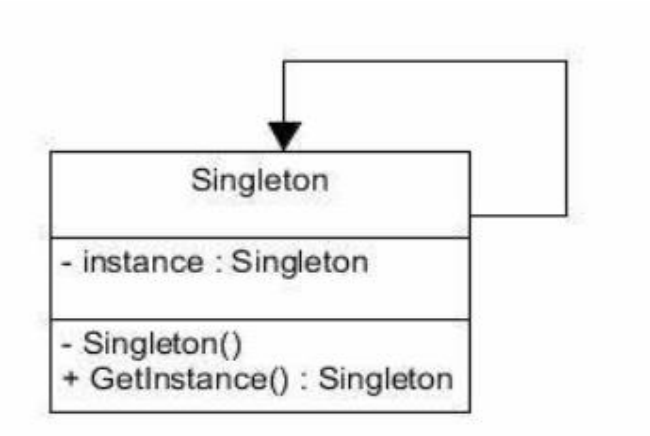
Mohamed Osama Fathy *7861*


<<interface>>
MyIterator
+hasNext(): boolean
+next(): Integer
+remove(): void
+getIndex(): int
ArrayIterator
-index: int
-list: List<Integer>
+hasNext(): boolean
+next(): Integer
+remove(): void
+getIndex(): int
<<interface>>
World
Circus
-MAX_TIME: int = 1*60*1000
-startTime: long
-time: long
-width: int
-heigh: int
-constant: List<GameObject>
-moving: List<GameObject>
-control: List<GameObject>
-leftHand: List<Integer>
-rightHand: List<Integer>
-count15: boolean = true
-count30: boolean = true
-count45: boolean = true
-state: DifficultyState
-themes: Themes
-playing: Playing
-subject: StickSubject
+Circus(int, int, String, int): none
+setStartTime(long): void
+refresh(): boolean
+getStartTime(): long
+getTime(): long
+setTime(long): void
+getSpeed(): int
+getControlSpeed(): int
+getConstantObjects: List<GameObjects>
+getMovableObjects: List<GameObjects>
+getControlableObjects: List<GameObjects>
+getWidth(): int
+getHeigh(): int
+getStatus(): String
+drawBars(): void
<<interface>>
Themes
+getLeftHand(): String
+getRightHand(): String
+getbackgroud(): String
+clown(): String
+clownrev(): String
+GetPlates(String s): String
+GetBalls(String s): String
Theme1
+getLeftHand(): String
+getRightHand(): String
+getbackgroud(): String
+clown(): String
+clownrev(): String
+GetPlates(String s): String
+GetBalls(String s): String
Theme2
+getLeftHand(): String
+getRightHand(): String
+getbackgroud(): String
+clown(): String
+clownrev(): String
+GetPlates(String s): String
+GetBalls(String s): String
Theme3
+getLeftHand(): String
+getRightHand(): String
+getbackgroud(): String
+clown(): String
+clownrev(): String
+GetPlates(String s): String
+GetBalls(String s): String
GameDataManager
-instace: GameDataManger
-lastSpawnTime: long
-randomSeconds: int =1
-score: int
-intersectCount: int =0
-width: int
-height: int
-firstLeftIntersect: boolean = true
-firstRightIntersect: boolean = true
-color: int
-levelIndex: int
-themes: Themes
-GameDataManager(): none
+getInstance(): GameDataManager
+setFirstLeftIntersect(boolean): void
+isFirstLeftIntersect(): boolean
+setFirstRightIntersect(boolean): void
+isFirstRightIntersec()t: boolean
+setMomentoScore(int): void
+getScore(): int
+setScore(): void
+setWidth(int): void
+setHeight(int): void
+getWidth(): int
+getHeight(): int
+getIntersectCount(): int
+setLastSpawnTime(): void
+setRandomSeconds(): void
+getLastSpawnTime(): long
+getRandomSeconds(): int
+Getintersect(GameObejct, GameObject): boolean
+setColor(int): void
+getColor(): int
+setLevelIndex(int): void
+getPlayObject(): Shape
+setScoreAfterNewGame(): void
+getThemes(): Themes
+setThemes(Themes): void
Playing
-manager: GameDataManger
-subject: StickSubject
-speed: int
-iter: MyIterator
+Playing(StickSubject): none
+excute(List<GameObject>, List<GameObject>, boolean, List<Integer>, List<Integer>): void
+checkDirection(Shape): void
+checkTimeOut(boolean): boolean
+setSpeed(int): void
+atEndOfScreen(Shape, int): boolean
+isFirstIntersect(int, List<GameObject>, List<gameObject>, List<Integer>, int): boolean
+otherIntersect(int, List<GameObject>, List<gameObject>, List<Integer>, int): boolean
+isThreeOrMore(List<Integer>): boolean
+areTheySame(List<GameObject>, List<Integer>): boolean
+spawnShape(List<GameObject, int): void
MovingObject
-obj: Shape
-instance: MovingObject
-MovingObject(): none
+getInstance(): MovingObject
+movingObject(int): Shape
<<interface>>
Subject
+attach(Observer): void
+dettach(Observer): void
+notifyObservers(): void
StickSubject
-observers: List<Observers>
-size: int
+StickSubject(): none
+attach(Observer): void
+deattach (Observer): void
+notifyObservers(): void
<<interface>>
DifficultyState
+getNoOfBars(): int
+getGameSpeed(): int
+getNumberOfColors(): int
Easy
+getNoOfBars(): int
Medium
+getNoOfBars(): int
Hard
+getNoOfBars(): int


