Alexandria University

Faculty of Engineering

Computer and Systems Engineering Dept.

The Smurfs Game

Team Members:

• Aya Abouzeid	2
• Aya Fouad	3
• Salma Yehia	35
• Nada Ayman	79

1) Introduction:

a. Project Description:

It is two-player game (one uses keyboard, and the other uses the mouse) in which each character carry two stacks of plates, and there are a set of colored different dynamically loaded shapes that end up falling down and players try to catch the falling shapes. The game has three levels of difficulty (Easy – Medium - Hard), with background music for more fun.

The main aim of the project is to implement an advanced user friendly GUI, and use at minimum eleven different design patterns to avoid any problems commonly occurring in software design on a large scale.

The project is designed in **Java**, GUI implemented using **JavaFX**, saving the game to load later in JSON files using **GSON**, besides complete logging file using **Log4j**.

b. Report Overview:

i. Software Design:

Illustrates the design and how the code is organized between different packages and classes, also explains the role of each class and how it interacts with the rest of them.

ii. Class Diagrams:

A type of static structure diagrams that describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among objects.

iii. Sequence Diagrams:

An interaction diagram that shows how objects operate with one another and in what order. It is a construct of a message sequence chart. It shows object interactions arranged in time sequence.

iv. Design Patterns:

Illustrates thoroughly the design patterns used, why they are most suitable where they are used and how they helped overcome design possible problems.

v. GUI Snapshots:

Shows the GUI of the game with different snapshots

vi. User Guide:

Shows the user how to use the project / game and the way to win.

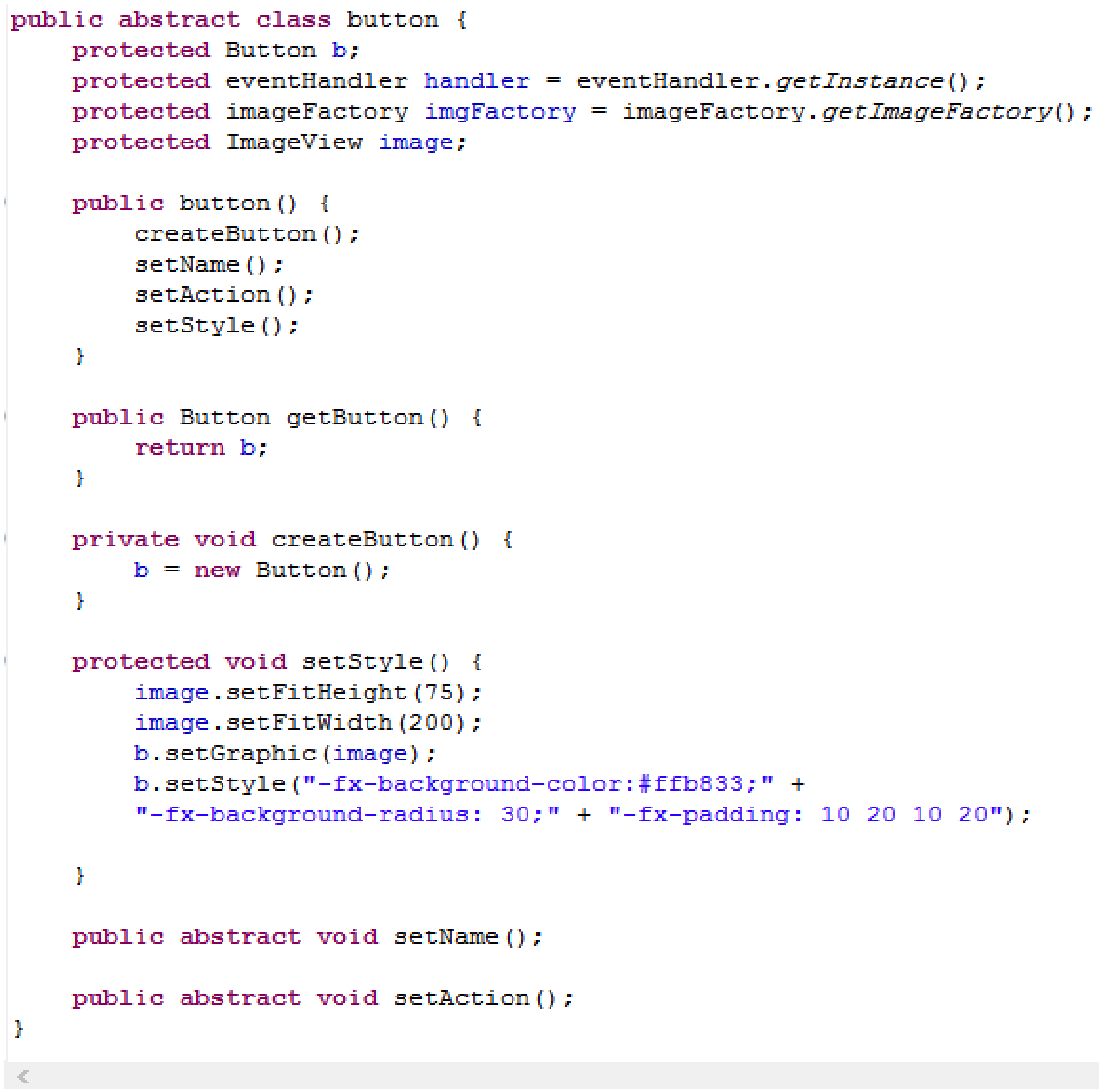
2) Software Design:

The project is divided into many packages:

- **Button**

  Contains the “button” abstract class that all the other button classes (14 Class)extend (GameOptionsButton – MainMenuButton – LoadGameButton – HardButton – ContinueGameButton – EasyButton …etc).

  The Button Abstract Class:

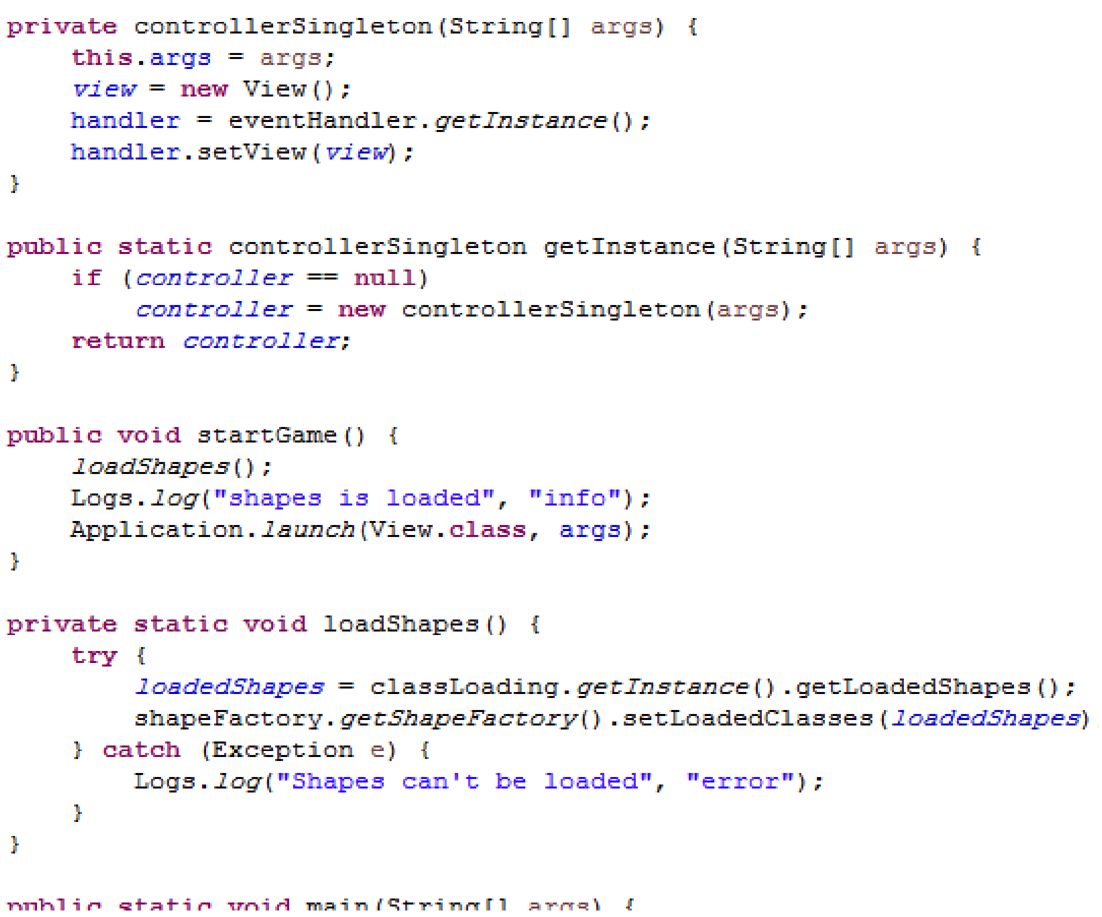


```
public abstract class button {
    protected Button b;
    protected eventHandler handler = eventHandler.getInstance();
    protected imageFactory imgFactory = imageFactory.getImageFactory();
    protected ImageView image;

    public button() {
        createButton();
        setName();
        setAction();
        setStyle();
    }

    public Button getButton() {
        return b;
    }

    private void createButton() {
        b = new Button();
    }

    protected void setStyle() {
        image.setFitHeight(75);
        image.setFitWidth(200);
        b.setGraphic(image);
        b.setStyle("-fx-background-color:#ffb833;" +
        "-fx-background-radius: 30;" + "-fx-padding: 10 20 10 20");

    }

    public abstract void setName();

    public abstract void setAction();
}
```

- **Controller**

  The classes that controls the game

  ControllerSingleton: Has the “main” method and initializes everything as the game starts.



```
private controllerSingleton(String[] args) {
    this.args = args;
    view = new View();
    handler = eventHandler.getInstance();
    handler.setView(view);
}

public static controllerSingleton getInstance(String[] args) {
    if (controller == null)
        controller = new controllerSingleton(args);
    return controller;
}

public void startGame() {
    loadShapes();
    Logs.log("shapes is loaded", "info");
    Application.launch(View.class, args);
}

private static void loadShapes() {
    try {
        loadedShapes = classLoading.getInstance().getLoadedShapes();
        shapeFactory.getShapeFactory().setLoadedClasses(loadedShapes)
    } catch (Exception e) {
        Logs.log("Shapes can't be loaded", "error");
    }
}

public static void main(String[] args) {
```

eventHandler - game Controller - Game Options music player TimerThread)

- **Factories :** contains the factories in the game (buttonFactory, ImageFactory, sceneFactory, shapeFactory).

- **Iterator:** Contains (creatIterator Interface – Iterator Interface) and the concrete iterators (PlayeIterator – ShapeIterator) that implements the Iterator Interface.

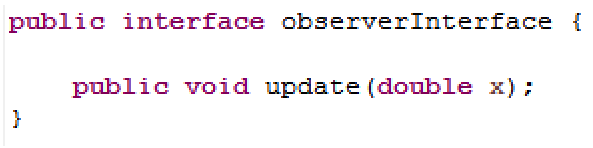
- **Layouts:** Contains the layout abstract Class that all the other layouts (9 Classes) extend (EndGame – mainOptions – Start - Instructions …etc).

```
public abstract class layout {

    protected double windowHeight;
    protected double windowWidth;
    protected Scene scene;
    protected buttonFactory factory;
    protected imageFactory imgFactory;

    public eventHandler handler = eventHandler.getInstance();
    public layout(double height, double width) {
        windowHeight = height;
        windowWidth = width;
        factory = buttonFactory.getButtonFactory();
        imgFactory = imageFactory.getImageFactory();
    }

    public Scene getScene() {
        return scene;
    }
}
```

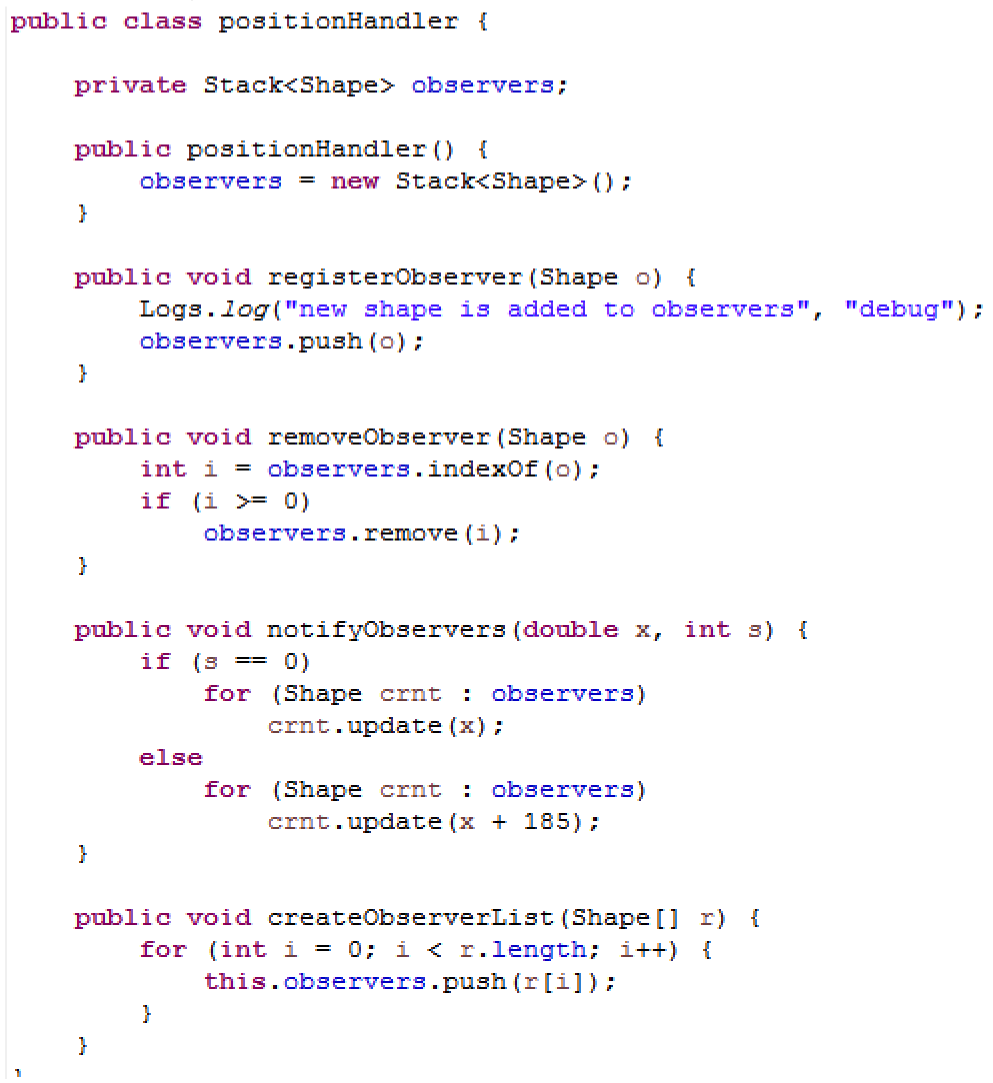
- **Logs**
  Contains the class responsible for writing the logs to the logging.log file.

```
public class Logs {

    static final Logger logger = LogManager.getLogger(Logs.class);

    public static void log(String msg, String level) {
        switch (level) {
        case "info":
            logger.info(msg);
            break;
        case "debug":
            logger.debug(msg);
            break;
        case "error":
            logger.error(msg);
            break;
        case "fatal":
            logger.fatal(msg);
            break;
        case "warn":
            logger.warn(msg);
            break;
        default:
            break;
        }
    }
}
```

