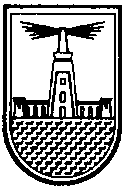
|  |
| --- |
| The Smurfs Game |
|  |
| Team Members:   * Aya Abouzeid 2 * Aya Fouad 3 * Salma Yehia 35 * Nada Ayman 79 |
|  |

******

***Alexandria University***

***Faculty of Engineering***

***Computer and Systems Engineering Dept.***

1. Introduction:
2. 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 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**.

1. Report Overview:
2. 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.

1. 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.

1. Sequence Diagrams:

An [interaction diagram](https://en.wikipedia.org/wiki/Interaction_diagram) that shows how objects operate with one another and in what order. It is a construct of a [message sequence chart](https://en.wikipedia.org/wiki/Message_sequence_chart). It shows object interactions arranged in time sequence.

1. 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.

1. GUI Snapshots:

Shows the GUI of the game with different snapshots

1. User Guide:

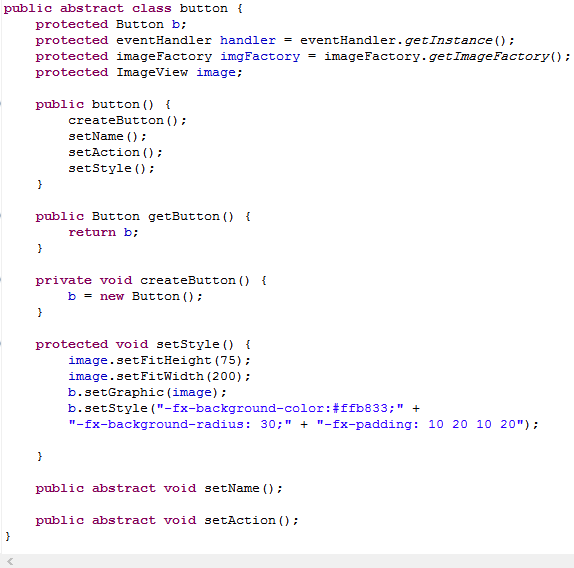
Shows the user how to use the project / game and the way to win.

1. Software Design:

The project is divided into many packages

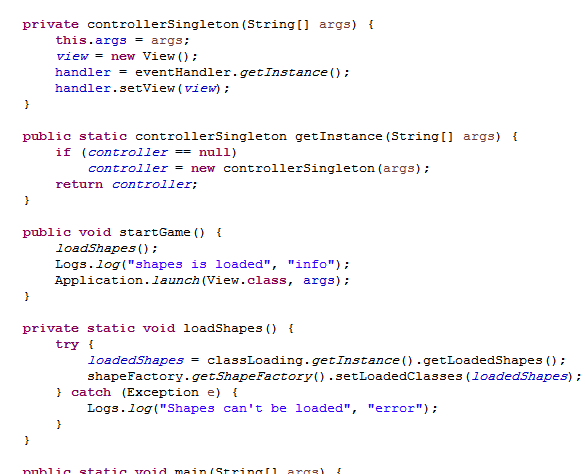
* **Button**

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:



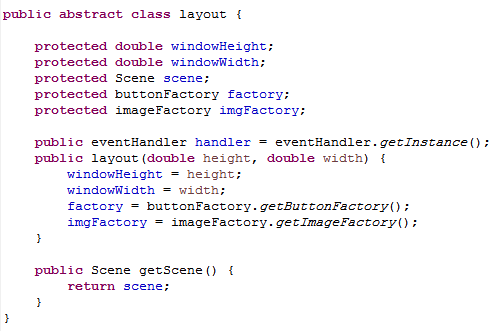
* **Controller**

The classes that controls the game   
ControllerSingleton: Has the “main” method and initializes everything as the game starts.



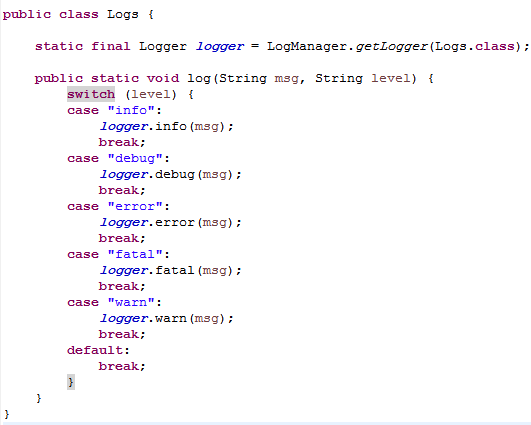
eventHandler - game Controller - Game Options music player TimerThread)

* **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).



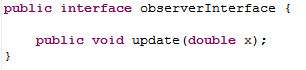
* **Logs**

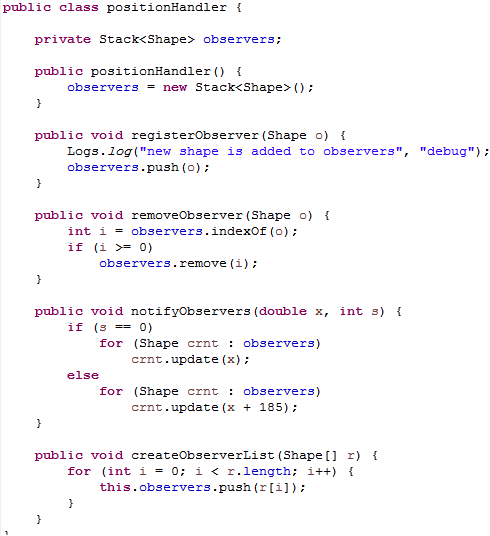
Contains the class responsible for writing the logs to the logging.log file.