Memento
-info: int [ ]
+Memento(int, int): none
+getScore(): int [ ]
Caretaker
-instance: Caretaker
-ScoresList: ArrayList<Memento>
-scoreCount: int = 0
-Caretaker(): none
+getInstance(): Caretaker
+addMomento(Memento): void
+getLastMomento(): Memento
Originator
-instace: Orginator
-Score: int [ ]
-Originator(): none
+getInstance(): Originator
+Set(int, int): void
+createMemento(): Memento
+RestoreLastMemento(Memento): int [ ]
Audio
-clip: Clip
-instance: Audio
-Audio(): none
+getInstance(): Audio
+playMusic(String): void
+stop(): void
+resumme(): void
Control
-view: view
-model: Model
+Control(view, Model): none
view
Model
-level: String [ ]
+getLevels(): String [ ]
ColorFactory
-instance: ColorFactory
-randomColors: Hashmap<Integer,Color>
+ColorFactory(): none
+getInstance(): ColorFactory
+getColor(int): Color
AbstractFactory
+createShape(int): Shape
+generateX(int [ ], int): int
+generateY(int [ ], int): int
+getRandColor(): Color
PlateFactory
-positionY: int [ ] = {40, 80}
-positionX: int [ ] = {0, 800}
-usedObjects: Hashmap<Color, Shape>
+createShape(int): Shape
BallFactory
-positionY: int [ ] = {18, 58}
-positionX: int [ ] = {0, 800}
-usedObjects: Hashmap<Color, Shape>
+createShape(int): Shape
FactoryMaker
-ballFactory: BallFactory
-bFactory: PlateFactory
+getFactory(): AbstractFactory
<<interface>>
GameObject
BarObject
-SPRITE_HEIGHT: int = 5
-MAX_MSTATE: int =1
-spriteImages: BufferedImage [ ]
-x: int
-y: int
-width: int
-visibile: boolean
-horizontalOnly: boolean
-color: Color
-movingDirection: Strategy
+BarObject(int, int, int , boolean, Color): none
+getMovingDirection(): Strategy
+setMovingDirection(Strategy): void
+getX(): int
+setX(int): void
+getY(): int
+setY(int): void
+setVisible(boolean): void
+getSpriteImages(): BufferedImage [ ]
+getWidth(): int
+getHeight(): int
+isVisible(): boolean
+getColor(): Color
Shape
-MAX_MSTATE: int =1
-spriteImages: BufferedImage [ ]
-x: int
-y: int
-width: int
-visibile: boolean
-horizontalOnly: boolean
-color: Color
-movingDirection: Strategy
-onScreen: boolean
-isInControl: boolean
+Shape(int, int, boolean, boolean, Color): none
+setIsInControl(boolean): void
+setOnScreen(boolean): void
+isOnScreen(): boolean
+getMovingDirection(): Strategy
+setMovingDirection(Strategy): void
+getX(): int
+setX(int): void
+getY(): int
+setY(int): void
+setVisible(boolean): void
+getSpriteImages(): BufferedImage [ ]
+getWidth(): int
+getHeight(): int
+isVisible(): boolean
+getColor(): Color
+setHorizontalOnly(boolean): void
ImageObject
-MAX_MSTATE: int = 1
-spriteImages= BufferedImage [ ]
-x: int
-y: int
-visible: boolean
-type: int
-horizontalOnly: boolean
-clown: boolean
-path2: String
+ImageObject(int, int, String, String): none
+ImageObject(int, int, String, int, Boolean, String): none
+getX(): int
+setX(int): void
+getY(): int
+setY(int): void
+setVisible(boolean): void
+getSpriteImages(): BufferedImage [ ]
+getWidth(): int
+getHeight(): int
+isVisible(): boolean
+setVisible(boolean): void
+getType(): int
+setType(int): void
<<interface>>
Strategy
+isMovingHorizontal(): boolean
+move(Shape): void
MovingHorizontal
-manager: GameDataManager
+isMovingHorizontal(): boolean
+move(Shape): void
MovingVertical
+isMovingHorizontal(): boolean
+move(Shape): void
Plate
-spriteImages: BufferedImage [ ]
+Plate(boolean, Color): none
+update(): void
Ball
-spriteImages: BufferedImage[ ]
+Ball(boolean, Color): none
+update(): void
<<interface>>
Observer
+update(): void
<<interface>>
Subject
+attach(Observer): void
+dettach(Observer): void
+notifyObservers(): void
StickSubject
-observers: List<Observers>
-size: int
+StickSubject(): none
+attach(Observer): void
+deattach (Observer): void
+notifyObservers(): void
<<interface>>
MyIterator
+hasNext(): boolean
+next(): Integer
+remove(): void
+getIndex(): int
ArrayIterator
-index: int
-list: List<Integer>
+hasNext(): boolean
+next(): Integer
+remove(): void
+getIndex(): int