- **Observer**
  Contains the observer interface that the observers (Shapes) implement and the PositionHandling class that notify the observers with the changes in position.



```java
public interface observerInterface {

    public void update(double x);
}
```

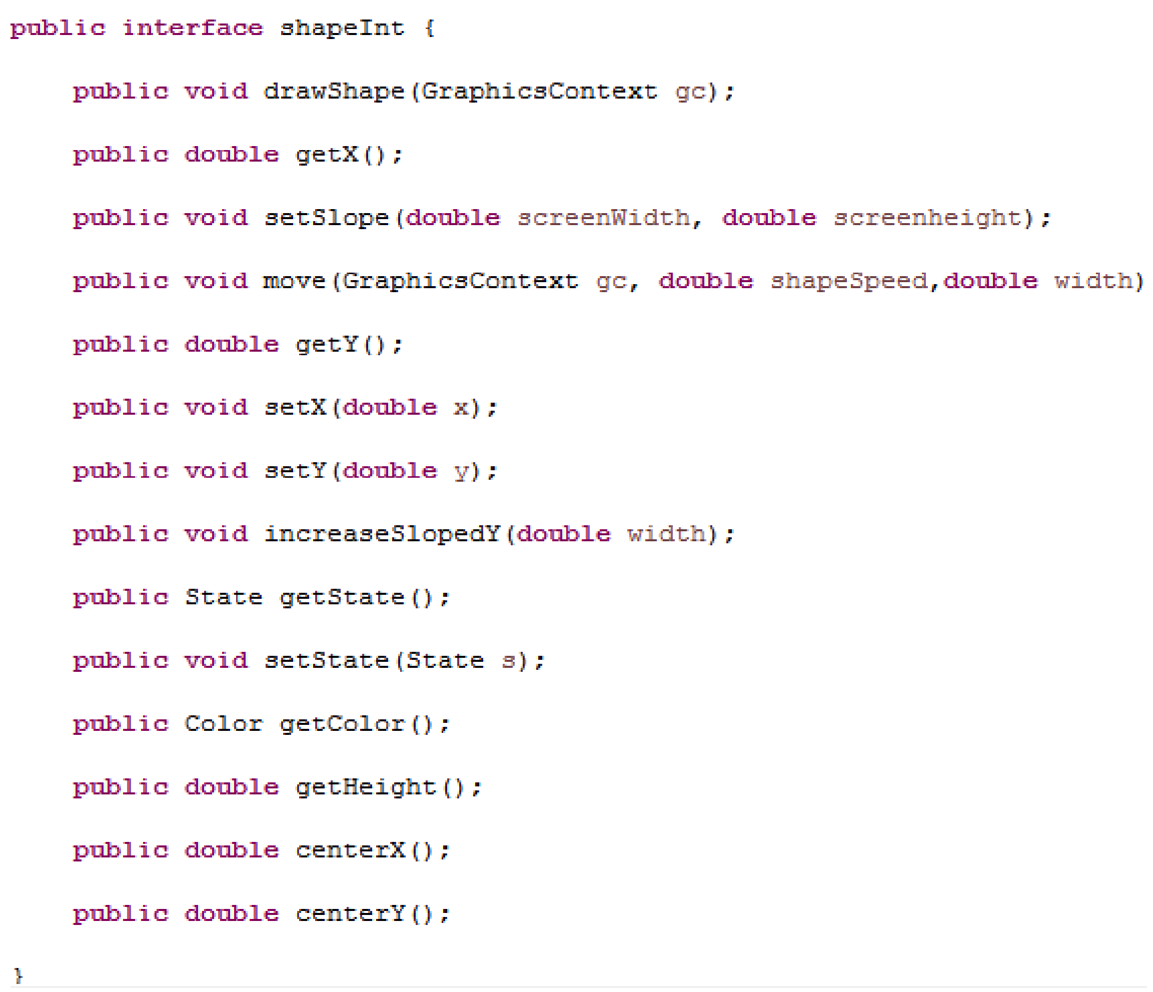


```java
public class positionHandler {

    private Stack<Shape> observers;

    public positionHandler() {
        observers = new Stack<Shape>();
    }

    public void registerObserver(Shape o) {
        Logs.log("new shape is added to observers", "debug");
        observers.push(o);
    }

    public void removeObserver(Shape o) {
        int i = observers.indexOf(o);
        if (i >= 0)
            observers.remove(i);
    }

    public void notifyObservers(double x, int s) {
        if (s == 0)
            for (Shape crnt : observers)
                crnt.update(x);
        else
            for (Shape crnt : observers)
                crnt.update(x + 185);
    }

    public void createObserverList(Shape[] r) {
        for (int i = 0; i < r.length; i++) {
            this.observers.push(r[i]);
        }
    }
}
```

- **Player:** This package contains the following classes:
  - Player: contains the different attributes of the player as well as a method to notify the stacks of each player that the player moved

- Player Proxy: it contains the same attributes of the player object except some attributes that we aren't concerned in saving. Used in saving
- Build player proxy: this class builds the player proxy object from the player object when saving.
- Build player: this class builds the player from the player proxy object that was saves. Used while loading the shapes.

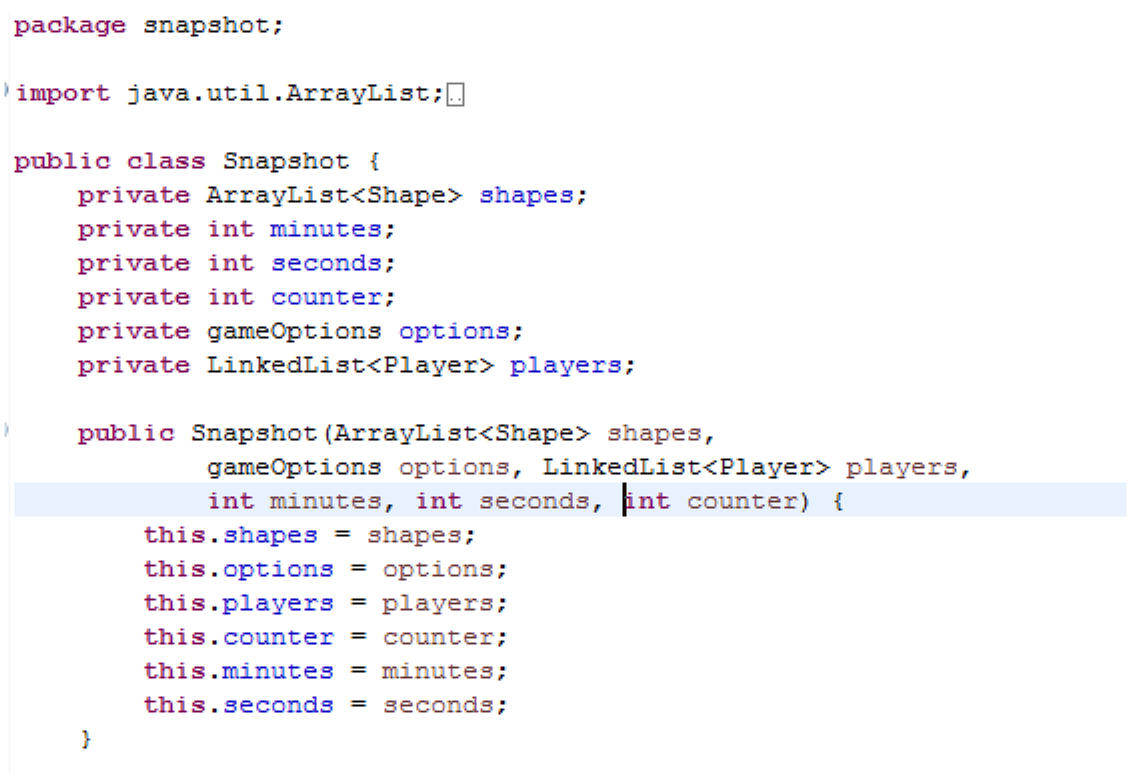
- **Save** : This package contains :
    - Save: this class is used while saving the objects and while loading them it is typically for saving.
    - Get array: this class is used to transform between array lists & arrays this is because the saving code needed arrays while the loading code needed array lists
- **Shape:** contains the shape Interface that all dynamically loaded shapes implement and the shape abstract class
    - ShapePool: the objectPool class to store the unused shapes for later needs.
    - shapeProxy: it contains the same attributes as the shape object except some fields that doesnt concern us while saving the shape. it is used in the saving process as we save the shape proxy instead of the shape it self because some of the shape's fields are not serializable.
    - buildShape: it is used to build the shape object from the proxy object after loading the proxy
    - buildShapeProxy: it is used to build the proxy shape from the real shape while saving because we are saving the proxy object



```
public interface shapeInt {

    public void drawShape(GraphicsContext gc);

    public double getX();

    public void setSlope(double screenWidth, double screenheight);

    public void move(GraphicsContext gc, double shapeSpeed,double width)

    public double getY();

    public void setX(double x);

    public void setY(double y);

    public void increaseSlopedY(double width);

    public State getState();

    public void setState(State s);

    public Color getColor();

    public double getHeight();

    public double centerX();

    public double centerY();

}
```

Snapshot This package contain a snapshot class which has the main parameters that define a game so that it can build a game from these parameters

Such as game options, timer of the game, players , the winningstrategy and the gamestrategy of the game. An oject of snapshot is created in the gameController when the user pause the game.

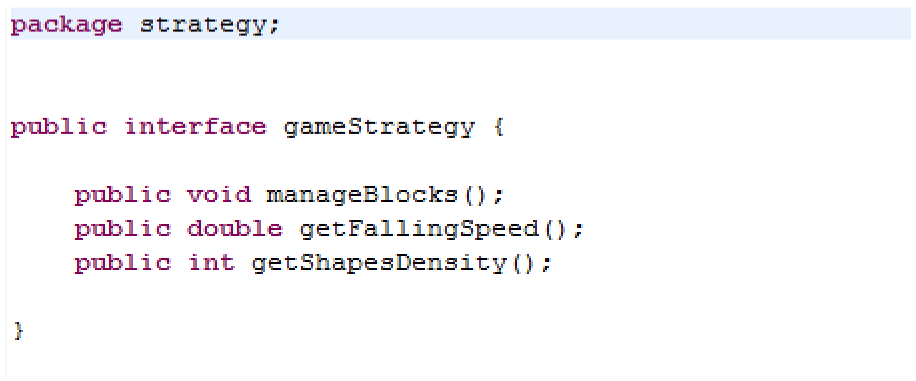


```
package snapshot;

import java.util.ArrayList;

public class Snapshot {
    private ArrayList<Shape> shapes;
    private int minutes;
    private int seconds;
    private int counter;
    private gameOptions options;
    private LinkedList<Player> players;

    public Snapshot(ArrayList<Shape> shapes,
            gameOptions options, LinkedList<Player> players,
            int minutes, int seconds, int counter) {
        this.shapes = shapes;
        this.options = options;
        this.players = players;
        this.counter = counter;
        this.minutes = minutes;
        this.seconds = seconds;
    }
```

- **States:** Contains the StackState and the ShapeIstate interfaces which are implemented by the concrete states
  StackState (Different - same)
  ShapeState (Stored – cought – fallingFtomRight - fallingFromRight)

- **Strategy :**
  gameStrategy interface:

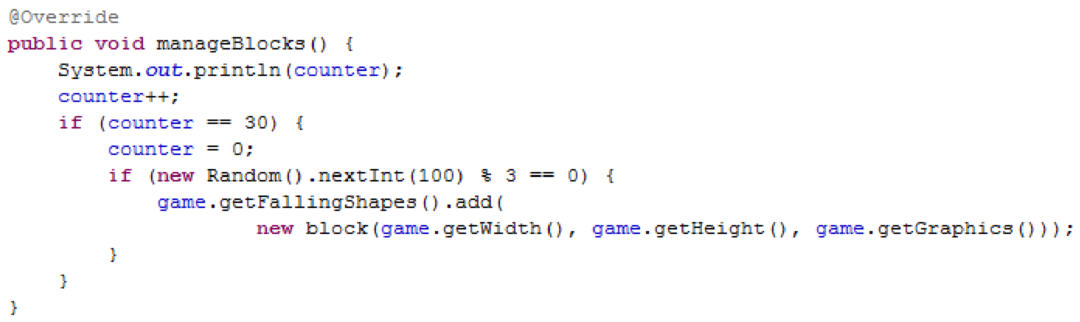
  This interface is used to determine the difficulty level strategy that control the speed of the falling shapes and the density of the blocks which are used to block the players’ stacks so that he cannot collect shapes any more.



```java
package strategy;


public interface gameStrategy {

    public void manageBlocks();
    public double getFallingSpeed();
    public int getShapesDensity();

}
```

A. difficultGame:

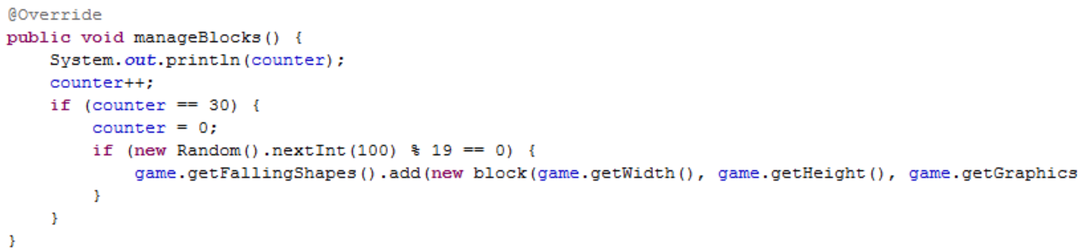
   in the difficult game the speed of the falling shapes is high and the density of the blocks are chosen randomly to have relatively high density than other levels



```java
@Override
public void manageBlocks() {
    System.out.println(counter);
    counter++;
    if (counter == 30) {
        counter = 0;
        if (new Random().nextInt(100) % 3 == 0) {
            game.getFallingShapes().add(
                    new block(game.getWidth(), game.getHeight(), game.getGraphics()));
        }
    }
}
```

B. mediumGame:

   it has medium speed of the falling shapes and also the density of the blocks are chosen randomely to have relatively low density than difficult game



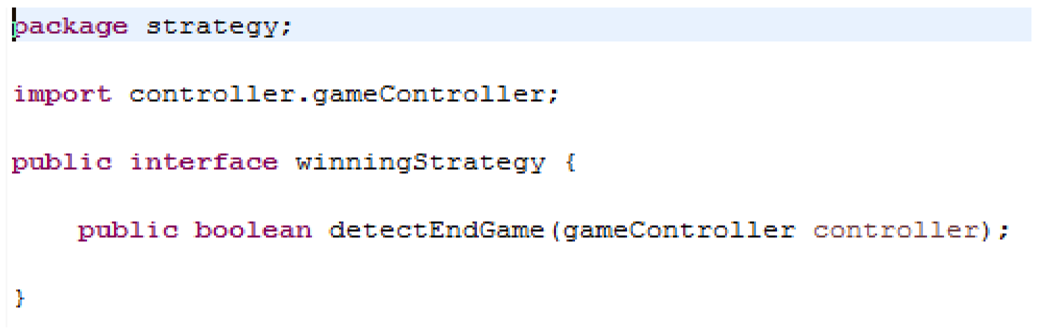
```java
@Override
public void manageBlocks() {
    System.out.println(counter);
    counter++;
    if (counter == 30) {
        counter = 0;
        if (new Random().nextInt(100) % 19 == 0) {
            game.getFallingShapes().add(new block(game.getWidth(), game.getHeight(), game.getGraphics
        }
    }
}
```