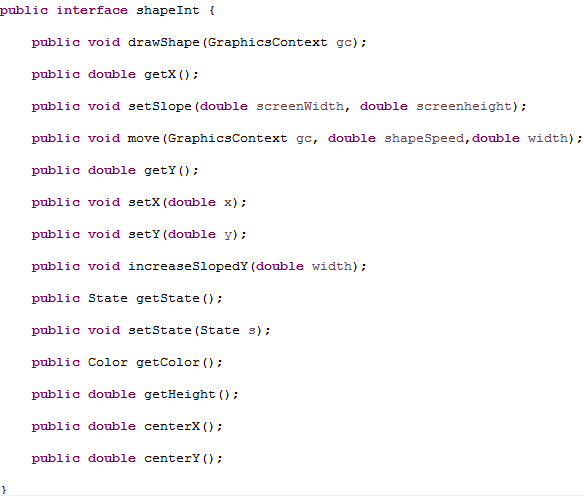
* **Observer**

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

****

****

* **Player -------------------------------------------------------------------------------**
* **Save :**
* **Shape: contains the shape Interface that all dynamically loaded shapes implement and the shape abstract class – shapeProxy – ShapePool – BuilderSahpe – BuilderShapeProxy**

****

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

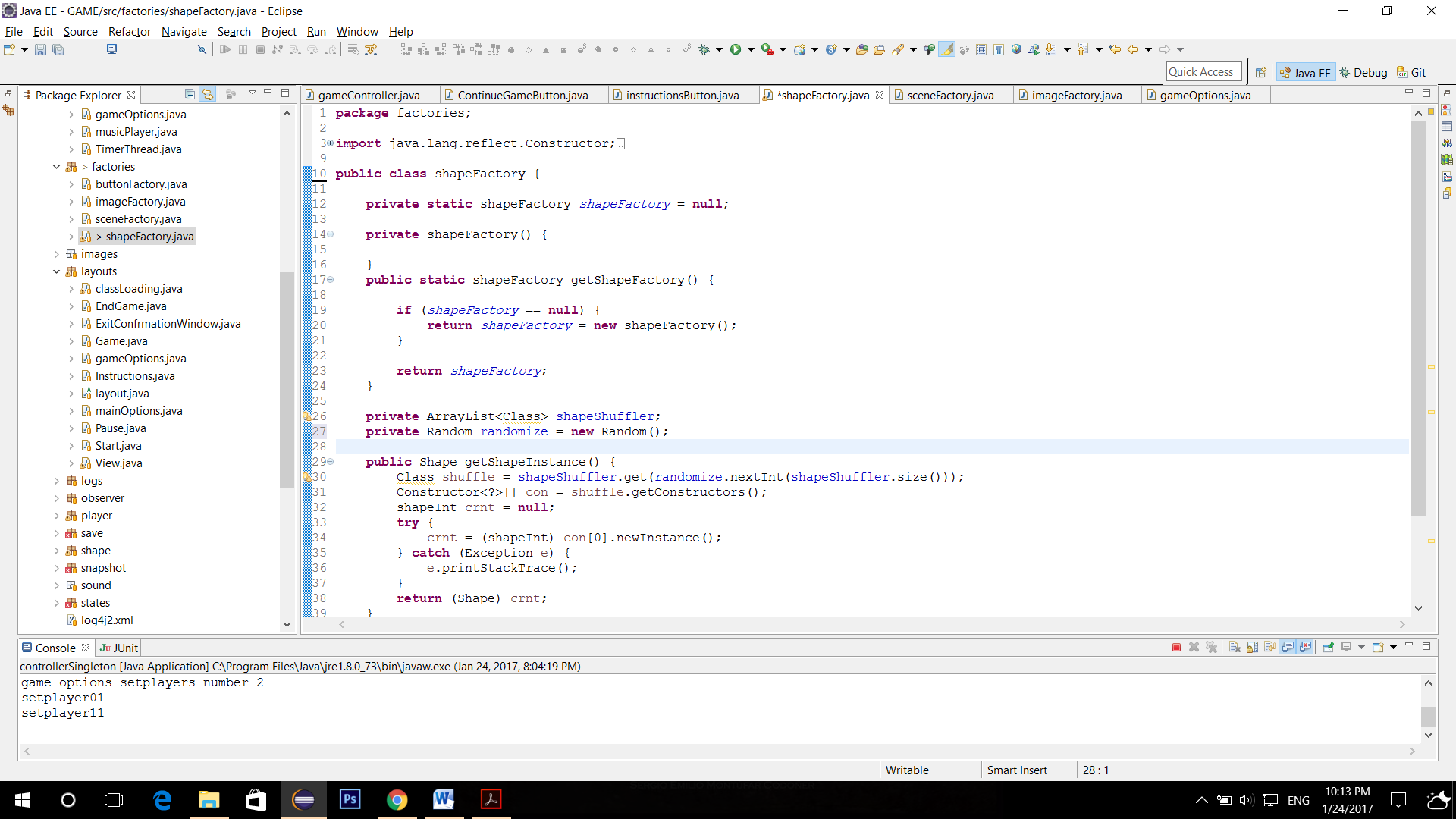
1. Class Diagrams:
2. Sequence Diagram:
3. Design patterns:

Singleton:

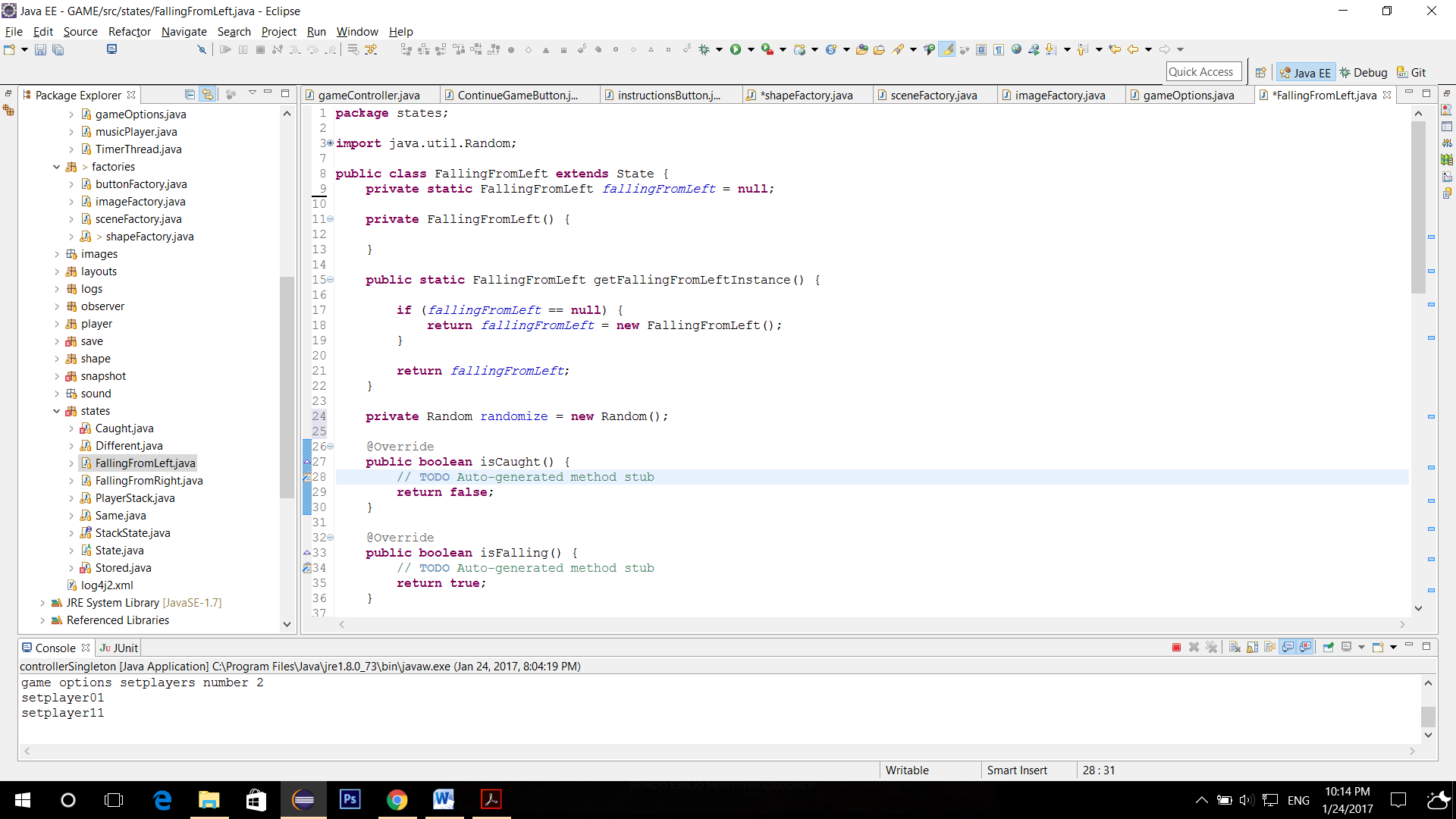
Singleton design Pattern is used in our project when a restriction was to be made on a class to limit it’s instantiation to only one object and provide global access to that instance, this is typically done by declaring the class’s constructors to be private and providing a private static instance and 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.