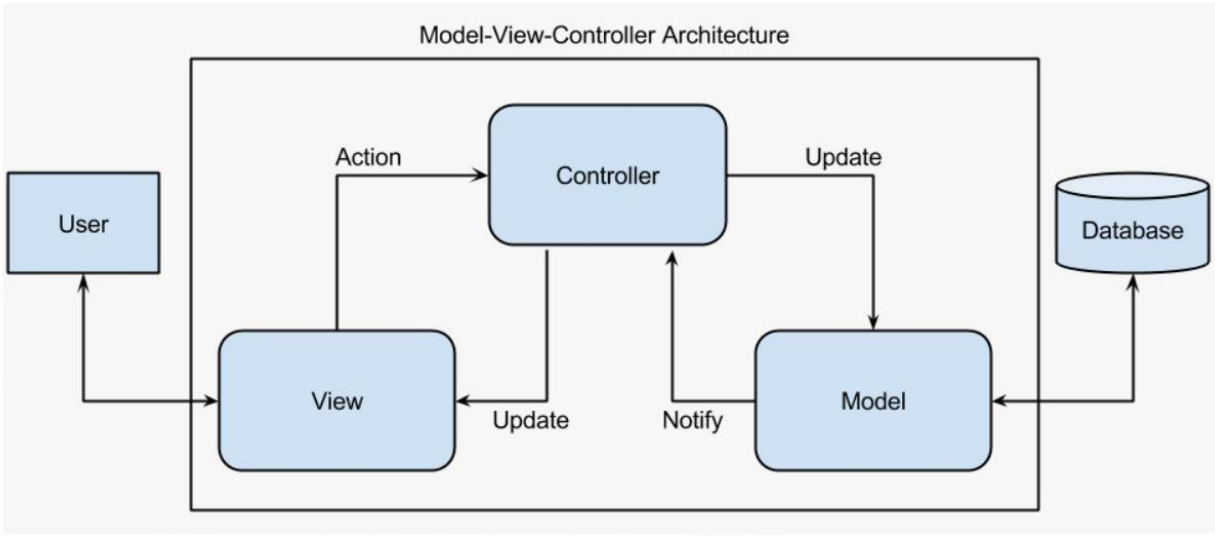
Singleton

The singleton pattern is one of the simplest design patterns. Sometimes we need to have only one instance of our class for example a single DB connection shared by multiple objects as creating a separate DB connection for every object may be costly. Similarly, there can be a single configuration manager or error manager in an application that handles all problems instead of creating multiple managers.



We use singleton in many classes like (Audio – GameDataManager – CoLorFactory – MovingObject – Memento)