C. easyGame:

   it has slow speed of the falling shapes and it has no blocks falling

D. WinningStratrgy interface:

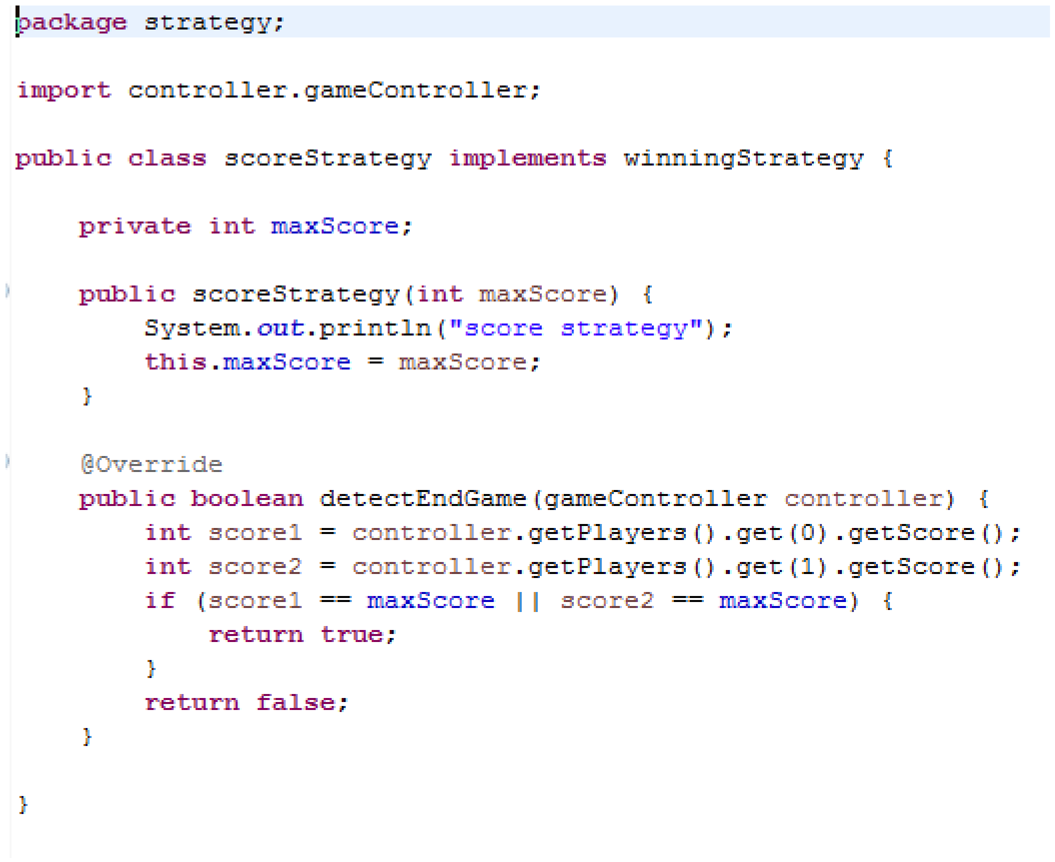
   This interface that determine if the game has ended or not by using different strategies



```java
package strategy;

import controller.gameController;

public interface winningStrategy {

    public boolean detectEndGame(gameController controller);

}
```

E. ScoreStrategy:

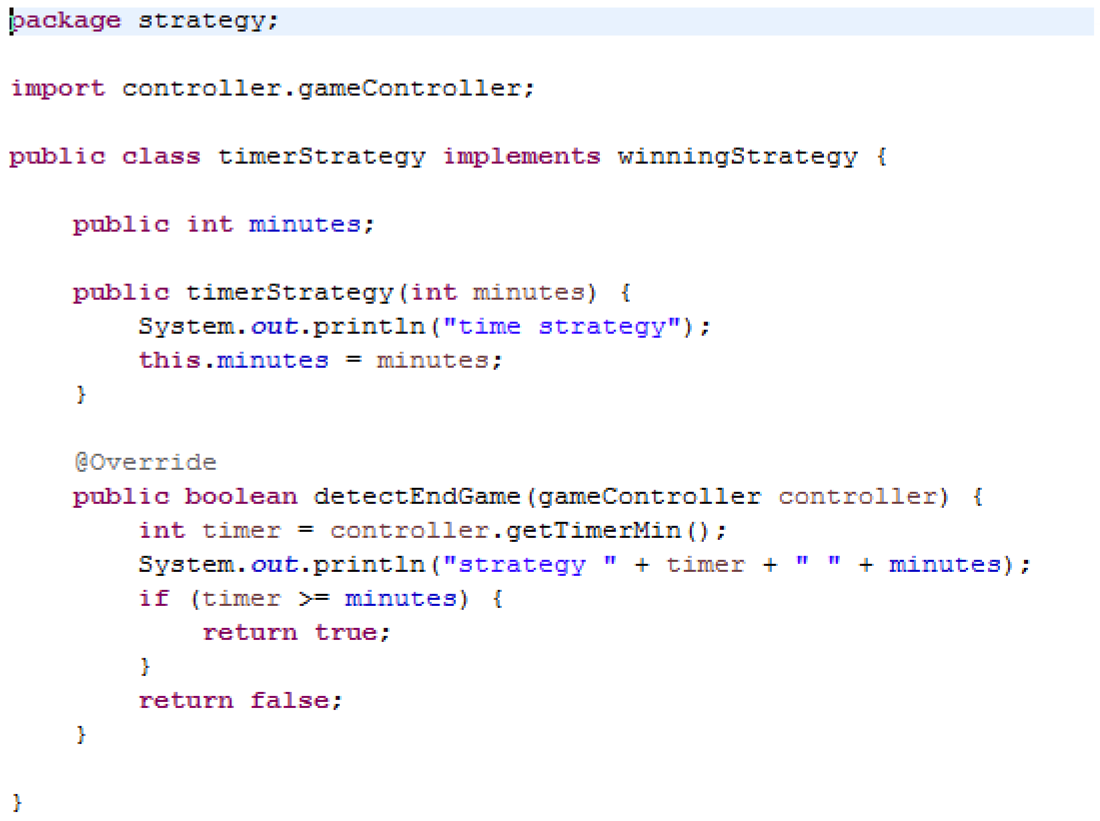
This strategy compares the current scores of the players by a max score that is chosen by the user in the options and it is 2 by default



```java
package strategy;

import controller.gameController;

public class scoreStrategy implements winningStrategy {

    private int maxScore;

    public scoreStrategy(int maxScore) {
        System.out.println("score strategy");
        this.maxScore = maxScore;
    }

    @Override
    public boolean detectEndGame(gameController controller) {
        int score1 = controller.getPlayers().get(0).getScore();
        int score2 = controller.getPlayers().get(1).getScore();
        if (score1 == maxScore || score2 == maxScore) {
            return true;
        }
        return false;
    }

}
```

TimerStrategy:

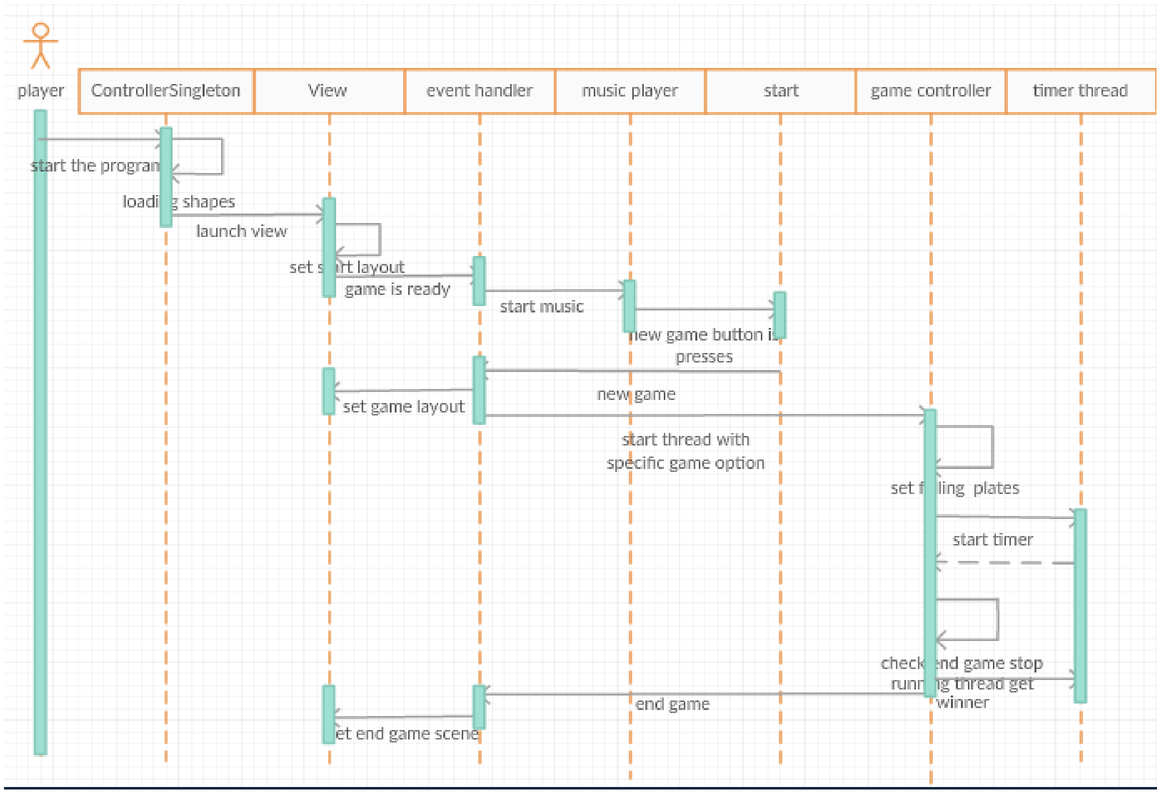
This strategy compares the current timer by a max timer that is set by the player in the options and it is 2 minutes by default



```java
package strategy;

import controller.gameController;

public class timerStrategy implements winningStrategy {

    public int minutes;

    public timerStrategy(int minutes) {
        System.out.println("time strategy");
        this.minutes = minutes;
    }

    @Override
    public boolean detectEndGame(gameController controller) {
        int timer = controller.getTimerMin();
        System.out.println("strategy " + timer + " " + minutes);
        if (timer >= minutes) {
            return true;
        }
        return false;
    }

}
```

3) Class Diagrams:

4) Sequence Diagram:

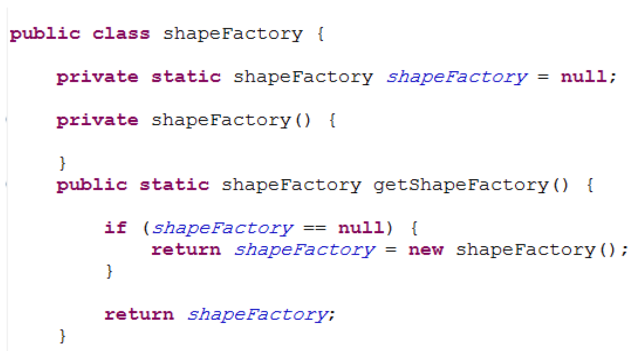


5) Design patterns:

1. Singleton:

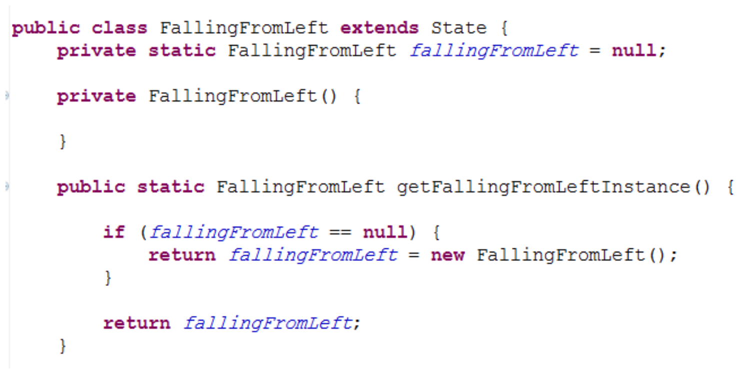
Singleton design Pattern is used in our project when a restriction was to be made on a class to limit it's instantiation to only one object and provide global access to that instance, this is typically done by declaring the class's constructors to be private and providing a private static instance and a method that returns nothing but that one and only instance.
Singleton pattern was chosen to be applied on Factory and State Design Patterns, as well as The Game Controller.

A. Singleton applied on Factory Design Pattern:



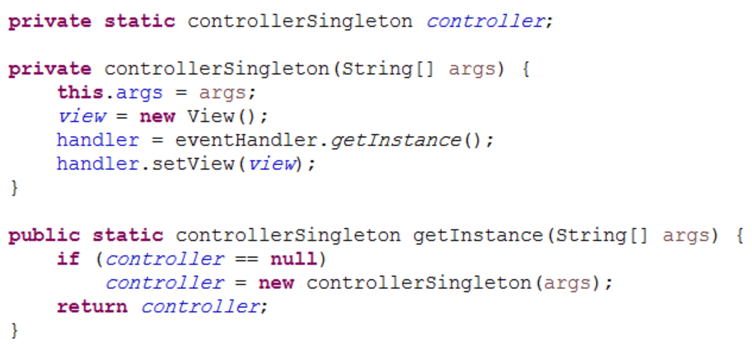
```
public class shapeFactory {

    private static shapeFactory shapeFactory = null;

    private shapeFactory() {

    }
    public static shapeFactory getShapeFactory() {

        if (shapeFactory == null) {
            return shapeFactory = new shapeFactory();
        }

        return shapeFactory;
    }
```

B. Singleton applied on State Design Pattern:



```
public class FallingFromLeft extends State {
    private static FallingFromLeft fallingFromLeft = null;

    private FallingFromLeft() {

    }

    public static FallingFromLeft getFallingFromLeftInstance() {

        if (fallingFromLeft == null) {
            return fallingFromLeft = new FallingFromLeft();
        }

        return fallingFromLeft;
    }
```

C. Singleton applied on The Game Controller :



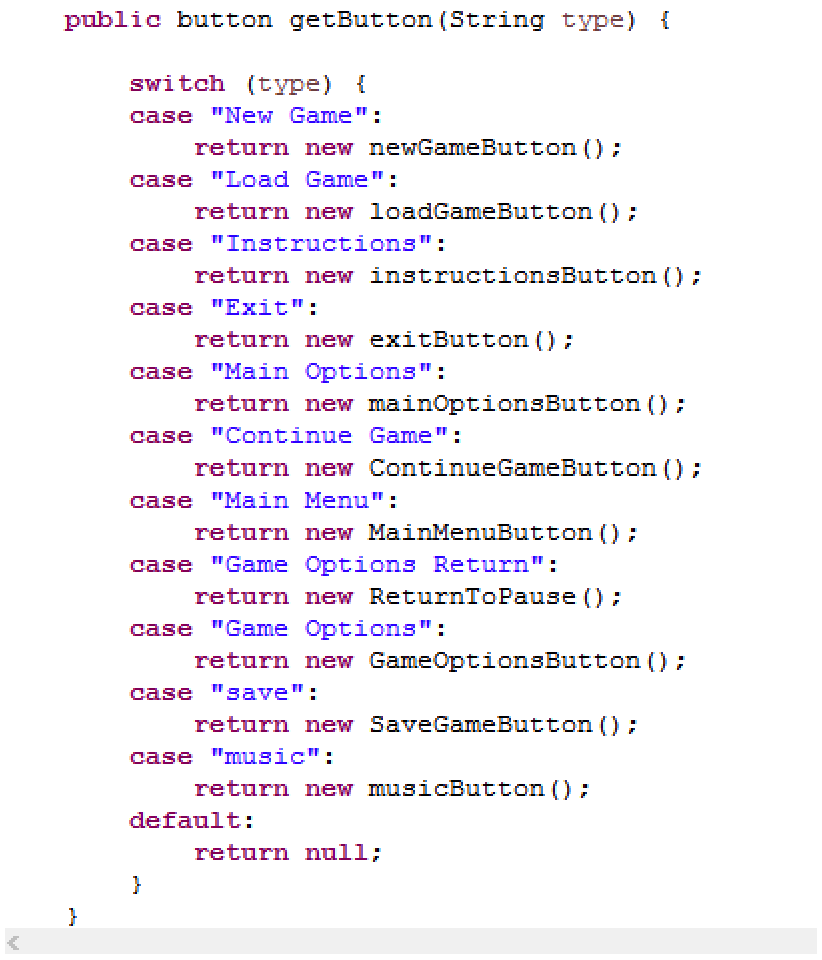
```
private static controllerSingleton controller;

private controllerSingleton(String[] args) {
    this.args = args;
    view = new View();
    handler = eventHandler.getInstance();
    handler.setView(view);
}

public static controllerSingleton getInstance(String[] args) {
    if (controller == null)
        controller = new controllerSingleton(args);
    return controller;
}
```

2. Factory:

Factory Design Pattern is used in our project to create objects on demand without specifying the required class to create this object, where the factory handles this operation and returns the exact required object.

