Computer and Communication Engineering Program

Faculty of Engineering

Alexandria University

CSE 226 Programming - II

Circus of Plates Game Project

Name	ID
Omar Ibrahim Elsayed	7442
Yasmine Hany Elsayed	7609
Mazen Nagy Mansour	7475
Ahmed Alaa Eldin Hegazy	7386

1. Introduction

In this project it is required to design and build a single-player game in which a clown has two stack of plates (generally collected shapes) and there is a random group of colored shapes that falls and the player tries to catch those shapes using the clown stacks (hands). The player gains a score unit (10 points) if he managed to collect three consecutive shapes by the same stack having the same color regardless to their shapes and thus these three shapes vanishes immediately.

The game is made out of three levels (Easy, Amateur and Hard) and the difficulty is increasing gradually among them through increasing number of falling items, speed of movement of falling items and introducing bombs. The player losses immediately if he managed to collect more than 20 plates without forming any set of three shapes of same color (size of one stack exceeded 20 plates) or if he collected a bomb

2. Project Description

The design of the assigned game contains 8 types of design patterns implemented within our code for the game as follow:

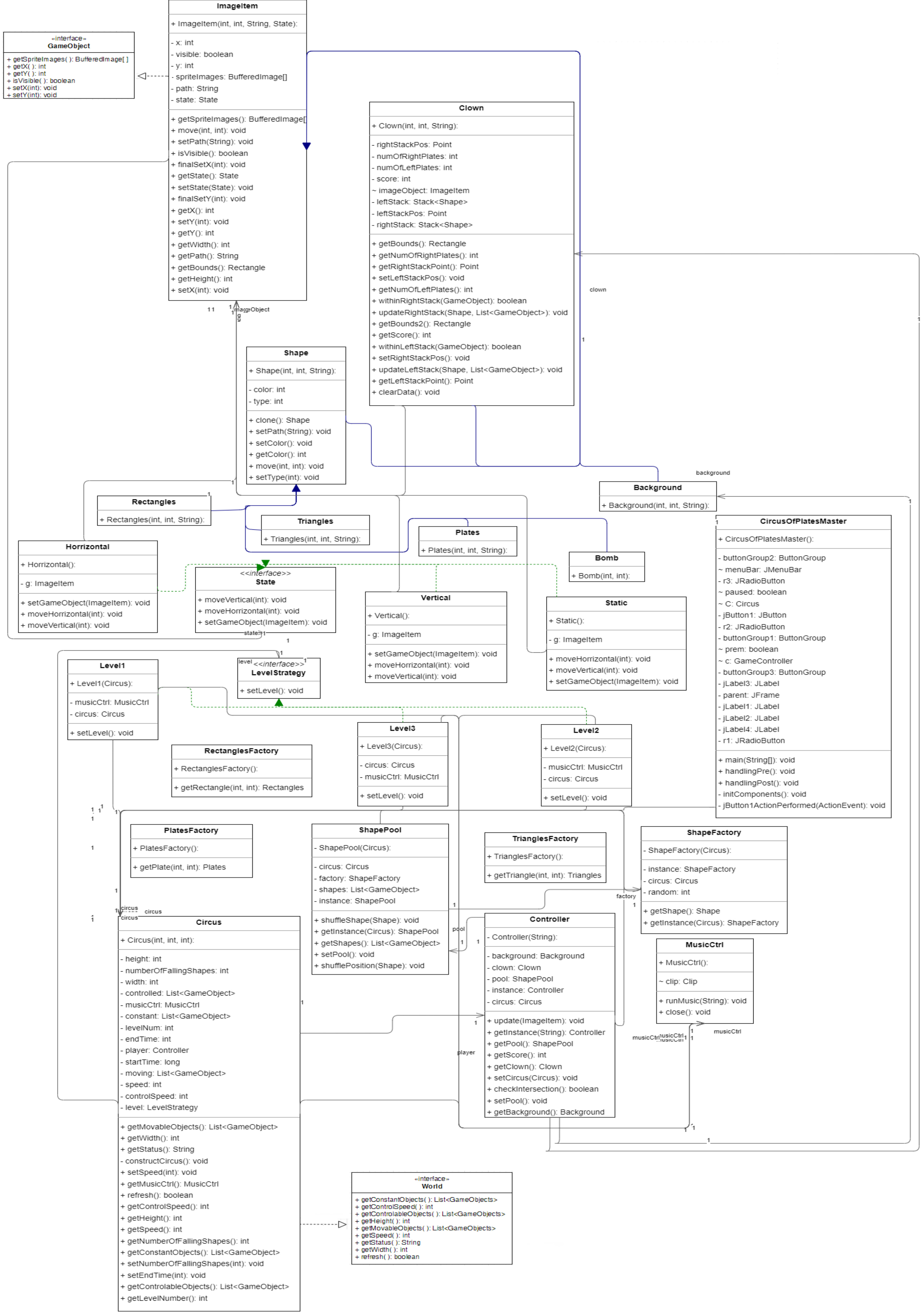
1. Singleton
2. Factory
3. Abstract Factory
4. State
5. Strategy
6. Pool
7. Iterator (not implemented manually but using its built in structure in the java programming language)
8. MVC (already implemented as the given engine handles the view and controller approach and we are required to implement the model sector. Also, we divided our classes into three packages to deeply implement it)

Our game consists of 3 shapes which are: Plates, Rectangles and Triangles. Each shape can exist in five colors which are: Blue, Magenta, Green, Black and Pink. The player is represented as a clown object with two plates (one on each hand) representing a base for the stack on each hand responsible for collecting the shapes.