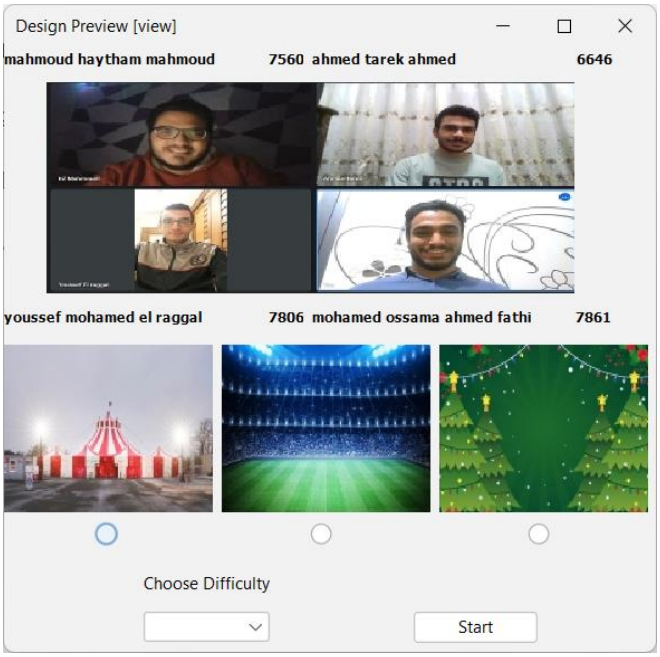
MVC



Model : A model is data used by a program. It will be Levels

```java
package MVC;


public class Model {

    private String[] Levels = {"Easy","Medium","Hard"};

    public String[] getLevels()
    {
        return Levels;
    }
}
```

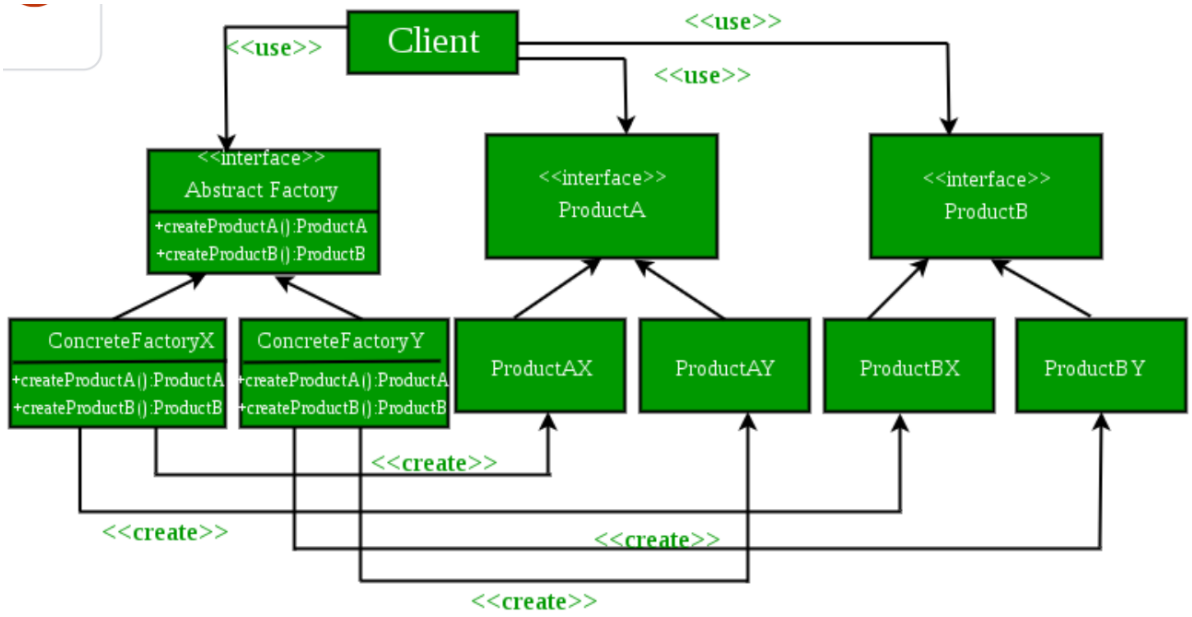
View: is the means of displaying objects within an application. Examples include displaying a window or buttons or text within a window. It includes anything that the user can see.