This was applied by four different factories in our design which are Button Factory, Scene Factory, Image Factory, and Shape Factory.

A. Factory applied on Buttons :



```
public button getButton(String type) {

    switch (type) {
    case "New Game":
        return new newGameButton();
    case "Load Game":
        return new loadGameButton();
    case "Instructions":
        return new instructionsButton();
    case "Exit":
        return new exitButton();
    case "Main Options":
        return new mainOptionsButton();
    case "Continue Game":
        return new ContinueGameButton();
    case "Main Menu":
        return new MainMenuButton();
    case "Game Options Return":
        return new ReturnToPause();
    case "Game Options":
        return new GameOptionsButton();
    case "save":
        return new SaveGameButton();
    case "music":
        return new musicButton();
    default:
        return null;
    }
}
```

B. Factory applied on Scenes :

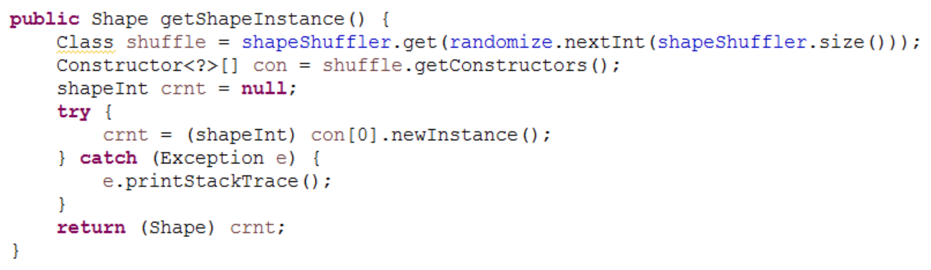
```
public Scene getScene(String name, double Height, double Width) {

    layout scene;

    switch (name) {
    case "Game":
        scene =  new Game(Height, Width);break;
    case "MainMenu":
        scene =  new Start(Height, Width);break;
    case "pause":
        scene =  new Pause(Height, Width);break;
    case "MainOptions":
        scene =  new mainOptions(Height, Width);break;
    case "GameOptions":
        scene =  new gameOptions(Height, Width);break;
    case "Instructions":
        scene =  new Instructions(Height, Width);break;
    case "EndGame":
        scene =  new EndGame(Height, Width);break;
    default:
        return null;
    }

    return scene.getScene();

}
```

C. Factory applied on Image Views:

```
private File getFile(String name) {
    switch (name) {
    case "background":
        return new File("src\\images\\back2.png");
    case "smurfette":
        return new File("src\\images\\2.png");
    case "smurff":
        return new File("src\\images\\1.png");
    case "mainMenu":
        return new File("src\\images\\mainMenu.jpg");
    case "howTo":
        return new File("src\\images\\HowTo.png");
    case "pause":
        return new File("src\\images\\pause.jpg");
    default:
        return null;
    }
}
```

D. Factory applied on Shapes :

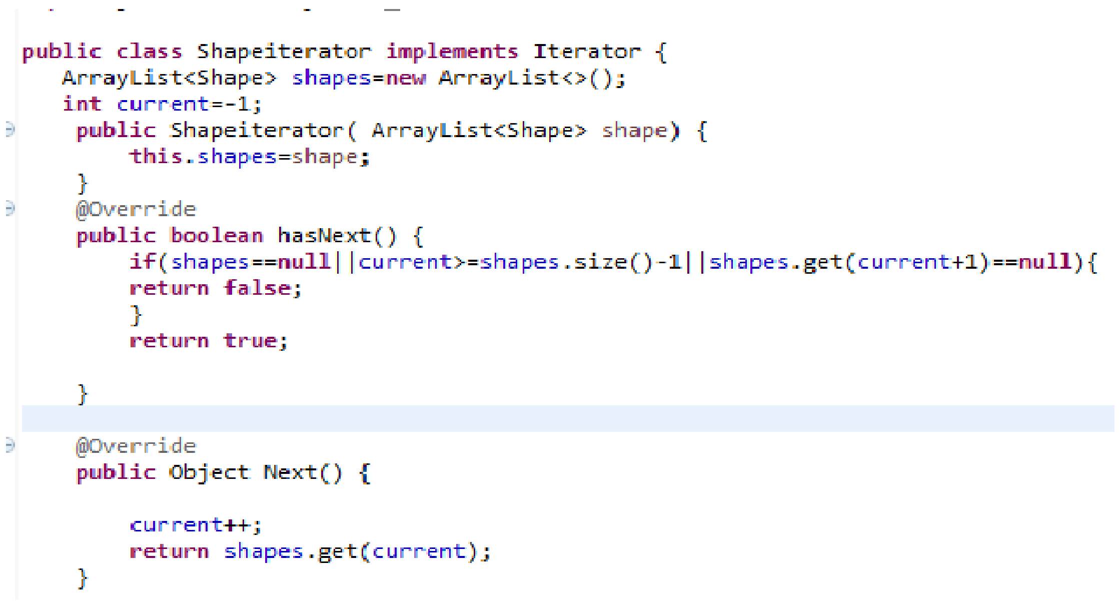
The shuffler here does all the work for randomizing returned shapes.



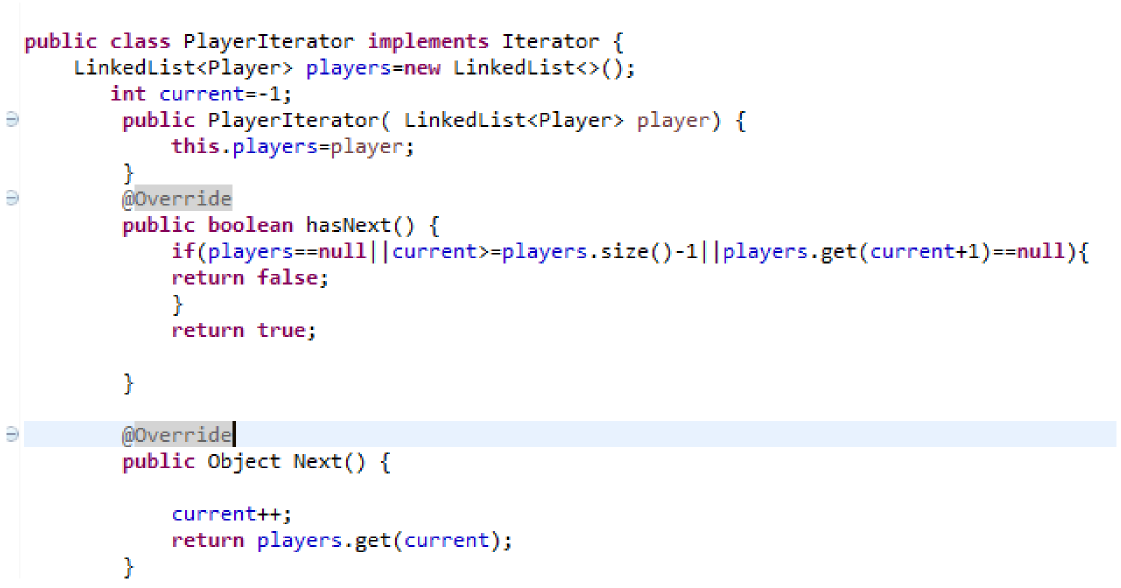
```java
public Shape getShapeInstance() {
    Class shuffle = shapeShuffler.get(randomize.nextInt(shapeShuffler.size()));
    Constructor<?>[] con = shuffle.getConstructors();
    shapeInt crnt = null;
    try {
        crnt = (shapeInt) con[0].newInstance();
    } catch (Exception e) {
        e.printStackTrace();
    }
    return (Shape) crnt;
}
```

3. Iterator:

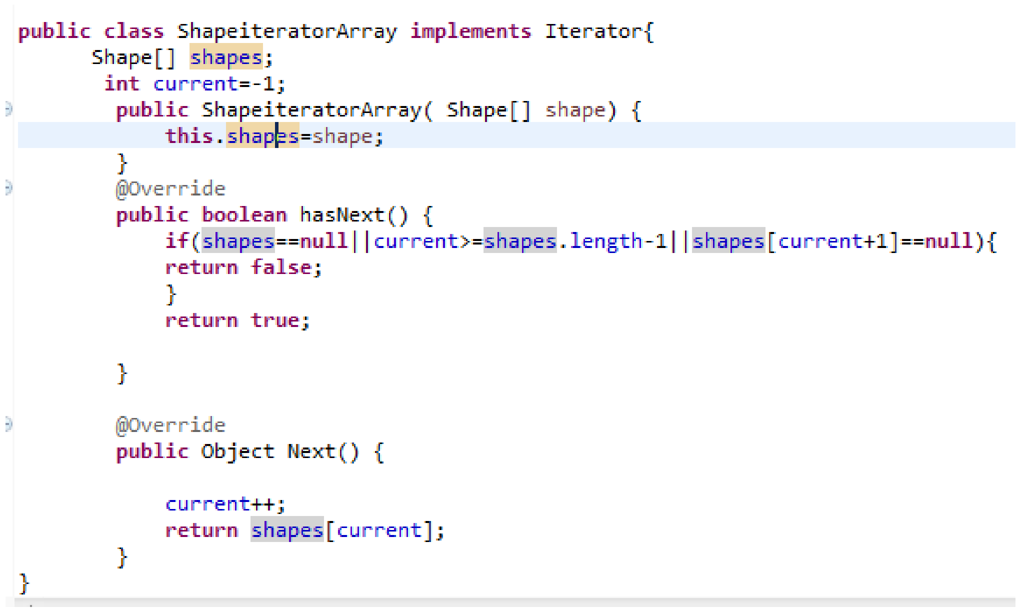
The iterator design pattern is used in our project as a uniform way to iterate through a List of elements without treating them differently they all implement the same methods and are treated similarly



```java
public class Shapeiterator implements Iterator {
   ArrayList<Shape> shapes=new ArrayList<>();
   int current=-1;
    public Shapeiterator( ArrayList<Shape> shape) {
        this.shapes=shape;
    }
    @Override
    public boolean hasNext() {
        if(shapes==null||current>=shapes.size()-1||shapes.get(current+1)==null){
        return false;
        }
        return true;

    }

    @Override
    public Object Next() {

        current++;
        return shapes.get(current);
    }
```

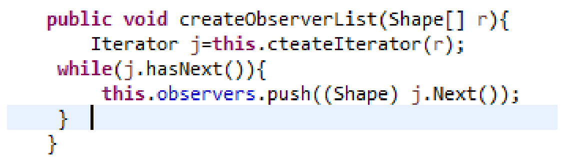


```java
public class PlayerIterator implements Iterator {
    LinkedList<Player> players=new LinkedList<>();
       int current=-1;
        public PlayerIterator( LinkedList<Player> player) {
            this.players=player;
        }
        @Override
        public boolean hasNext() {
            if(players==null||current>=players.size()-1||players.get(current+1)==null){
            return false;
            }
            return true;

        }

        @Override
        public Object Next() {

            current++;
            return players.get(current);
        }
```



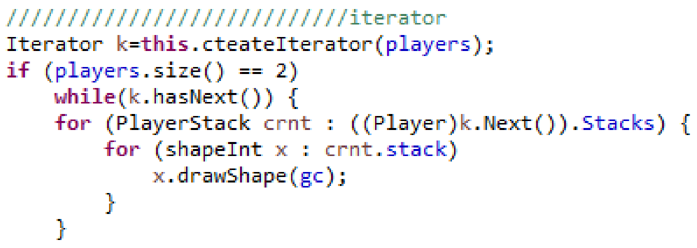
```java
public class ShapeiteratorArray implements Iterator{
      Shape[] shapes;
       int current=-1;
        public ShapeiteratorArray( Shape[] shape) {
            this.shapes=shape;
        }
        @Override
        public boolean hasNext() {
            if(shapes==null||current>=shapes.length-1||shapes[current+1]==null){
            return false;
            }
            return true;

        }

        @Override
        public Object Next() {

            current++;
            return shapes[current];
        }
}
```

A . in Class positionHandler



```
public void createObserverList(Shape[] r){
    Iterator j=this.cteateIterator(r);
 while(j.hasNext()){
     this.observers.push((Shape) j.Next());
 }
}
```

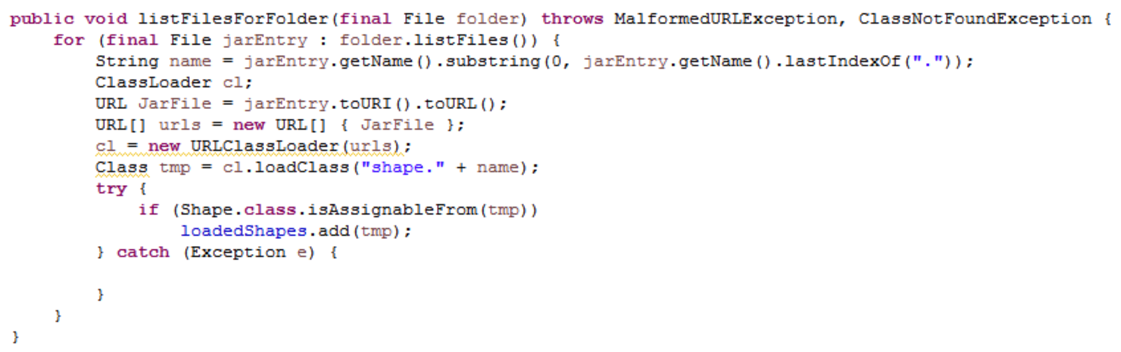
B. in Class gameController:



```
////////////////////////////iterator
Iterator k=this.cteateIterator(players);
if (players.size() == 2)
    while(k.hasNext()) {
    for (PlayerStack crnt : ((Player)k.Next()).Stacks) {
        for (shapeInt x : crnt.stack)
            x.drawShape(gc);
        }
    }
```