Our game consists of three levels each has its own image as a background. The speed of falling of the shapes increases gradually through the levels as a method of difficulty. Moreover, in the second level a bomb is added to the list of falling objects which causes immediate losing if it is collected. In the last level, two bombs are added to the list to increase the difficulty of the game much more.

The player is able to pause the game and can resume it again easily. The player can also close the current level and choose whether to play once more after choosing the level or to exit the game. If the player loses, then he must close the current level in order to be able to replay once again. The player can choose to reset the current level played by choosing (new) from the file menu as long as the game hasn’t terminated.

2.1. UML (Class Diagram)


«interface»
GameObject
+ getSpriteImages( ): BufferedImage[ ]
+ getX( ): int
+ getY( ): int
+ isVisible( ): boolean
+ setX(int): void
+ setY(int): void
ImageItem
+ ImageItem(int, int, String, State):
- x: int
- visible: boolean
- y: int
- spriteImages: BufferedImage[]
- path: String
- state: State
+ getSpriteImages(): BufferedImage[]
+ move(int, int): void
+ setPath(String): void
+ isVisible(): boolean
+ finalSetX(int): void
+ getState(): State
+ setState(State): void
+ finalSetY(int): void
+ getX(): int
+ setY(int): void
+ getY(): int
+ getWidth(): int
+ getPath(): String
+ getBounds(): Rectangle
+ getHeight(): int
+ setX(int): void
Clown
+ Clown(int, int, String):
- rightStackPos: Point
- numOfRightPlates: int
- numOfLeftPlates: int
- score: int
~ imageObject: ImageItem
- leftStack: Stack<Shape>
- leftStackPos: Point
- rightStack: Stack<Shape>
+ getBounds(): Rectangle
+ getNumOfRightPlates(): int
+ getRightStackPoint(): Point
+ setLeftStackPos(): void
+ getNumOfLeftPlates(): int
+ withinRightStack(GameObject): boolean
+ updateRightStack(Shape, List<GameObject>): void
+ getBounds2(): Rectangle
+ getScore(): int
+ withinLeftStack(GameObject): boolean
+ setRightStackPos(): void
+ updateLeftStack(Shape, List<GameObject>): void
+ getLeftStackPoint(): Point
+ clearData(): void
clown
imageObject
Shape
+ Shape(int, int, String):
- color: int
- type: int
+ clone(): Shape
+ setPath(String): void
+ setColor(): void
+ getColor(): int
+ move(int, int): void
+ setType(int): void
background
Background
+ Background(int, int, String):
Rectangles
+ Rectangles(int, int, String):
Triangles
+ Triangles(int, int, String):
Plates
+ Plates(int, int, String):
Bomb
+ Bomb(int, int):
Horrizontal
+ Horrizontal():
- g: ImageItem
+ setGameObject(ImageItem): void
+ moveHorrizontal(int): void
+ moveVertical(int): void
<<interface>>
State
+ moveVertical(int): void
+ moveHorrizontal(int): void
+ setGameObject(ImageItem): void
state
Vertical
+ Vertical():
- g: ImageItem
+ setGameObject(ImageItem): void
+ moveHorrizontal(int): void
+ moveVertical(int): void
Static
+ Static():
- g: ImageItem
+ moveHorrizontal(int): void
+ moveVertical(int): void
+ setGameObject(ImageItem): void
CircusOfPlatesMaster
+ CircusOfPlatesMaster():
- buttonGroup2: ButtonGroup
~ menuBar: JMenuBar
- r3: JRadioButton
- paused: boolean
~ C: Circus
- jButton1: JButton
- r2: JRadioButton
- buttonGroup1: ButtonGroup
~ prem: boolean
~ c: GameController
- buttonGroup3: ButtonGroup
- jLabel3: JLabel
- parent: JFrame
- jLabel1: JLabel
- jLabel2: JLabel
- jLabel4: JLabel
- r1: JRadioButton
+ main(String[]): void
+ handlingPre(): void
+ handlingPost(): void
- initComponents(): void
- jButton1ActionPerformed(ActionEvent): void
Level1
+ Level1(Circus):
- musicCtrl: MusicCtrl
- circus: Circus
+ setLevel(): void
level
<<interface>>
LevelStrategy
+ setLevel(): void
Level3
+ Level3(Circus):
- circus: Circus
- musicCtrl: MusicCtrl
+ setLevel(): void
Level2
+ Level2(Circus):
- musicCtrl: MusicCtrl
- circus: Circus
+ setLevel(): void
RectanglesFactory
+ RectanglesFactory():
+ getRectangle(int, int): Rectangles
PlatesFactory
+ PlatesFactory():
+ getPlate(int, int): Plates
ShapePool
- ShapePool(Circus):
- circus: Circus
- factory: ShapeFactory
- shapes: List<GameObject>
- instance: ShapePool
+ shuffleShape(Shape): void
+ getInstance(Circus): ShapePool
+ getShapes(): List<GameObject>
+ setPool(): void
+ shufflePosition(Shape): void
TrianglesFactory
+ TrianglesFactory():
+ getTriangle(int, int): Triangles
ShapeFactory
- ShapeFactory(Circus):
- instance: ShapeFactory
- circus: Circus
- random: int
+ getShape(): Shape
+ getInstance(Circus): ShapeFactory
factory
pool
circus
Circus
+ Circus(int, int, int):
- height: int
- numberOfFallingShapes: int
- width: int
- controlled: List<GameObject>
- musicCtrl: MusicCtrl
- constant: List<GameObject>
- levelNum: int
- endTime: int
- player: Controller
- startTime: long
- moving: List<GameObject>
- speed: int
- controlSpeed: int
- level: LevelStrategy
+ getMovableObjects(): List<GameObject>
+ getWidth(): int
+ getStatus(): String
- constructCircus(): void
+ setSpeed(int): void
+ getMusicCtrl(): MusicCtrl
+ refresh(): boolean
+ getControlSpeed(): int
+ getHeight(): int
+ getSpeed(): int
+ getNumberOfFallingShapes(): int
+ getConstantObjects(): List<GameObject>
+ setNumberOfFallingShapes(int): void
+ setEndTime(int): void
+ getControlableObjects(): List<GameObject>
+ getLevelNumber(): int
Controller
- Controller(String):
- background: Background
- clown: Clown
- pool: ShapePool
- instance: Controller
- circus: Circus
+ update(ImageItem): void
+ getInstance(String): Controller
+ getPool(): ShapePool
+ getScore(): int
+ getClown(): Clown
+ setCircus(Circus): void
+ checkIntersection(): boolean
+ setPool(): void
+ getBackground(): Background
player
MusicCtrl
+ MusicCtrl():
~ clip: Clip
+ runMusic(String): void
+ close(): void
musicCtrl
«interface»
World
+ getConstantObjects( ): List<GameObjects>
+ getControlSpeed( ): int
+ getControlableObjects( ): List<GameObjects>
+ getHeight( ): int
+ getMovableObjects( ): List<GameObjects>
+ getSpeed( ): int
+ getStatus( ): String
+ getWidth( ): int
+ refresh( ): boolean