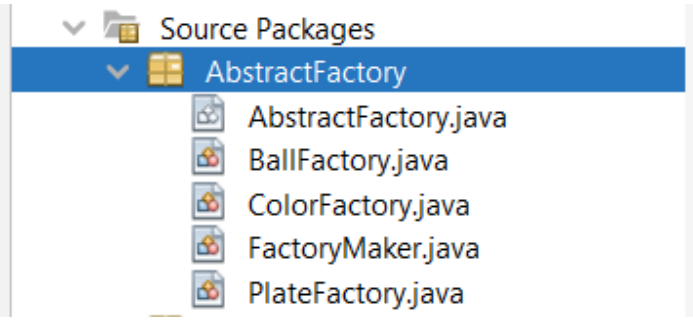


Control: A controller updates both models and views. It accepts input and performs the corresponding update. For example, a controller can update a model by changing the attributes of a character in a video game. It may modify the view by displaying the updated character in the game.

```
package MVC;

public class Control {
    public view view;
    public Model model;
    public Control(view view , Model model)
    {
        this.view=view;
        this.model=model;
        this.view.setLevels( Levels: this.model.getLevels());
    }

}
```

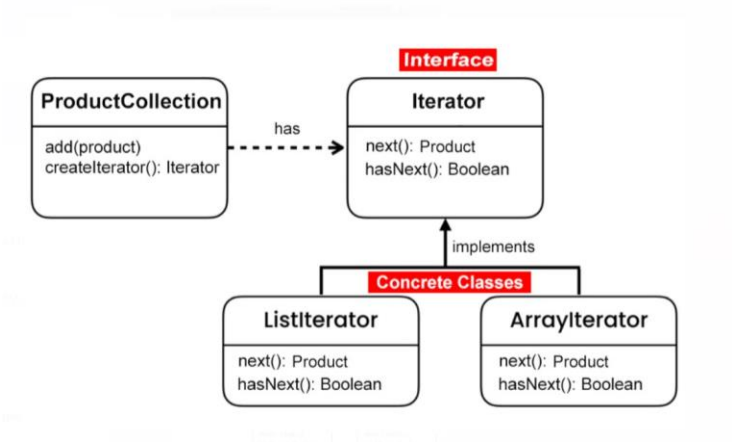
AbstractFactory: is almost similar to Factory Pattern and is considered as another layer of abstraction over factory pattern. Abstract Factory patterns work around a super-factory which creates other factories. Abstract factory pattern implementation provides us with a framework that allows us to create objects that follow a general pattern. So at runtime, the abstract factory is coupled with any desired concrete factory which can create objects of the desired type.





Iterator: is a relatively simple and frequently used design pattern. There are a lot of data structures/collections available in every language. Each collection must provide an iterator that lets it iterate through its objects. However while doing so it should make sure that it does not expose its implementation.