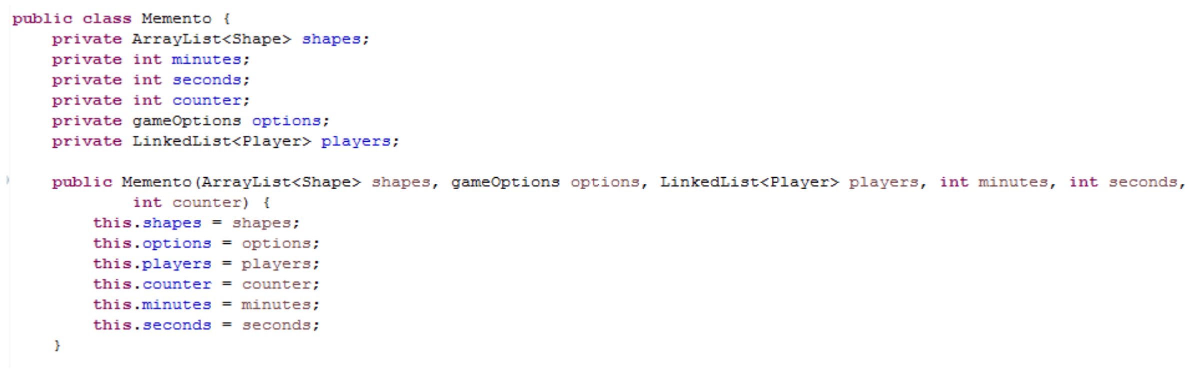
4. Dynamic Linkage:

Dynamic linkage (Dynamic class loading) is used to load all the shapes at the beginning of the game at run time – loads all the jar files is the jars folder after checking that it implements the required interface (shapeInt) by default there are two shapes to load [plate - ellipse] but the user can later add as much shapes as he wants by providing new jars.


```
public void listFilesForFolder(final File folder) throws MalformedURLException, ClassNotFoundException {
    for (final File jarEntry : folder.listFiles()) {
        String name = jarEntry.getName().substring(0, jarEntry.getName().lastIndexOf("."));
        ClassLoader cl;
        URL JarFile = jarEntry.toURI().toURL();
        URL[] urls = new URL[] { JarFile };
        cl = new URLClassLoader(urls);
        Class tmp = cl.loadClass("shape." + name);
        try {
            if (Shape.class.isAssignableFrom(tmp))
                loadedShapes.add(tmp);
        } catch (Exception e) {

        }
    }
}
```

5. Snapshot:

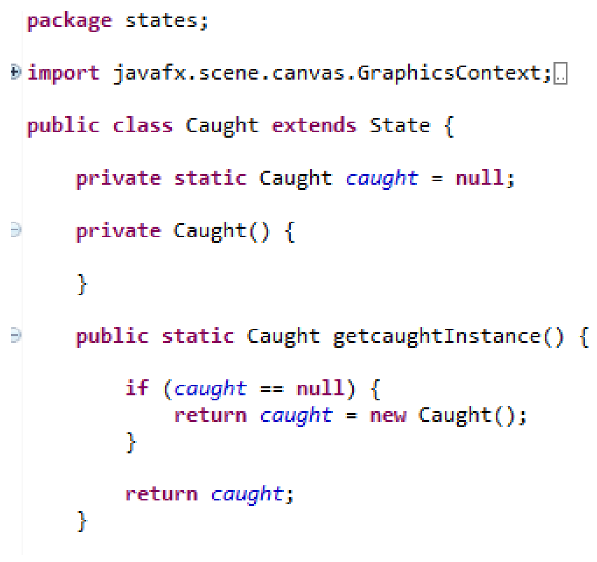
Snapshot is taken when for the game parameters as timer of the game, game options and players' states so that the game can be back from pause where exactly it was. Also when saving the game to be loaded again we just save the snapshot taken for the game.


```
public class Memento {
    private ArrayList<Shape> shapes;
    private int minutes;
    private int seconds;
    private int counter;
    private gameOptions options;
    private LinkedList<Player> players;

    public Memento(ArrayList<Shape> shapes, gameOptions options, LinkedList<Player> players, int minutes, int seconds,
            int counter) {
        this.shapes = shapes;
        this.options = options;
        this.players = players;
        this.counter = counter;
        this.minutes = minutes;
        this.seconds = seconds;
    }
```

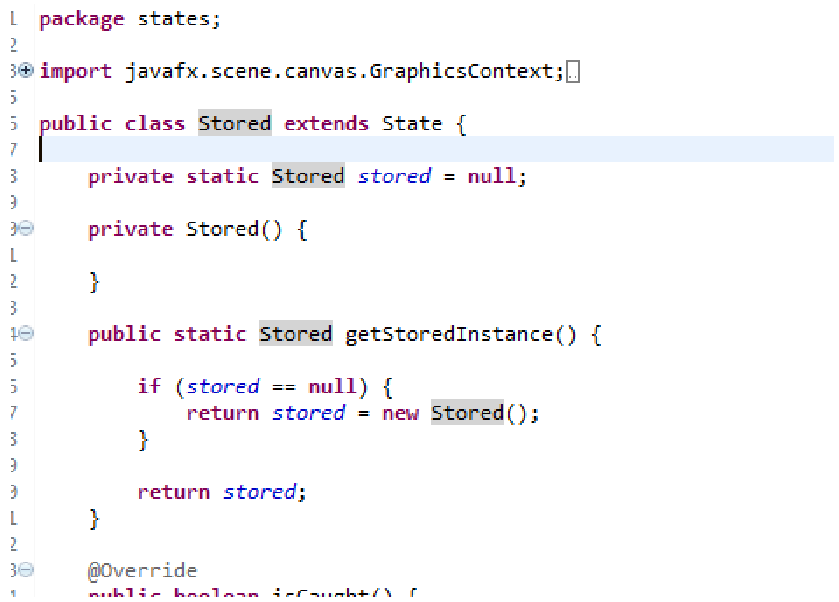
6. State:

The state design pattern is used in our project in to differentiate between different states of the object in order to act differently according to its current state and is applied in two places:

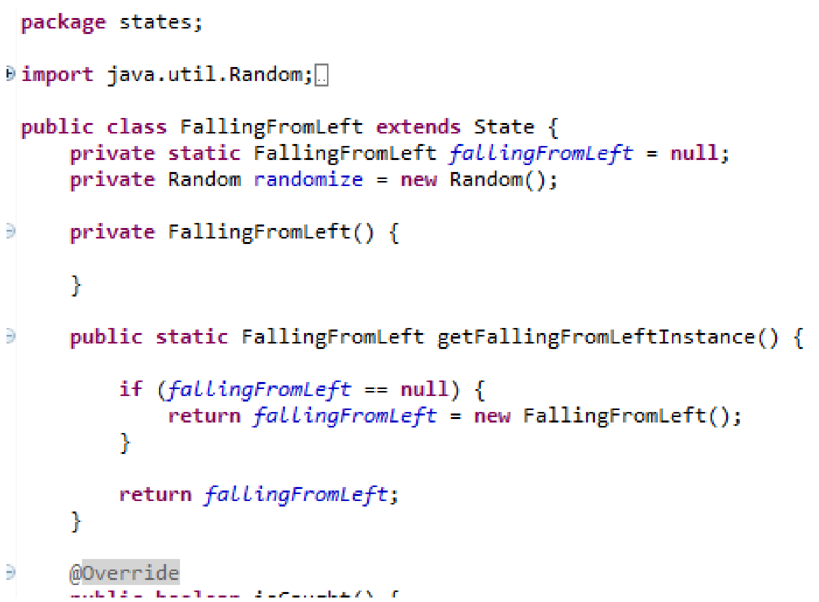
1- Shapes have four states they are either caught, stored (in pool), falling from left or falling from right



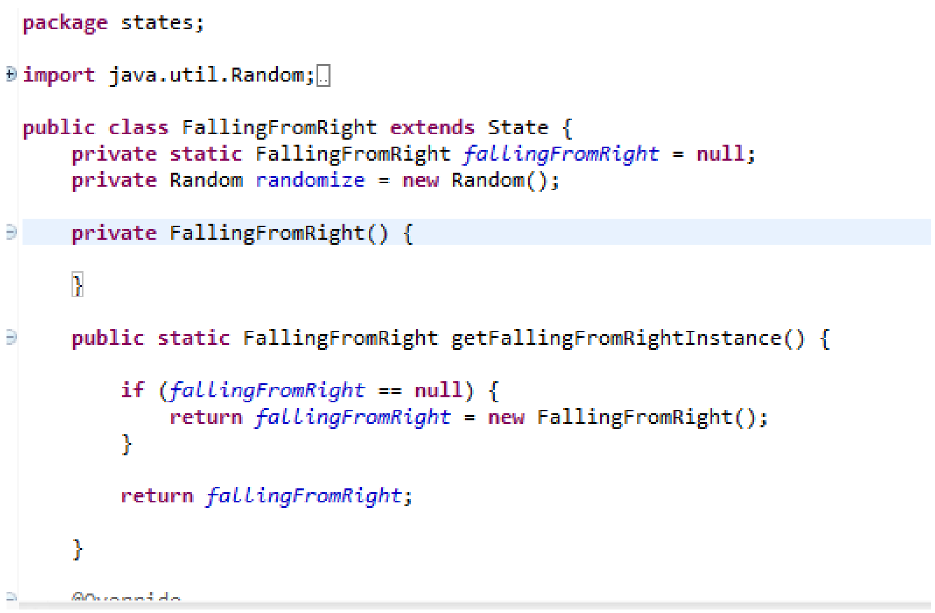
```java
package states;

import javafx.scene.canvas.GraphicsContext;

public class Caught extends State {

    private static Caught caught = null;

    private Caught() {

    }

    public static Caught getcaughtInstance() {

        if (caught == null) {
            return caught = new Caught();
        }

        return caught;
    }
```



```java
package states;

import javafx.scene.canvas.GraphicsContext;

public class Stored extends State {

    private static Stored stored = null;

    private Stored() {

    }

    public static Stored getStoredInstance() {

        if (stored == null) {
            return stored = new Stored();
        }

        return stored;
    }

    @Override
    public boolean isCaught() {
```

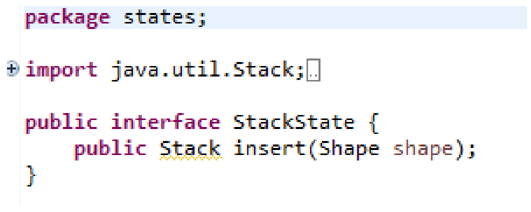


```java
package states;

import java.util.Random;

public class FallingFromLeft extends State {
    private static FallingFromLeft fallingFromLeft = null;
    private Random randomize = new Random();

    private FallingFromLeft() {

    }

    public static FallingFromLeft getFallingFromLeftInstance() {

        if (fallingFromLeft == null) {
            return fallingFromLeft = new FallingFromLeft();
        }

        return fallingFromLeft;
    }

    @Override
```

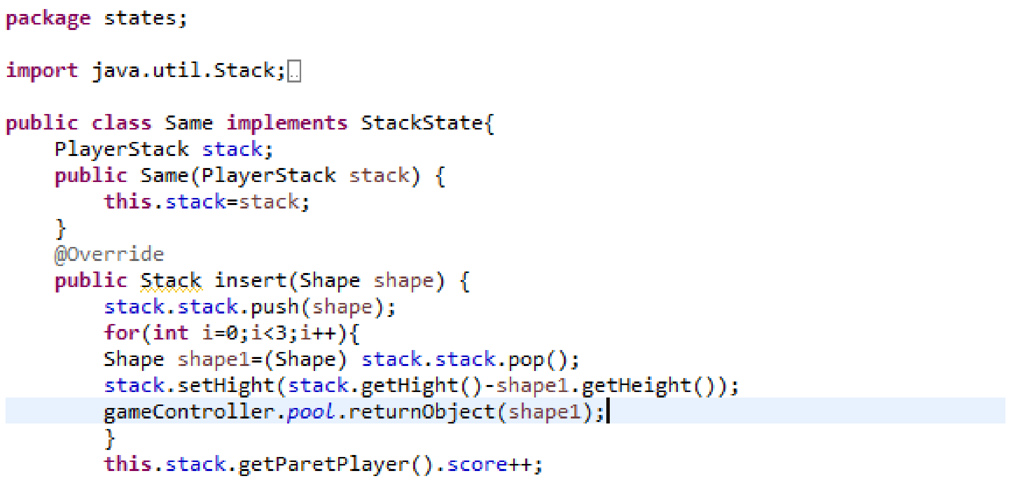


```java
package states;

import java.util.Random;

public class FallingFromRight extends State {
    private static FallingFromRight fallingFromRight = null;
    private Random randomize = new Random();

    private FallingFromRight() {

    }

    public static FallingFromRight getFallingFromRightInstance() {

        if (fallingFromRight == null) {
            return fallingFromRight = new FallingFromRight();
        }

        return fallingFromRight;

    }
```

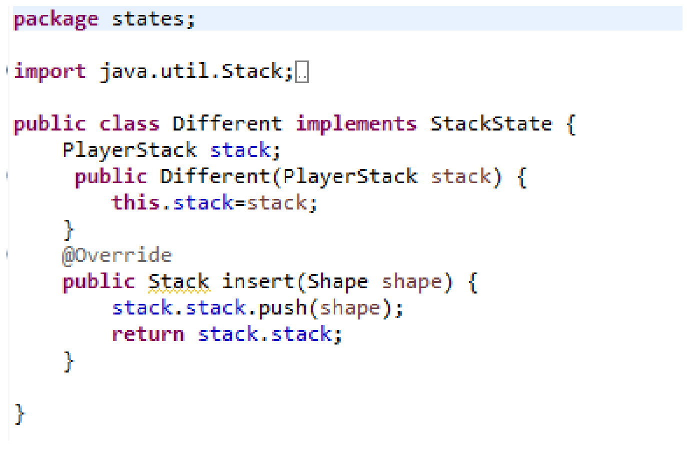
2- The stacks that each player has have two states whenever the plates is captured by them and that is different if the plates are not similar in color and same if three plates of the same color are captured and that's when these plates are removed and are sent back to the pool



```java
package states;

import java.util.Stack;

public interface StackState {
    public Stack insert(Shape shape);
}
```