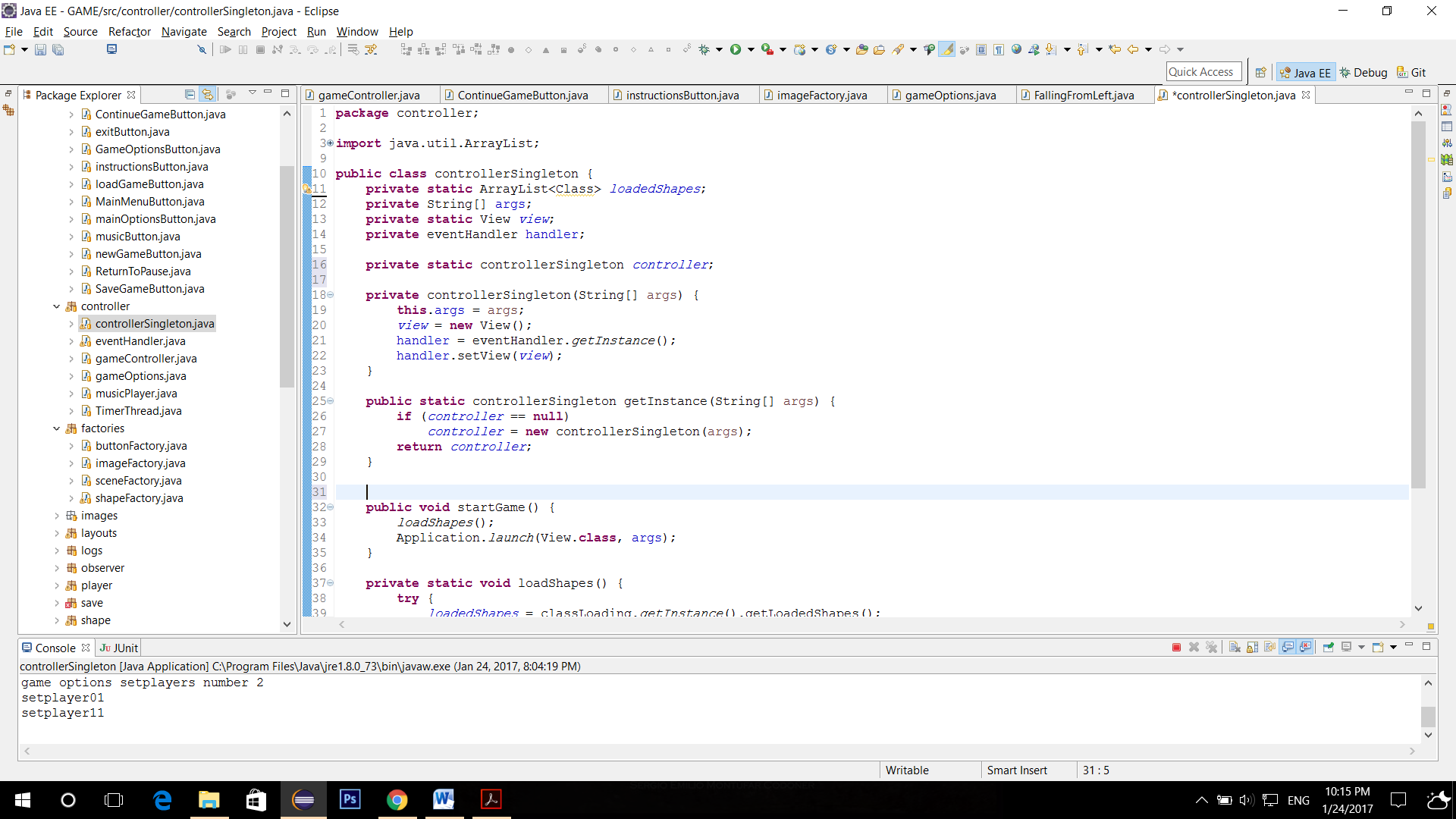
**A. Singleton applied on Factory Design Pattern:**



**B. Singleton applied on State Design Pattern:**



**C. Singleton applied on The Game Controller :**



Factory:

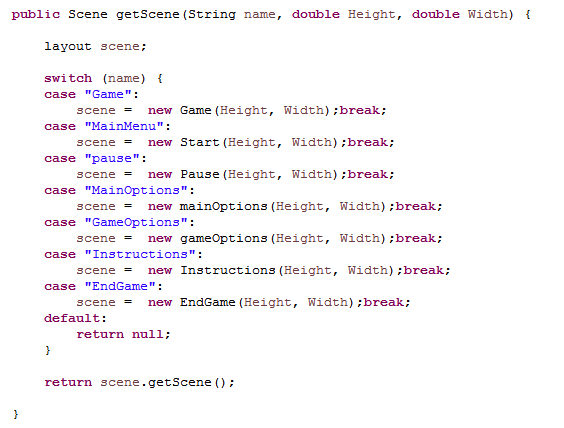
Factory Design Pattern is used in our project to create objects on demand without specifying the required class to create this object, where the factory handles this operation and returns the exact required object.