2.2. Design Patterns

- ### State Pattern

This design pattern is implemented as an interface called State which contains three functions implemented by three classes named: Static, Horizontal and Vertical. Each ImageItem (class implementing GameObject interface) has an object of type State representing its method of motion in the game. For example, the background has Static State while shapes have Vertical State until they are collected by the clown when they change their state to Horizontal State similar to the controlled clown.

ImageItems of state Static don’t change their x or y co-ordinates no matter what happened while these with Vertical state change only their y co-ordinate only unlike to those with Horizontal state change their x co-ordinate.

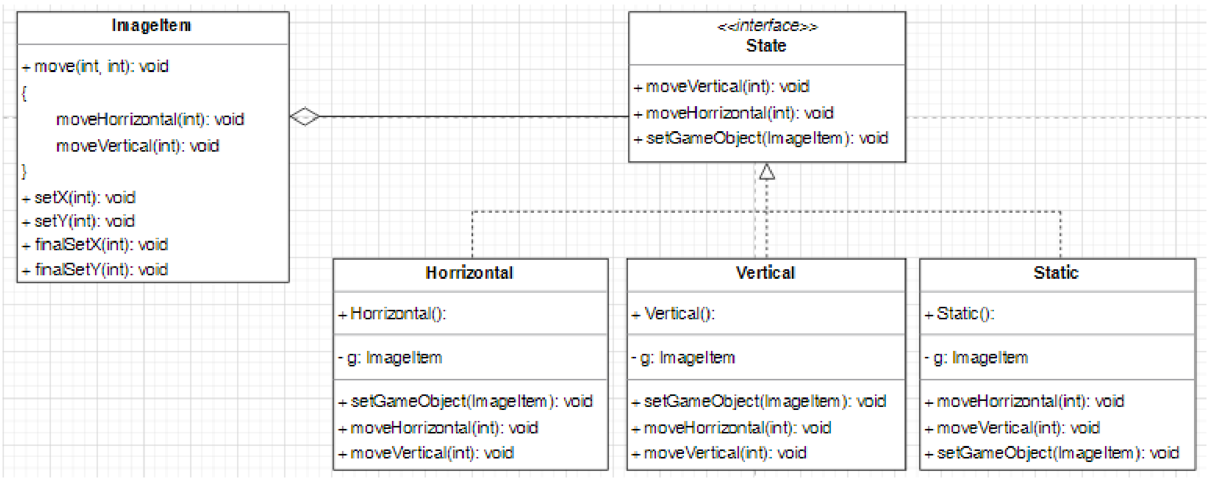

ImageItem
+ move(int, int): void
{
moveHorrizontal(int): void
moveVertical(int): void
}
+ setX(int): void
+ setY(int): void
+ finalSetX(int): void
+ finalSetY(int): void
<<interface>>
State
+ moveVertical(int): void
+ moveHorrizontal(int): void
+ setGameObject(ImageItem): void
Horrizontal
+ Horrizontal():
- g: ImageItem
+ setGameObject(ImageItem): void
+ moveHorrizontal(int): void
+ moveVertical(int): void
Vertical
+ Vertical():
- g: ImageItem
+ setGameObject(ImageItem): void
+ moveHorrizontal(int): void
+ moveVertical(int): void
Static
+ Static():
- g: ImageItem
+ moveHorrizontal(int): void
+ moveVertical(int): void
+ setGameObject(ImageItem): void

Fig. I: State Design Pattern

- <u>Factory Pattern</u>

  This design pattern is implemented by creating a factory class for each shape which is responsible for returning an object of type of the shape corresponding to this factory as well as randomizing its color.

- <u>Abstract Factory Pattern</u>

  This design pattern is implemented by creating a common factory for all the shapes falling during the game. The class randomizes the type of chosen shape and accordingly calls the get function of the factory corresponding to this shape.

- <u>Pool Pattern</u>

  This design pattern is implemented by creating a class in which all the shapes falling in the current played level are built through calling the abstract factory for number of times equivalent to the number of falling shapes set for each level. It also provides the functionality of reusing the shape after being collected or after exiting the frame by shuffling its type and color from the shapes in the pool as well as shuffling its position.