Class diagram of Iterator

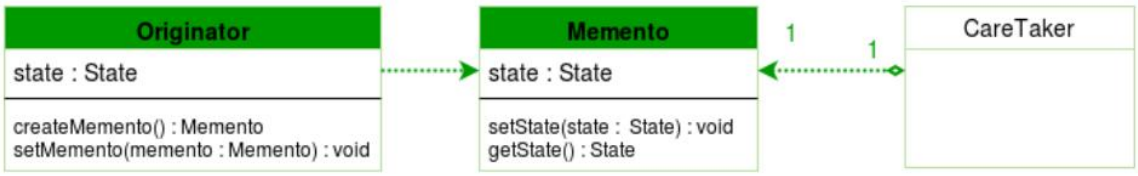


Implementation of Iterator in our code

```
public interface MyIterator
{
    public boolean hasNext();
    public Integer next();
    public void remove();
    public int getIndex();
}
```

```
public class ArrayIterator implements MyIterator {
    private int index ;
    private List<Integer> list;
   public ArrayIterator(List list) {
        this.list = list;
        index=0;
    }

    public boolean hasNext() {
        if (index < list.size()) {
            return true;
        }
        return false;
    }

    public Integer next() {
        if (this.hasNext()) {
            return list.get(index++);
        }
        return 0;
    }

    public void remove() {
        list.remove(--index);
    }

    public int getIndex() {
       return index;
    }
```

Memento : is a behavioral design pattern. Memento pattern is used to restore state of an object to a previous state. As your application is progressing, you may want to save checkpoints in your application and restore back to those checkpoints later



The System create a checkpoint during game every 15 seconds and save the score of the game

```
public class Orginator {
    private static Orginator instance;
    private int[] Score = new int[2];
    private Orginator() {
    }
    public static Orginator getInstance() {

        if (instance == null) {
            instance = new Orginator();
        }
        return instance;
    }


    public void Set(int score , int Time) {
         Score[0]=score;
         Score[1]=Time;
         System.out.println("Score is "+Score[0] + " Time is "+Score[1]);
    }

    public Memento createMomento()
    {
       System.out.println("Score is "+Score[0] + " Time is "+Score[1]);
        return new Memento(Score[0],Score[1]);
    }

    public int[] RestoreLastMemento(Memento memento)
    {
        int[] score =memento.getScore();
        return score;
    }

  public class Memento {

      private int info[] = new int[2];
      public Memento(int scores , int currentTime) {
          info[0] = scores;
          info[1]=currentTime;
      }

      public int[] getScore()
      {
          return info;
      }

  }
```

```java
public class Caretaker {

    private static Caretaker instance;
    private ArrayList<Memento> ScoresList = new ArrayList<Memento>();
    private int scoreCount=0;

    private Caretaker() {
    }
    public static Caretaker getInstance() {

        if (instance == null) {
            instance = new Caretaker();
        }
        return instance;
    }

    public void addMemento(Memento score)
    {
        System.out.println("Prev "+ScoresList.size());
        ScoresList.add( e: score);
        scoreCount++;
        System.out.println("After "+ScoresList.size());
    }

    public Memento getLastMemento()
    {
        System.out.println( x: ScoresList.size());
        return ScoresList.get(ScoresList.size()-1);
    }

}
```

Observer: is a software design pattern in which an object, named the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods.

```java
public interface Subject {

    public void attach(Observer observer);
    public void deattach(Observer observer);
    public void notifyObservers();

}
```

```java
public interface Observer {
    public void update();
}
```

```java
ublic class PileSubject implements Subject {

  private List<Observer> observers = new ArrayList<Observer>();
  private int size;
  public PileSubject()
  {
      size=0;
  }
  @Override
  public void attach(Observer observer) {
      observers.add( e: observer);
      System.out.println( x: "observer plate added");
      size++;
      System.out.println("SIZE OF ARRAY: "+size);
  }
   @Override
  public void deattach(Observer observer) { // lw 3 atba2 nfs lon hy5fyhom
      observers.remove( o: observer);
      System.out.println( x: "observer plate removed");
      size--;
  }
  @Override
  public void notifyObservers() {

      for (int i = 0; i < observers.size(); i++) {
          observers.get( index: i).update();
        // deattach(observers.get(lastElement));
      }
  }
```

State : is used when an Object changes its behavior based on its internal state.