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

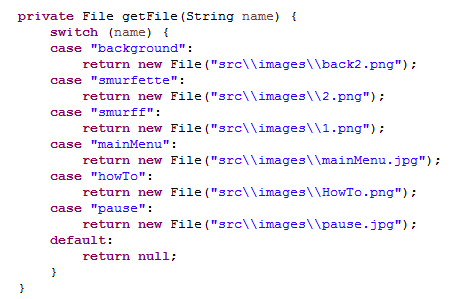
**A. Factory applied on Buttons :**



**B. Factory applied on Scenes :**

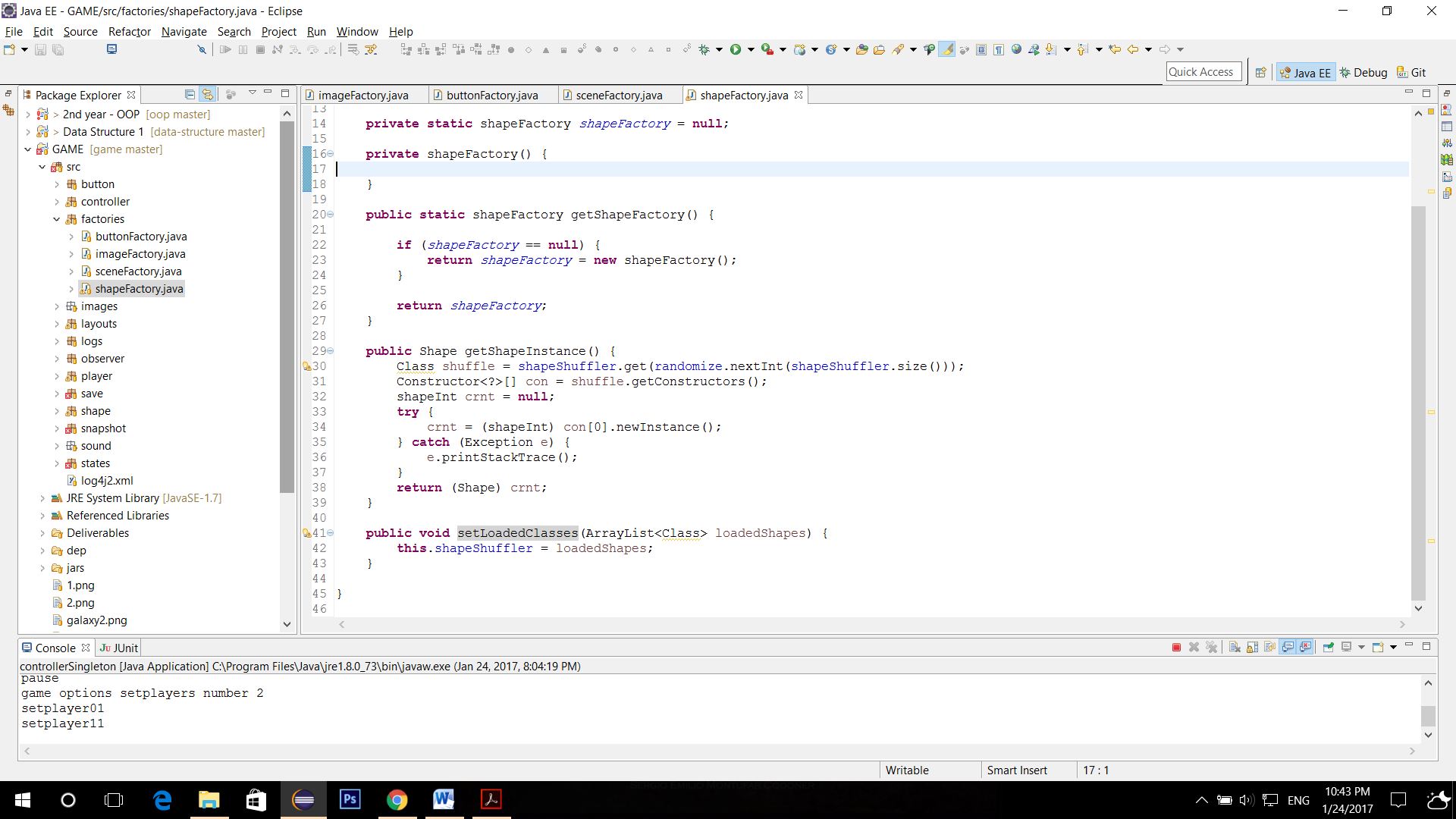


**C. Factory applied on Image Views:**



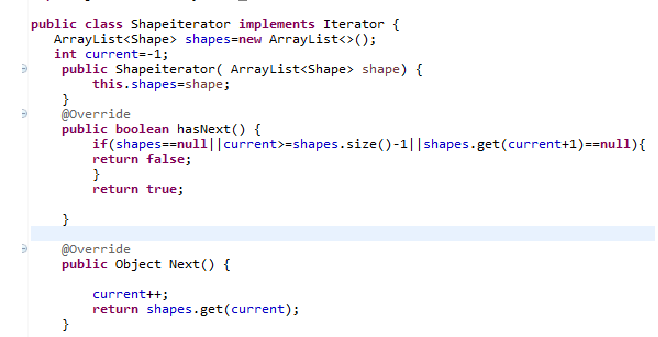
**D. Factory applied on Shapes :**

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

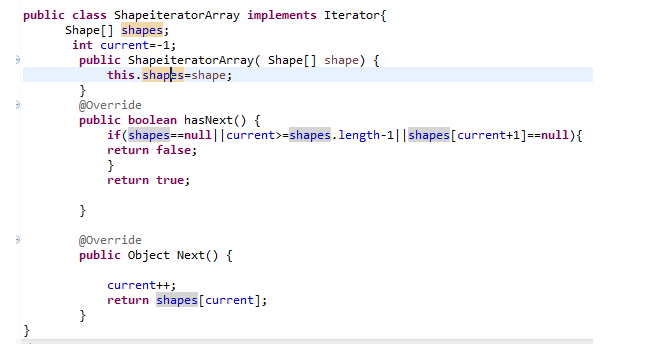


Iterator:

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



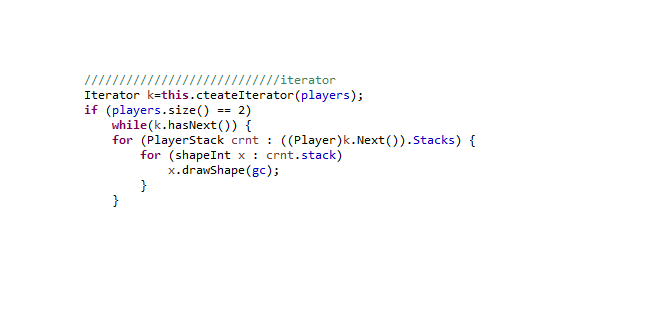




**A . in Class positionHandler**

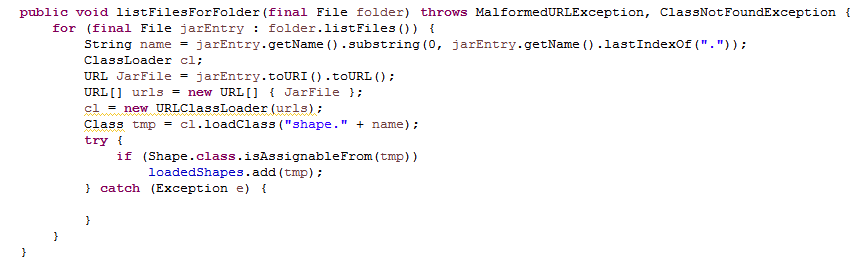


**B. in Class gameController:**



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.

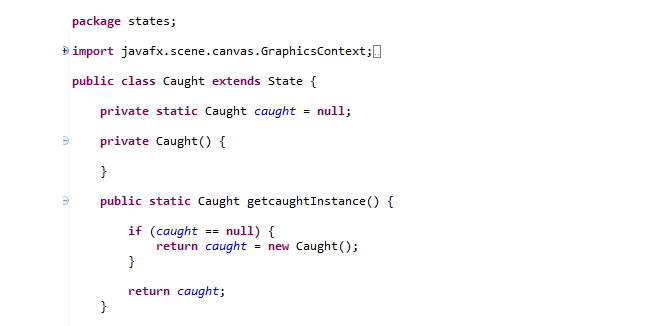


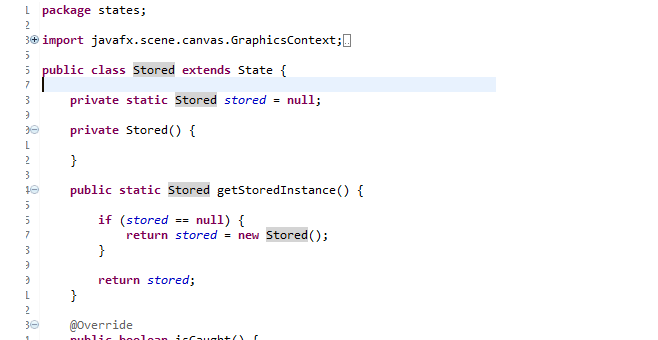
Snapshot:

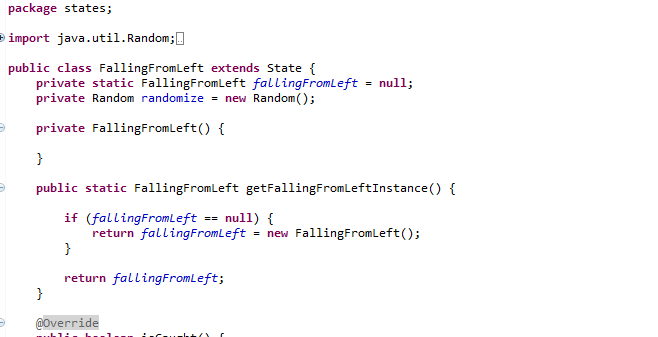
State:

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

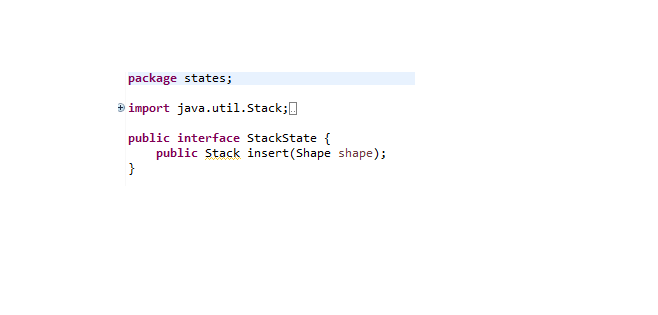


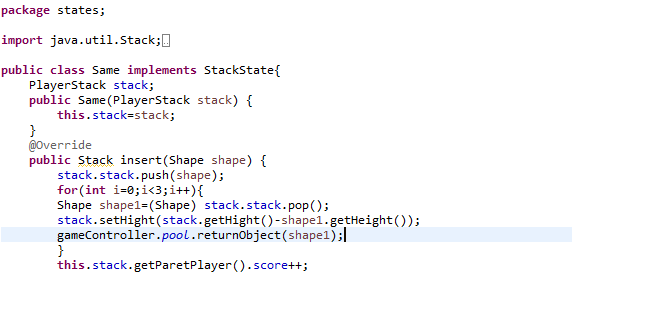


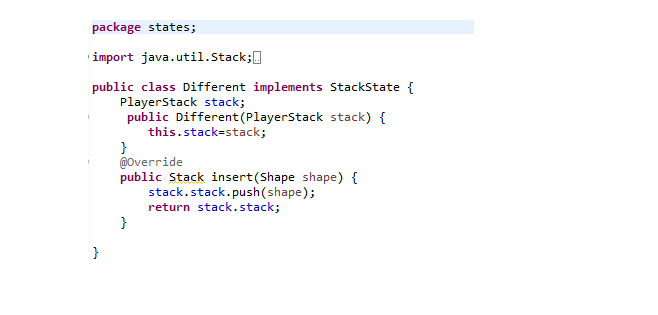




2- The stacks that each player has have two states whenever the plates is captured by them and that is different if the plates are not similar in color and same if three plates of the same color are captured and that's when these plates are removed and are sent back to the pool