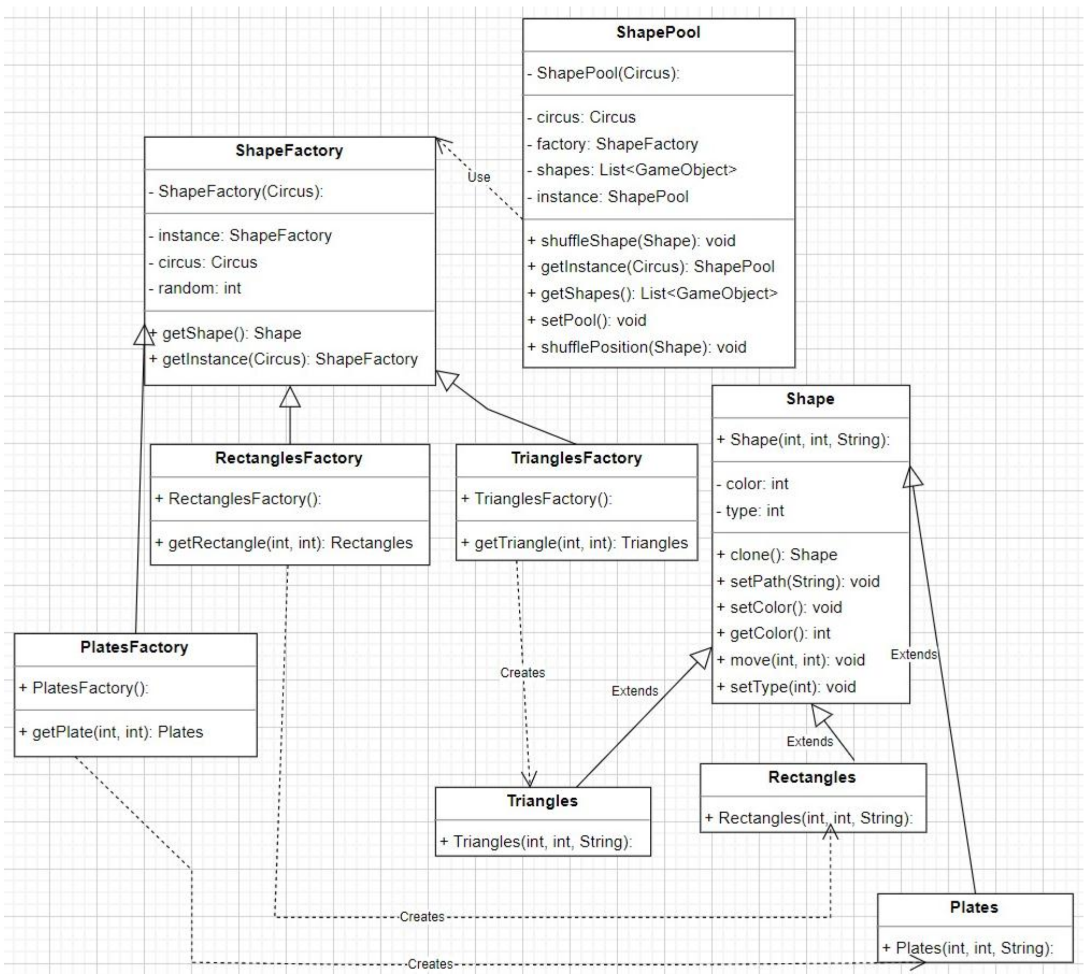


Fig. II: Factory, Abstract Factory and Pool Design Patterns

- <u>Strategy Pattern</u>

  This pattern is implemented as an interface called LevelStrategy which is implemented by three classes named: Level1, Level2 and Level3. Each Circus (which is a class implementing the World interface) has a level strategy which is responsible for setting the number of falling shapes as well as the speed of movement of shapes while falling as well as the time allowed for the level. Moreover, it conducts the corresponding music to the level into the game frame as well as adding bombs for desired level difficulty.

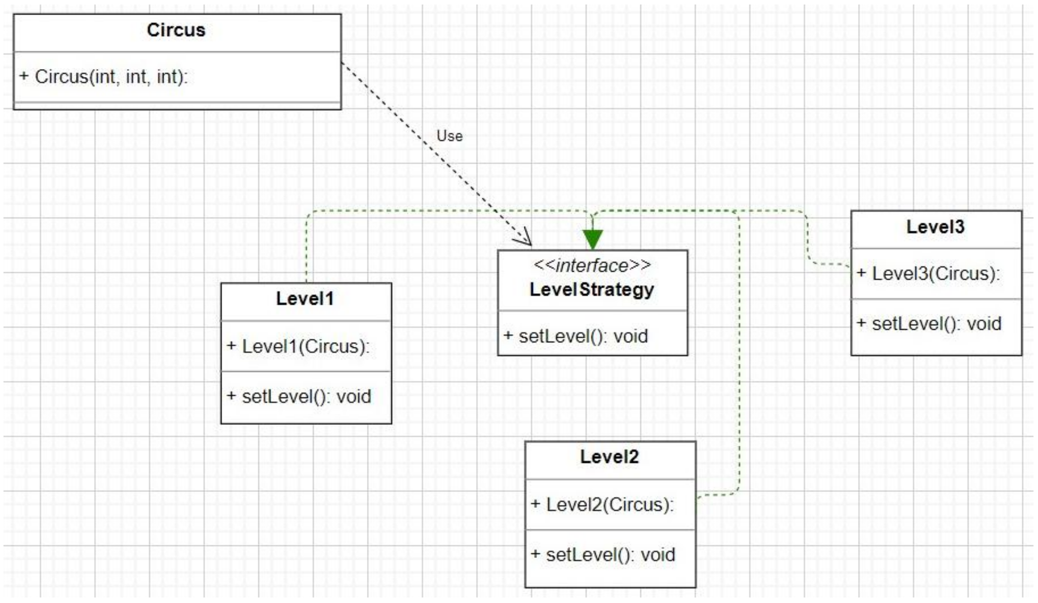


Fig. III: Strategy Design Pattern

- <u>Singleton Pattern</u>

  This pattern is used in many classes by constructing them only once then any time they are needed afterwards an instance is produced from this single object and only its internal instances are modified if needed. The following classes are implemented as Singleton: Controller, ShapePool and ShapeFactory.

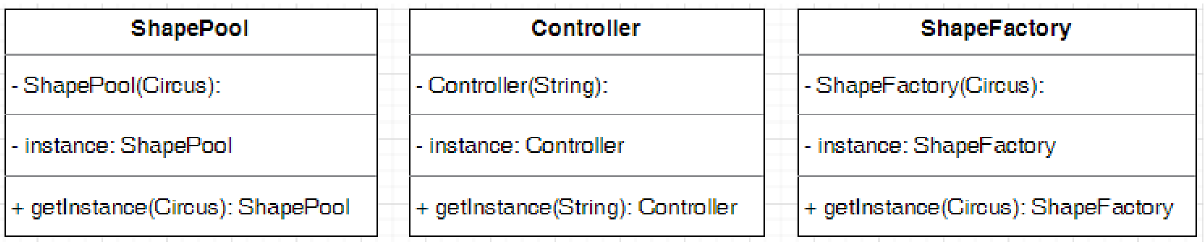


Fig. IV: Singleton Design Patterns

- Iterator Pattern

  This design pattern is not manually implemented by us in the project although it is used by the aid of the previously built in interface (Iterator). It is used on looping over the list containing moving objects (falling shapes) during checking their intersection with the clown's stacks to process them whether by moving them or by adding them to the clown's stack they have intersected with.

- MVC (Model, View, Controller) Pattern

  This design pattern is already built in previously in our project as the provided game engine is representing the implementation of the view and the controller parts while we are asked to implement the model part. Although, we classified our classes into three packages: Model, View and Control.

  Model package contains the ImageItem class (which implements GameObject interface) and the classes extending it (Background, Shape (Plates, Triangles, Rectangles and Bomb) and Clown), the State interface and the classes implementing it (Static, Horizontal and Vertical), the LevelStrategy interface and the three classes implementing it (Level1, Level2 and Level3), the three factories of the shapes as well as the common shape factory and the shapes pool.