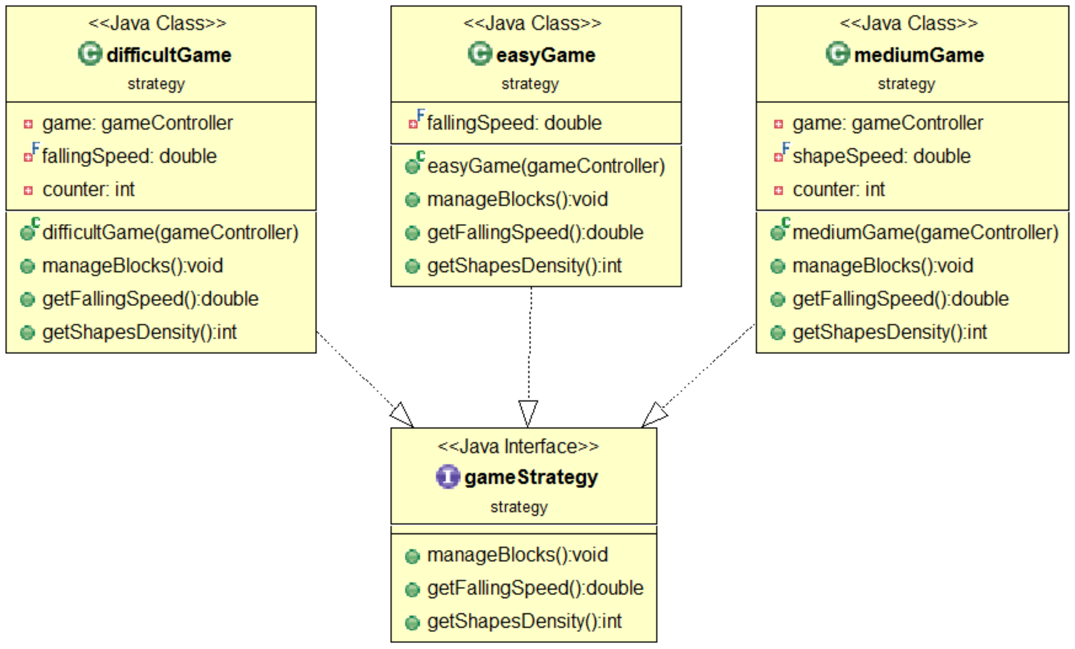
```java
package states;

import java.util.Stack;

public class Same implements StackState{
    PlayerStack stack;
    public Same(PlayerStack stack) {
        this.stack=stack;
    }
    @Override
    public Stack insert(Shape shape) {
        stack.stack.push(shape);
        for(int i=0;i<3;i++){
        Shape shape1=(Shape) stack.stack.pop();
        stack.setHight(stack.getHight()-shape1.getHeight());
        gameController.pool.returnObject(shape1);
        }
        this.stack.getParetPlayer().score++;
```



```
package states;

import java.util.Stack;

public class Different implements StackState {
    PlayerStack stack;
     public Different(PlayerStack stack) {
        this.stack=stack;
    }
    @Override
    public Stack insert(Shape shape) {
        stack.stack.push(shape);
        return stack.stack;
    }

}
```

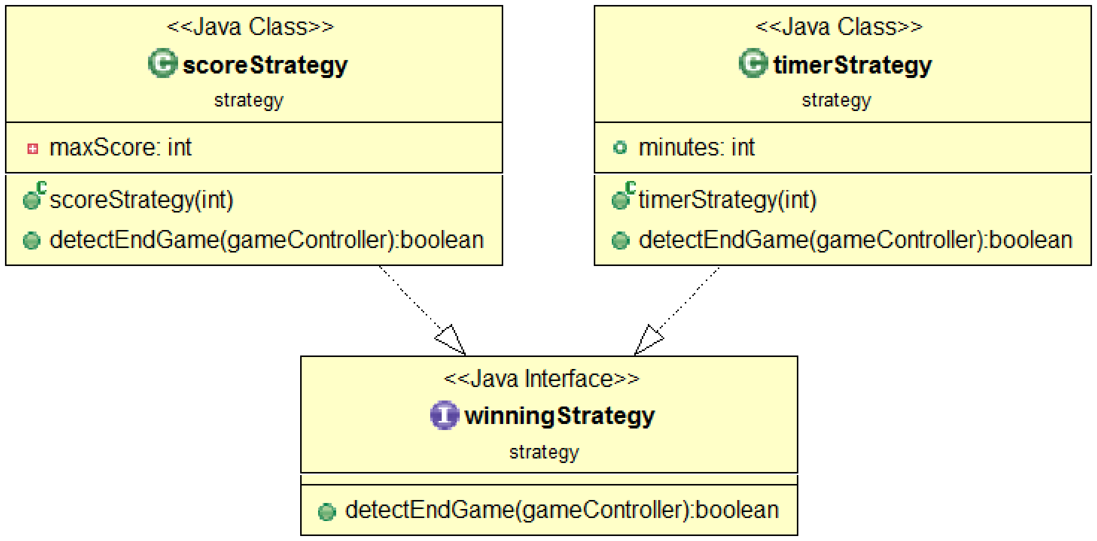
7. Strategy:

Strategy is applied on the difficulty levels of the game:
Difficulty levels have different strategies to deal with the speed of the falling shapes also to manage the density of the blocks “blocks are used to block the stack of the player so he couldn’t collect shapes anymore”. These strategies are encapsulated in three classes “easyGame” , “mediumGame” and “difficultGame”. All of them implement the same interface “gameStrategy”



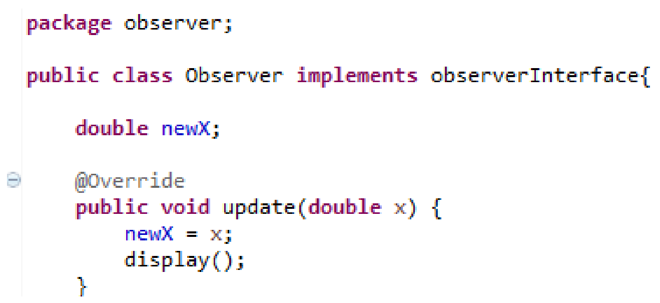
Strategy is applied on two different ways to end a game:

The game can be limited by a specific time “timerStrategy” or it can be ended by reaching one of the players to specific score ”scoreStrategy”. The user can choose between these two strategies to end the game as each logic is stored in a separate class implementing the same interface “winningStrategy”

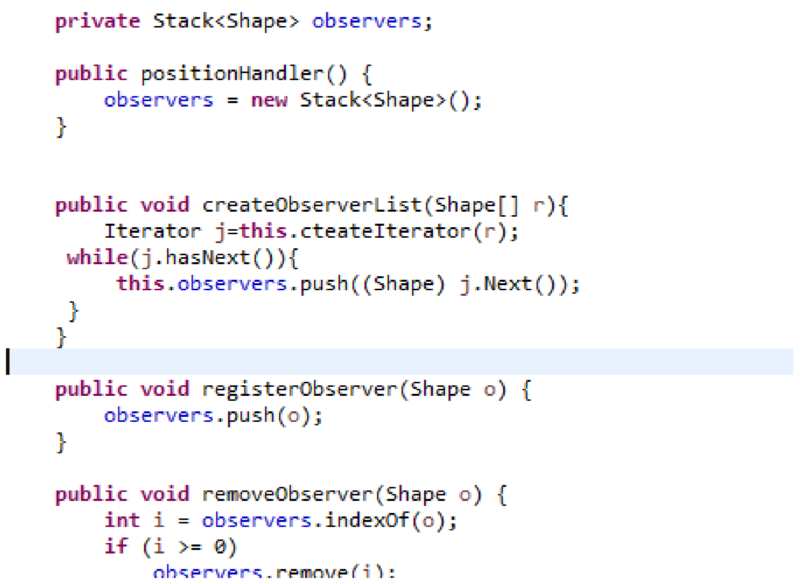


8. Observer:

The observer design pattern is used in our game to allow the player shape to notify the shapes he is holding that he started moving so they could move with his movement so in our design the shapes are the observers and the player is the observable hence each player has his own observer list



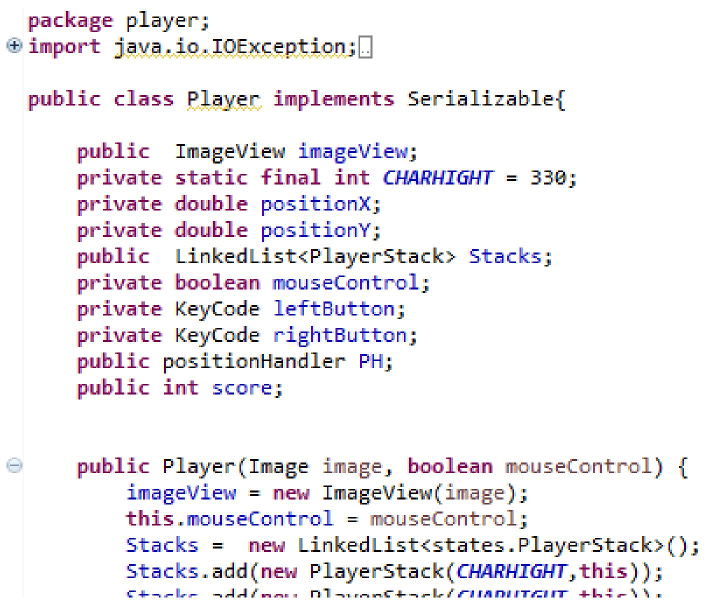
```java
package observer;

public class Observer implements observerInterface{

    double newX;

    @Override
    public void update(double x) {
        newX = x;
        display();
    }
```



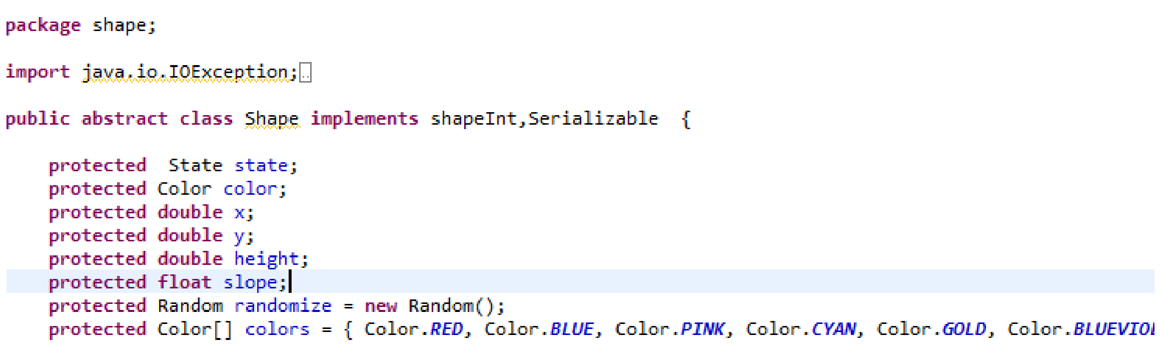
```java
private Stack<Shape> observers;

public positionHandler() {
    observers = new Stack<Shape>();
}


public void createObserverList(Shape[] r){
    Iterator j=this.cteateIterator(r);
 while(j.hasNext()){
     this.observers.push((Shape) j.Next());
 }
}

public void registerObserver(Shape o) {
    observers.push(o);
}

public void removeObserver(Shape o) {
    int i = observers.indexOf(o);
    if (i >= 0)
        observers.remove(i);
```

9. MVC

The MVC design pattern is applied in our project to separate the 3 main components of the game. Model, the objects that is used in our game like shapes and players each in separate packages:



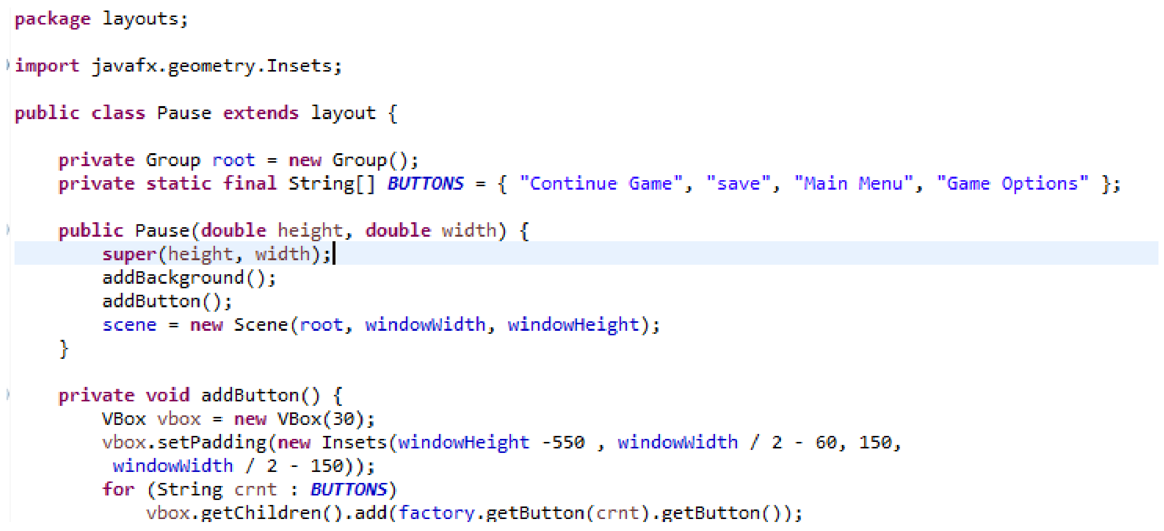
```
package player;
import java.io.IOException;

public class Player implements Serializable{

    public  ImageView imageView;
    private static final int CHARHIGHT = 330;
    private double positionX;
    private double positionY;
    public  LinkedList<PlayerStack> Stacks;
    private boolean mouseControl;
    private KeyCode leftButton;
    private KeyCode rightButton;
    public positionHandler PH;
    public int score;


    public Player(Image image, boolean mouseControl) {
        imageView = new ImageView(image);
        this.mouseControl = mouseControl;
        Stacks =  new LinkedList<states.PlayerStack>();
        Stacks.add(new PlayerStack(CHARHIGHT,this));
        Stacks.add(new PlayerStack(CHARHIGHT,this));
```



```
package shape;

import java.io.IOException;

public abstract class Shape implements shapeInt,Serializable  {

    protected  State state;
    protected Color color;
    protected double x;
    protected double y;
    protected double height;
    protected float slope;
    protected Random randomize = new Random();
    protected Color[] colors = { Color.RED, Color.BLUE, Color.PINK, Color.CYAN, Color.GOLD, Color.BLUEVIOL
```

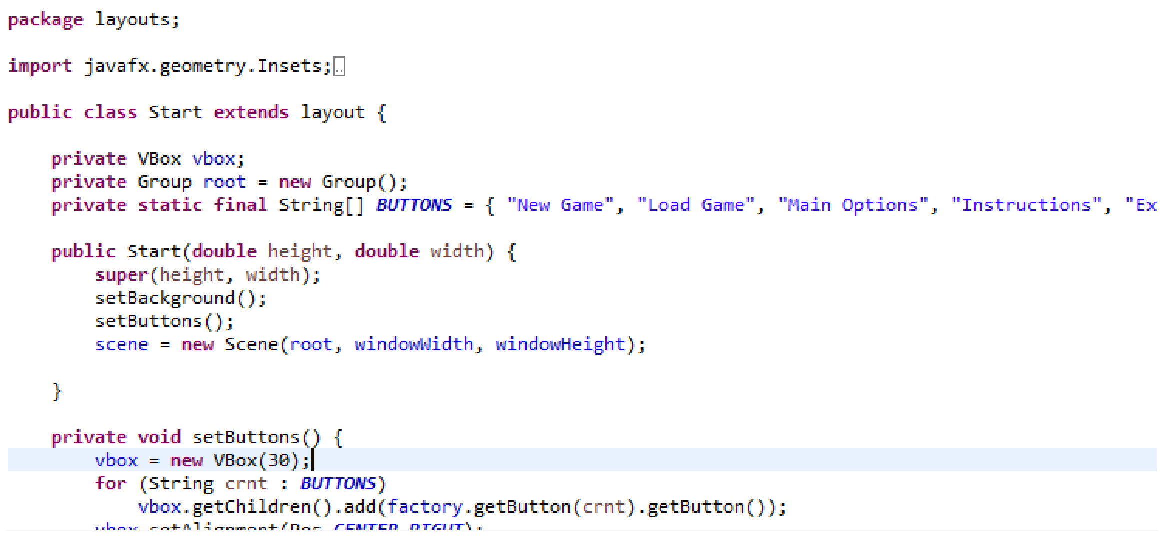
View, we put the classes that are responsible of the view of the game in general in a separate package and that is the scene package or layouts