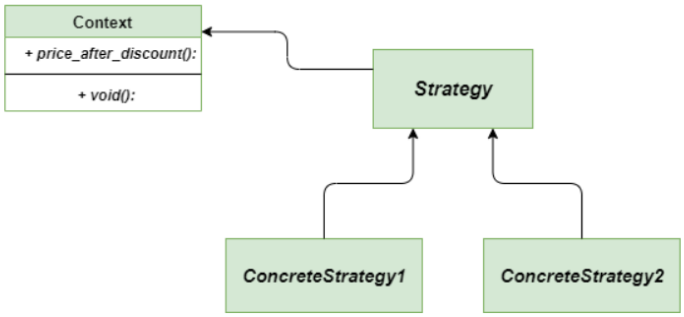
```
public interface DifficultyState {
    public int getNoOfBars();
    public int getGameSpeed();

}
```

```
public class Easy implements DifficultyState {

    @Override
    public int getNoOfBars() {
        return 2;
    }

    @Override
    public int getGameSpeed() {
        return 10;
    }

}
```

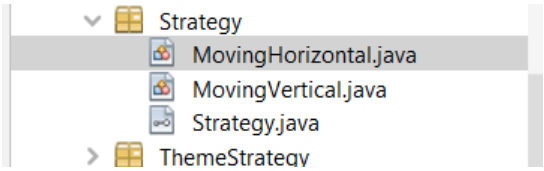
```
public class Medium implements DifficultyState {

    @Override
    public int getNoOfBars() {
        return 3;
    }

    @Override
    public int getGameSpeed() {
        return 7;
    }
```

```
public class Hard implements DifficultyState {

    @Override
    public int getNoOfBars() {
        return 4;
    }

    @Override
    public int getGameSpeed() {
        return 3;
    }

}
```

Strategy :Behavioral Design pattern that allows you to define the complete family of algorithms, encapsulates each one and putting each of them into separate classes and also allows to interchange there objects.



class-diagram-Strategy-method




```java
public interface Strategy {

    boolean isMovingHorizontal();
    void move(Shape plate);

}
```

```java
public class MovingHorizontal implements Strategy{

    private GameDataManager manager;

    public MovingHorizontal() {
        this.manager = GameDataManager.getInstance();
    }

    @Override
    public boolean isMovingHorizontal() {
        return true;
    }

        @Override
    public void move(Shape shape){
        if (shape.getX() < (int) (manager.getWidth() * 0.4) && shape.getY()<=50) {
                shape.setX((shape.getX() + 2));
        }else if(shape.getX() < (int) (manager.getWidth() * 0.2) && shape.getY()>50){
                shape.setX((shape.getX() + 2));
        }else if (shape.getX() > (int) (manager.getWidth() -
                GameDataManager.getInstance().getWidth() * 0.4 - shape.getWidth())&& shape.getY()<=50) {
                shape.setX((shape.getX() - 2));
        }else if(shape.getX() > (int) (manager.getWidth() -
                GameDataManager.getInstance().getWidth() * 0.2 - shape.getWidth())&& shape.getY()>50){
                shape.setX((shape.getX() - 2));
        }

    }
```

```java
public class MovingVertical implements Strategy {

    @Override
    public boolean isMovingHorizontal() {
        return false;
    }

     @Override
    public void move(Shape shape){
         shape.setY(shape.getY() + 2);
    }

}
```

Notes

Flyweight design pattern

we use the concept of flyweight design pattern in class BallFactory & PlateFactory by using static HashMap that stores every shape is made and every time we need to create a new shape w check this shape is in HashMap with a choosen color or not if is then we check if this is the shape is on screen or not and if the shape is not on the screen we will set position of this shape and reuse it.