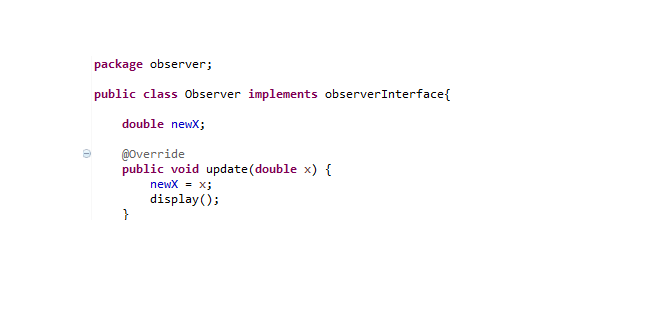


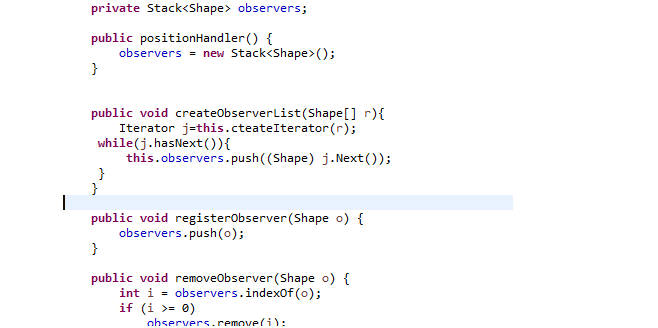


Strategy:

Observer:

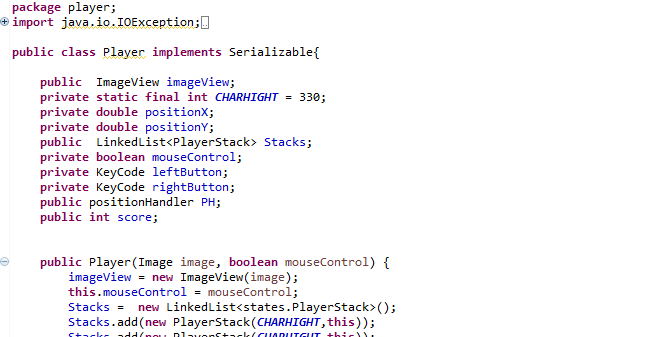
The observer design pattern is used in our game to allow the player shape to notify the shapes he is holding that he started moving so they could move with his movement so in our design the shapes are the observers and the player is the observable hence each player has his own observer list

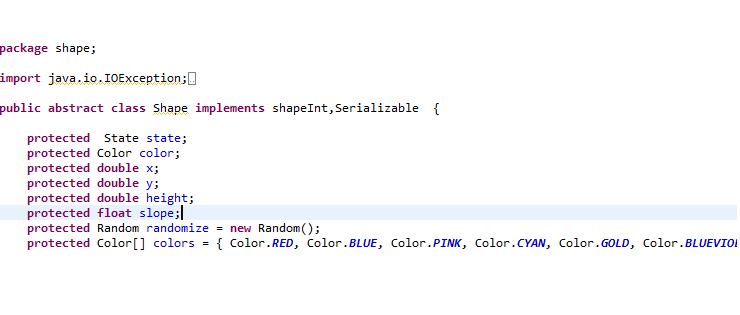




MVC

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:



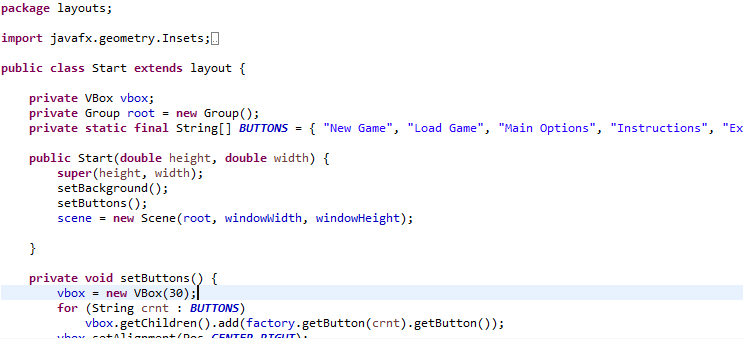


View, we put the classes that are responsible of the view of the game in general in a separate package and that is the scene package or layouts

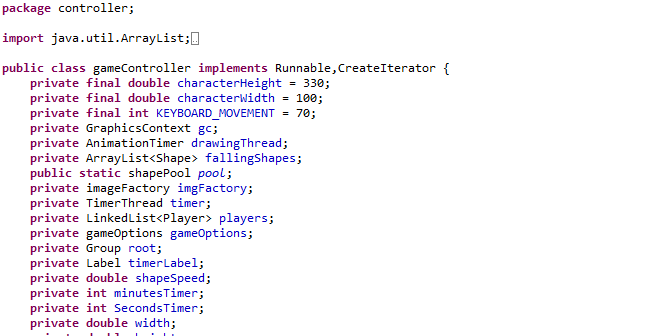
Like the pause layout:

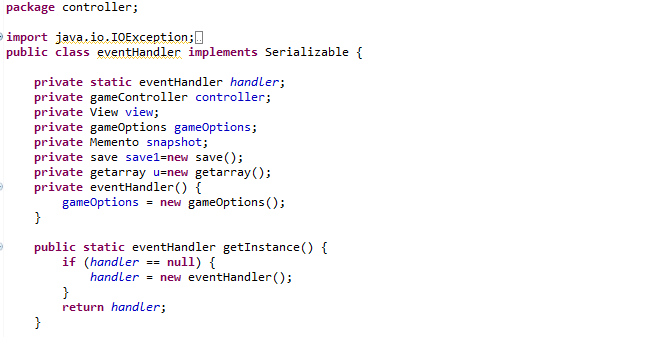


And the start layout:



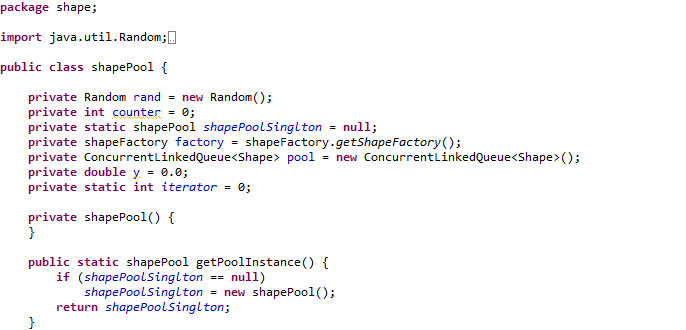
and the controller that is responsible of the interact between the scene and models or the movement and controlling in general

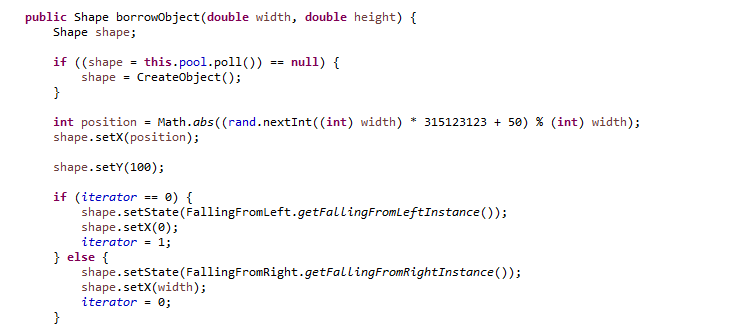




Object Pool:

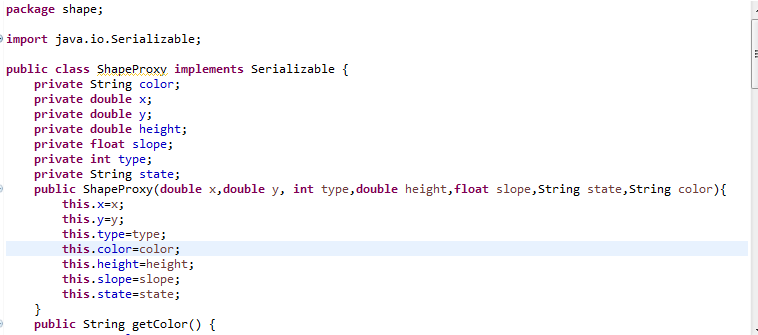
We used the object pool design pattern in our game to create a pool of shapes with different random colors typically when the pool is empty a new shape is created and when the player collect three shapes of the same color the shapes disappear from the player's stack and return to the pool

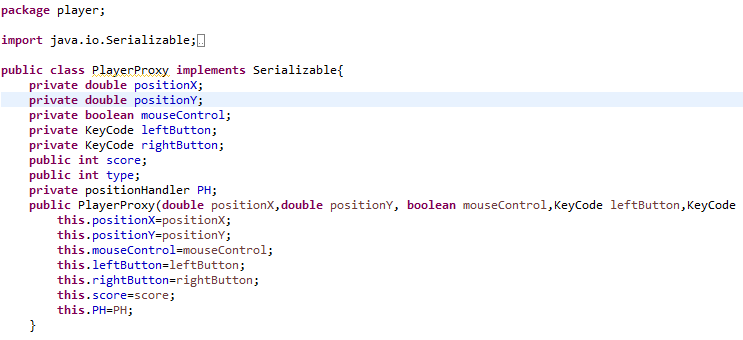




11. Proxy:

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





1. GUI Snapshots:
2. User Guide:

REQUIRED بالصلاة على النبي

**Describe your design thoroughly.**

**Class diagram of your design**. == UML ?????????????????

**Sequence diagram showing the typical scenarios of the game.**

**Section for each pattern (the required and any other patterns you used) and how you used it in your design, and a class diagram explaining this.**

**Snapshots of your GUI.**

**User guide explains how to play the game.**

**Any design decisions that you have made should be listed clearly.**

**A significant part of the grade will go to the report, so make sure you make it clean and readable**.

**Stete graphs**