Like the pause layout:



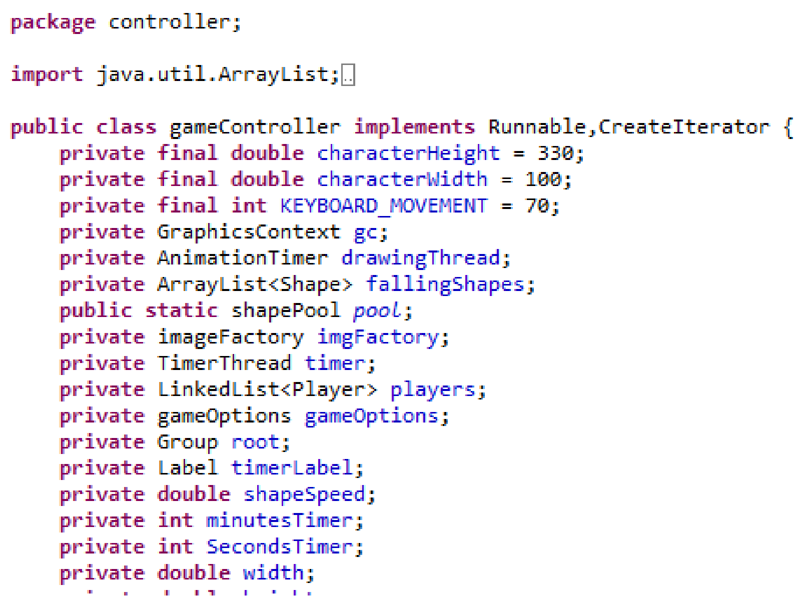
```java
package layouts;

import javafx.geometry.Insets;

public class Pause extends layout {

    private Group root = new Group();
    private static final String[] BUTTONS = { "Continue Game", "save", "Main Menu", "Game Options" };

    public Pause(double height, double width) {
        super(height, width);
        addBackground();
        addButton();
        scene = new Scene(root, windowWidth, windowHeight);
    }

    private void addButton() {
        VBox vbox = new VBox(30);
        vbox.setPadding(new Insets(windowHeight -550 , windowWidth / 2 - 60, 150,
         windowWidth / 2 - 150));
        for (String crnt : BUTTONS)
            vbox.getChildren().add(factory.getButton(crnt).getButton());
```

And the start layout:

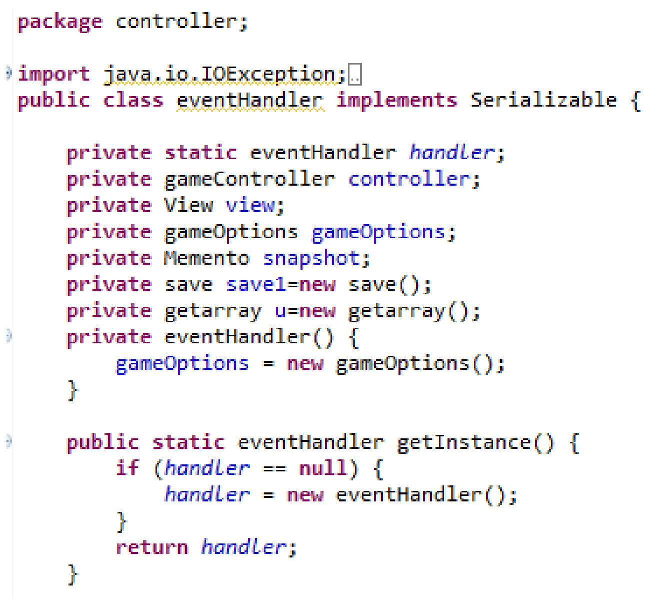


```java
package layouts;

import javafx.geometry.Insets;

public class Start extends layout {

    private VBox vbox;
    private Group root = new Group();
    private static final String[] BUTTONS = { "New Game", "Load Game", "Main Options", "Instructions", "Ex

    public Start(double height, double width) {
        super(height, width);
        setBackground();
        setButtons();
        scene = new Scene(root, windowWidth, windowHeight);

    }

    private void setButtons() {
        vbox = new VBox(30);
        for (String crnt : BUTTONS)
            vbox.getChildren().add(factory.getButton(crnt).getButton());
        vbox.setAlignment(Pos.CENTER_RIGHT);
```

and the controller that is responsible of the interact between the scene and models or the movement and controlling in general



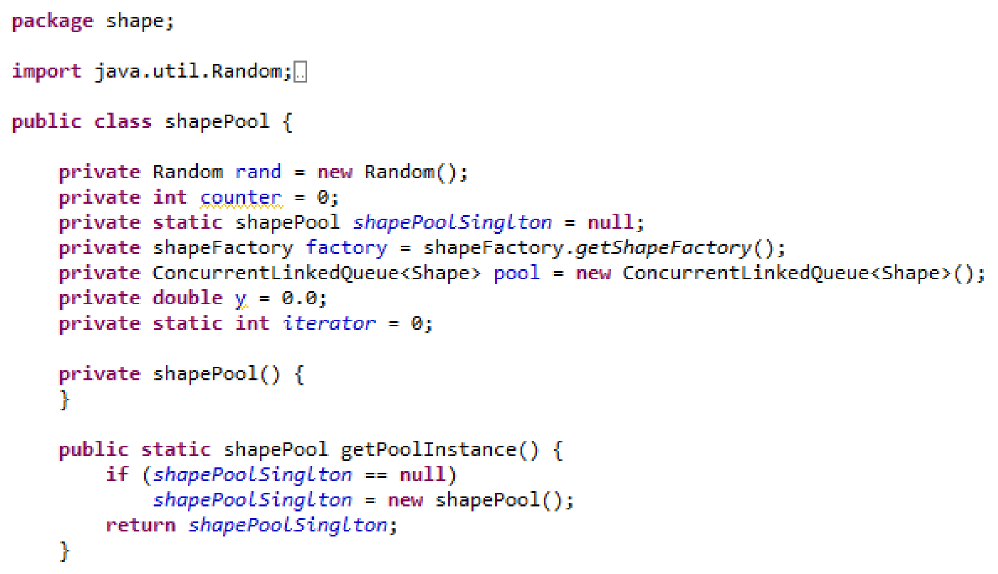
```
package controller;

import java.util.ArrayList;

public class gameController implements Runnable,CreateIterator {
    private final double characterHeight = 330;
    private final double characterWidth = 100;
    private final int KEYBOARD_MOVEMENT = 70;
    private GraphicsContext gc;
    private AnimationTimer drawingThread;
    private ArrayList<Shape> fallingShapes;
    public static shapePool pool;
    private imageFactory imgFactory;
    private TimerThread timer;
    private LinkedList<Player> players;
    private gameOptions gameOptions;
    private Group root;
    private Label timerLabel;
    private double shapeSpeed;
    private int minutesTimer;
    private int SecondsTimer;
    private double width;
```



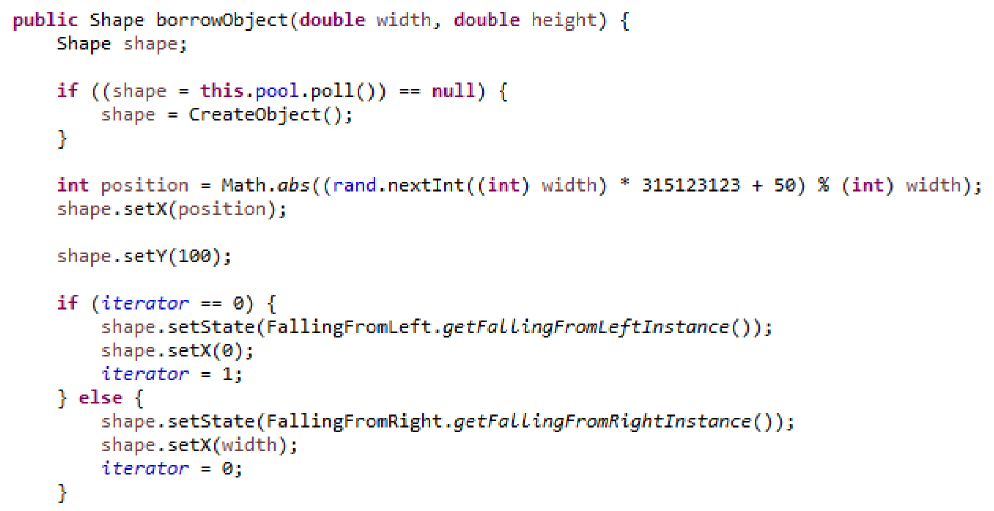
```
package controller;

import java.io.IOException;
public class eventHandler implements Serializable {

    private static eventHandler handler;
    private gameController controller;
    private View view;
    private gameOptions gameOptions;
    private Memento snapshot;
    private save save1=new save();
    private getarray u=new getarray();
    private eventHandler() {
        gameOptions = new gameOptions();
    }

    public static eventHandler getInstance() {
        if (handler == null) {
            handler = new eventHandler();
        }
        return handler;
    }
```

10. Object Pool:

We used the object pool design pattern in our game to create a pool of shapes with different random colors typically when the pool is empty a new shape is created and when the player collect three shapes of the same color the shapes disappear from the player's stack and return to the pool



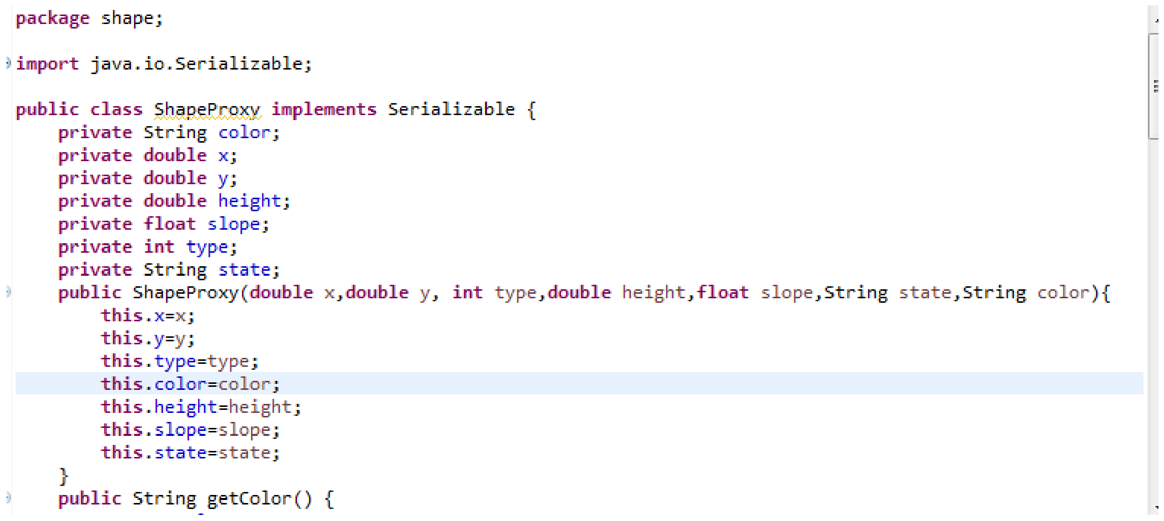
```java
package shape;

import java.util.Random;

public class shapePool {

    private Random rand = new Random();
    private int counter = 0;
    private static shapePool shapePoolSinglton = null;
    private shapeFactory factory = shapeFactory.getShapeFactory();
    private ConcurrentLinkedQueue<Shape> pool = new ConcurrentLinkedQueue<Shape>();
    private double y = 0.0;
    private static int iterator = 0;

    private shapePool() {
    }

    public static shapePool getPoolInstance() {
        if (shapePoolSinglton == null)
            shapePoolSinglton = new shapePool();
        return shapePoolSinglton;
    }
```



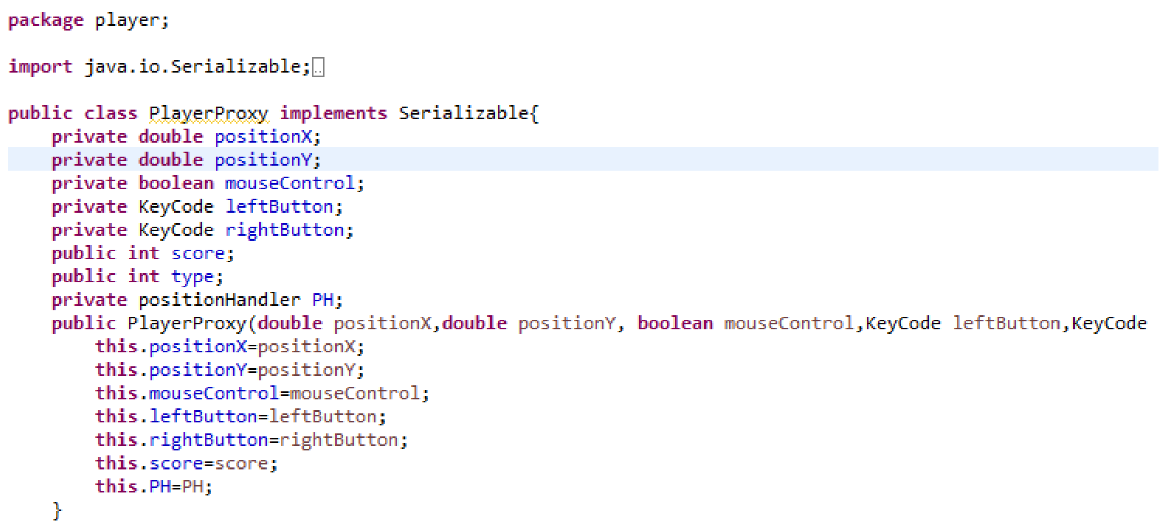
```java
public Shape borrowObject(double width, double height) {
    Shape shape;

    if ((shape = this.pool.poll()) == null) {
        shape = CreateObject();
    }

    int position = Math.abs((rand.nextInt((int) width) * 315123123 + 50) % (int) width);
    shape.setX(position);

    shape.setY(100);

    if (iterator == 0) {
        shape.setState(FallingFromLeft.getFallingFromLeftInstance());
        shape.setX(0);
        iterator = 1;
    } else {
        shape.setState(FallingFromRight.getFallingFromRightInstance());
        shape.setX(width);
        iterator = 0;
    }
```

11. Proxy:

We used the proxy design pattern to make an image object of both Shape and Player without taking some fields from both of the classes we used these proxy objects in saving and loading the objects we would save the proxy object then transform it to the real object because some of the fields in the real classes are not serializable hence we didn’ t make them present in the saving and loading processes