  View package includes the Circus class (which implements World interface) as well as a Master frame which is a GUI frame used for receiving the requested level by the player.

  Control package has Controller class that handles all the game interactions through implementing the algorithms required for the refresh function in the Circus class (checking intersection as well as descending the shapes and shuffling them after being used) and a MusicCtrl class which is responsible for controlling the music played in the background during playing the level.

3. User Manual

- You can pause and resume the game anytime while playing and don’t get worried, if you paused the game then you choose to play new game, the new game will begin normally.
- You must avoid collecting bombs they will cause your loss.
- You should not reach more than 20 shapes in the same stack otherwise you will lose.
- When the game ends whether by the end of time or due to any of the two previously mentioned cases and you want to play a new game, don’t select new from the file window but you should close the current level and automatically you will be sent once again to the home page in order to choose the level of your next game.
- The shape is collected if it intersects with the top of your stack not your clown’s actual hand.
- This game is a mini-game you don’t need to save your scores or load your previous sessions. Just enjoy the game for your play time and have fun ☺.

4. Sample Runs

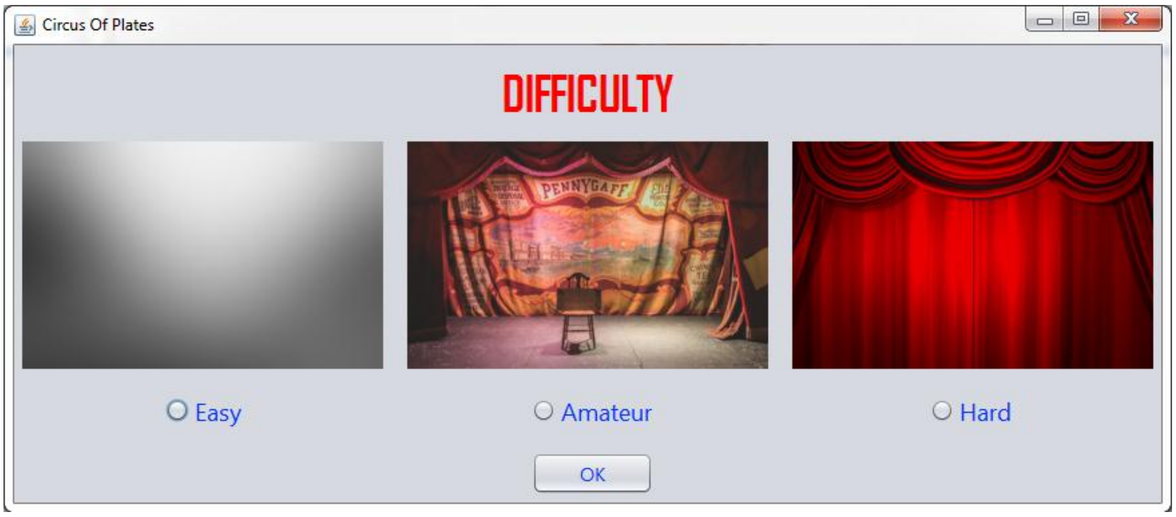


Fig. 01: Home Page

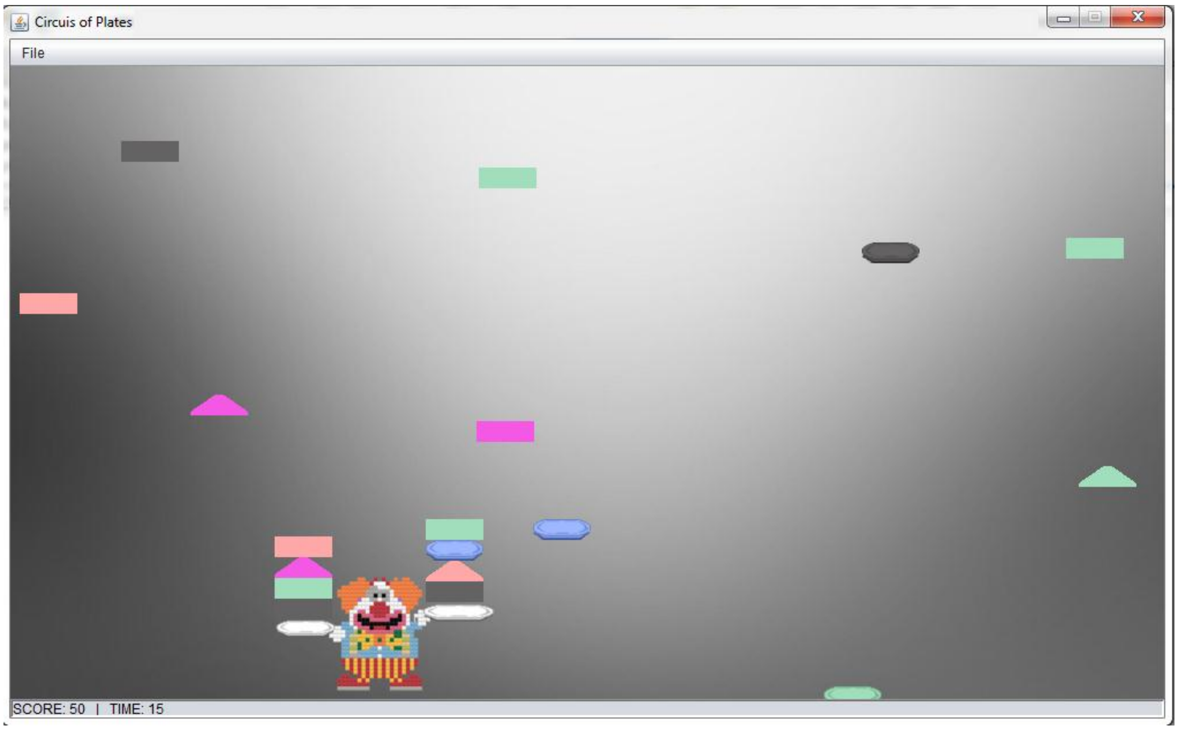


Fig. 02: Level One

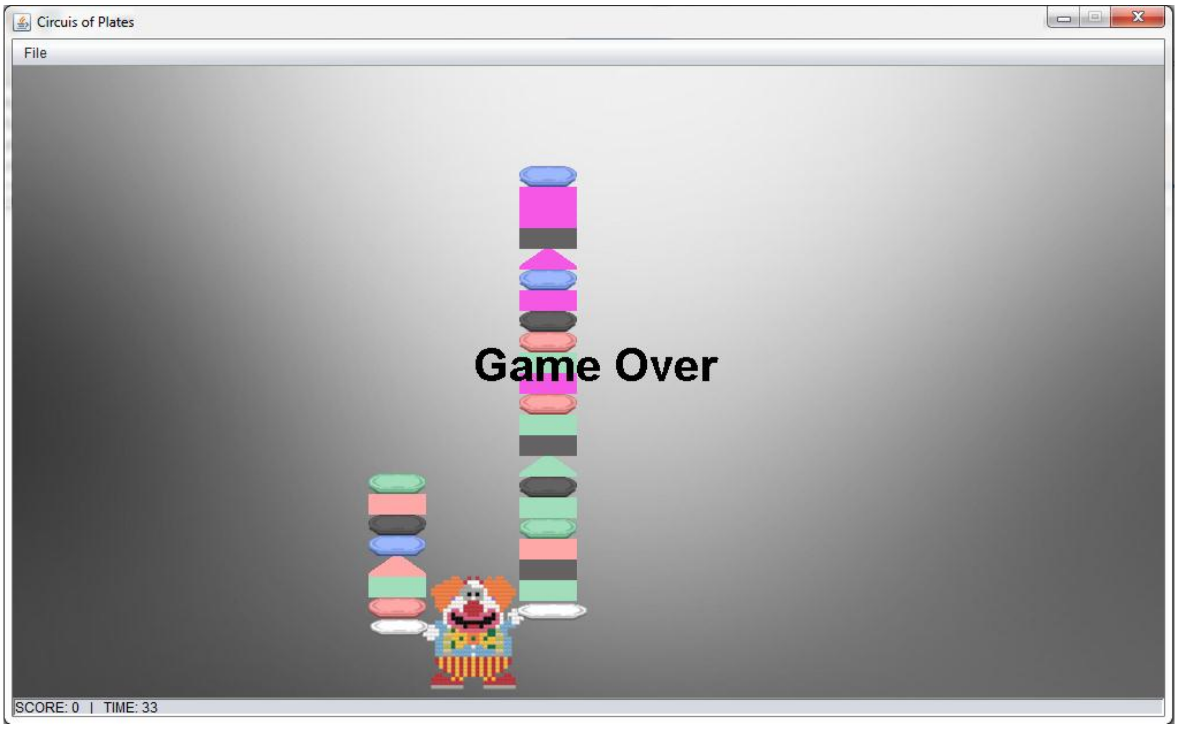


Fig. 03: Level One (Game Over due to more than 20 plates in one stack)