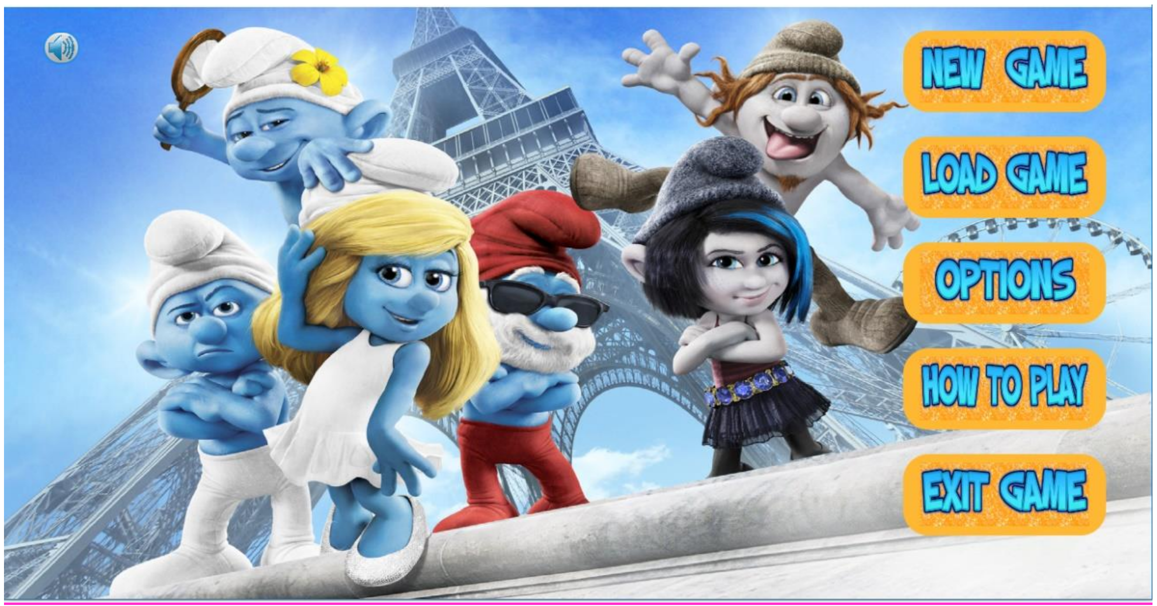


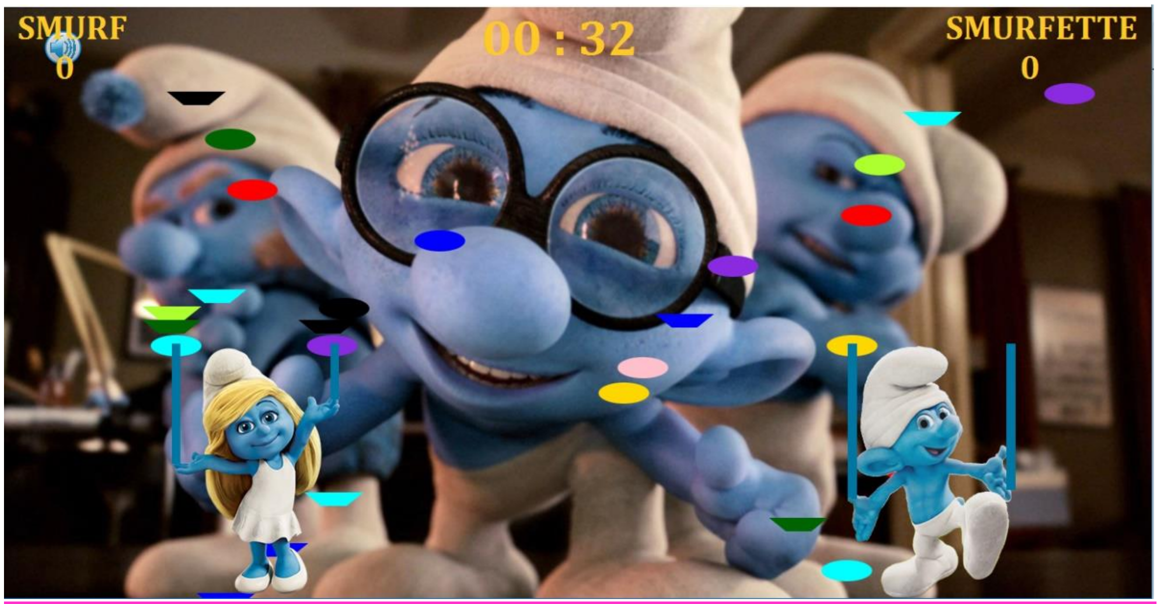
```
package shape;

import java.io.Serializable;

public class ShapeProxy implements Serializable {
    private String color;
    private double x;
    private double y;
    private double height;
    private float slope;
    private int type;
    private String state;
    public ShapeProxy(double x,double y, int type,double height,float slope,String state,String color){
        this.x=x;
        this.y=y;
        this.type=type;
        this.color=color;
        this.height=height;
        this.slope=slope;
        this.state=state;
    }
    public String getColor() {
```

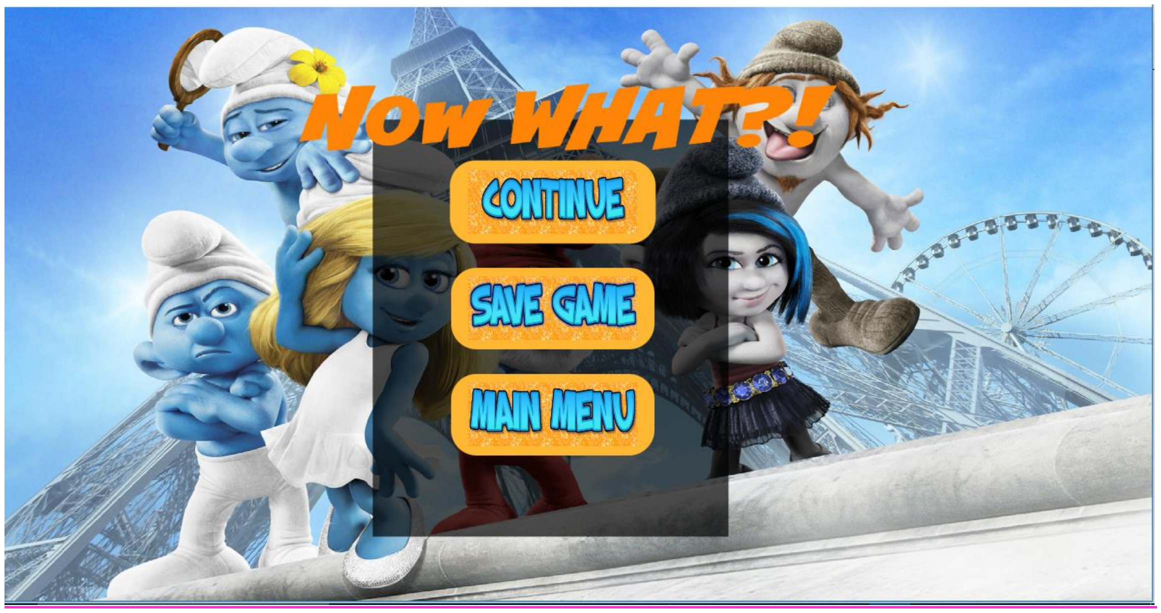


```
package player;

import java.io.Serializable;

public class PlayerProxy implements Serializable{
    private double positionX;
    private double positionY;
    private boolean mouseControl;
    private KeyCode leftButton;
    private KeyCode rightButton;
    public int score;
    public int type;
    private positionHandler PH;
    public PlayerProxy(double positionX,double positionY, boolean mouseControl,KeyCode leftButton,KeyCode
        this.positionX=positionX;
        this.positionY=positionY;
        this.mouseControl=mouseControl;
        this.leftButton=leftButton;
        this.rightButton=rightButton;
        this.score=score;
        this.PH=PH;
    }
```

6) <u>GUI Snapshots:</u>

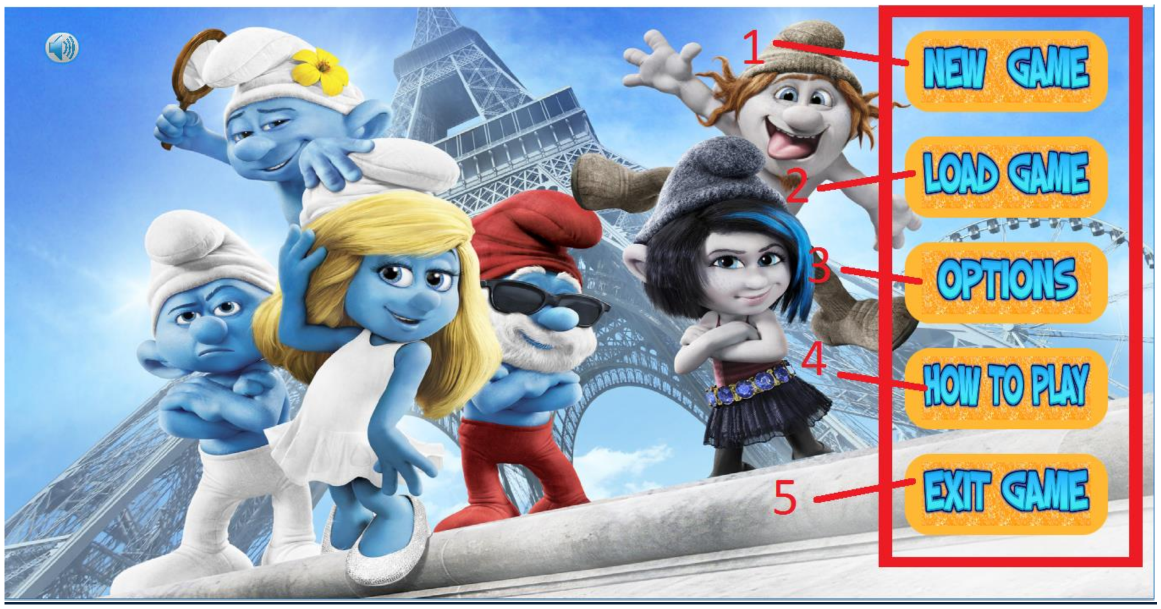

NEW GAME
LOAD GAME
OPTIONS
HOW TO PLAY
EXIT GAME


SMURF
0
00 : 32
SMURFETTE
0



7) User Guide:

multi-player game where one uses the mouse, the other uses the keyboard.



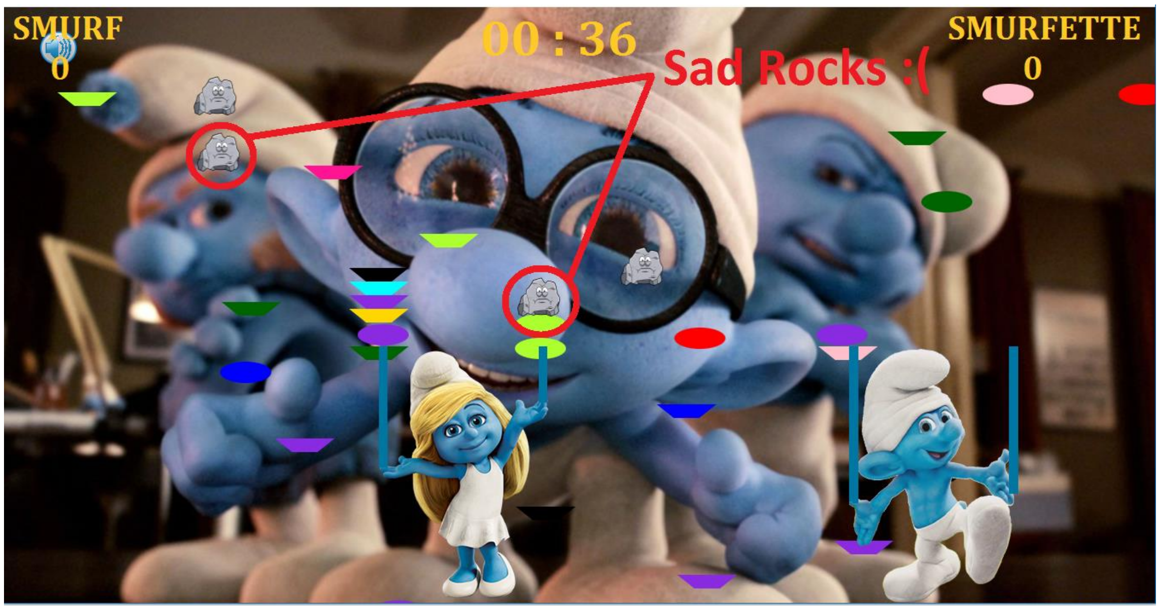
When you first open the game, you see the Main menu (in red rectangle) with lots of options – click on the button to choose:

1- New Game: to start a new game with the chosen level.

2- Load Game: to load a pre saved game.
3- Options: where you can choose

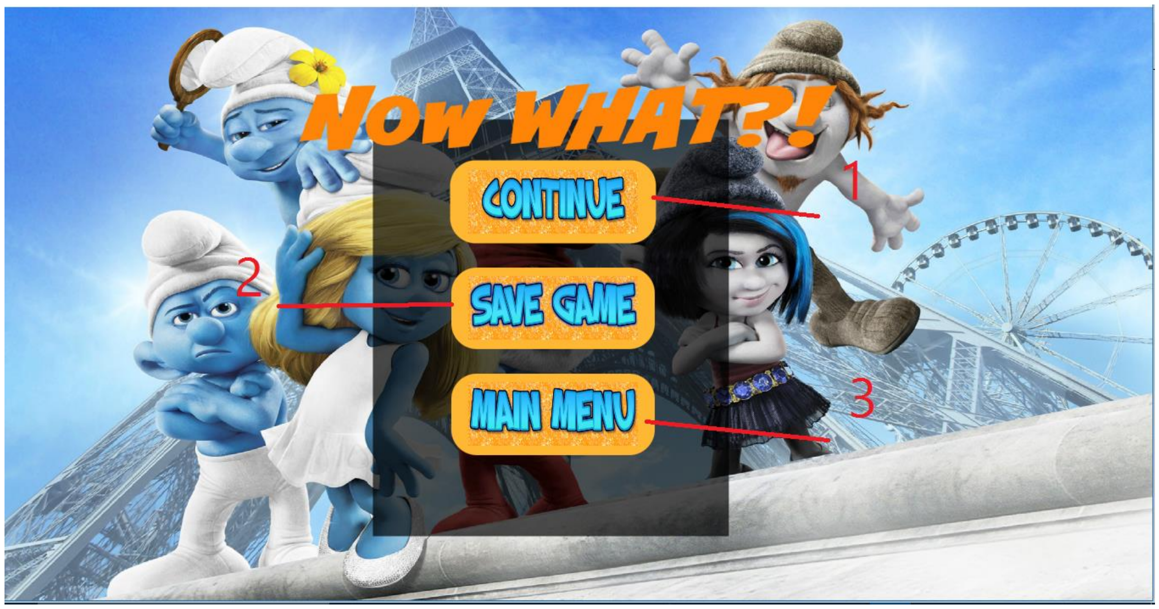
- The difficulty level.
- The control keys
- The winning strategy (time / score)

4- How To Play: for simple instruction on how the game goes and the way to win.
5- Exit Game: to close the game to play later.



You start the game and as you increase the difficulty the speed of the shapes increases and the SAD ROCKS appear more frequently!!
Your goal is to collect 3 shapes with the same color in row and avoid catching the sad rocks as they can make you sad and make you unable to collect shapes with your stacks.
While you are playing appears the mute button on the upper left corner of the window in case you want to mute / unmute the background music.

Anytime during your game you can press Escape from your keyboard to pause the game and go to the following screen.



Where you can choose:

1- Continue: back to game.
2- Save Game: to save the current game state to load later.
3- Main Menu: Go to main menu.

After the end of the game the winner is declared and you can head back to the Main Menu screen to start play again.