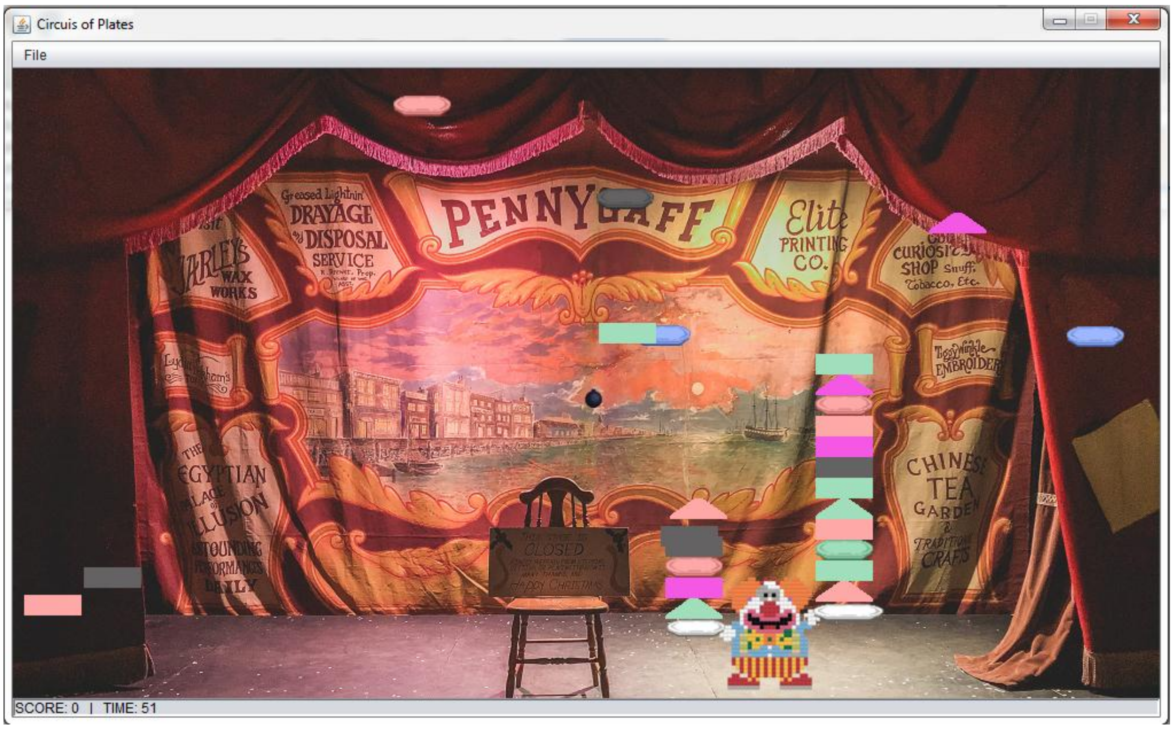


Fig. 04: Level Two

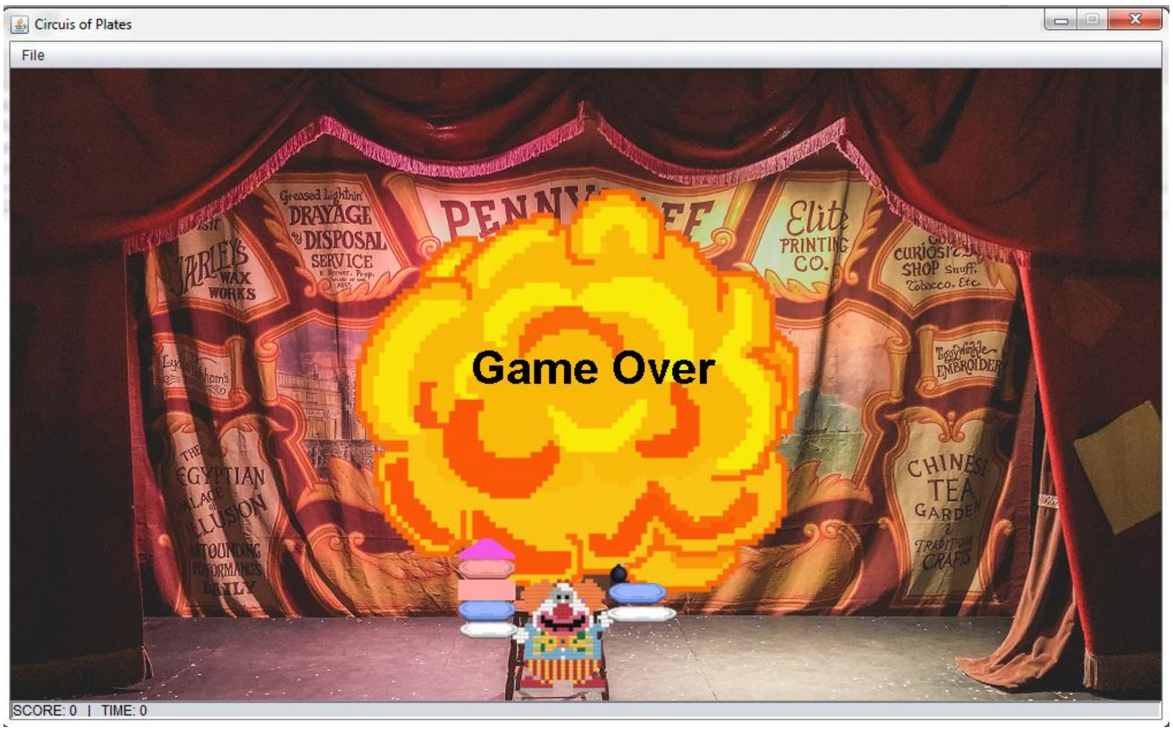


Fig. 05: Level Two (Game Over due to collecting bomb)

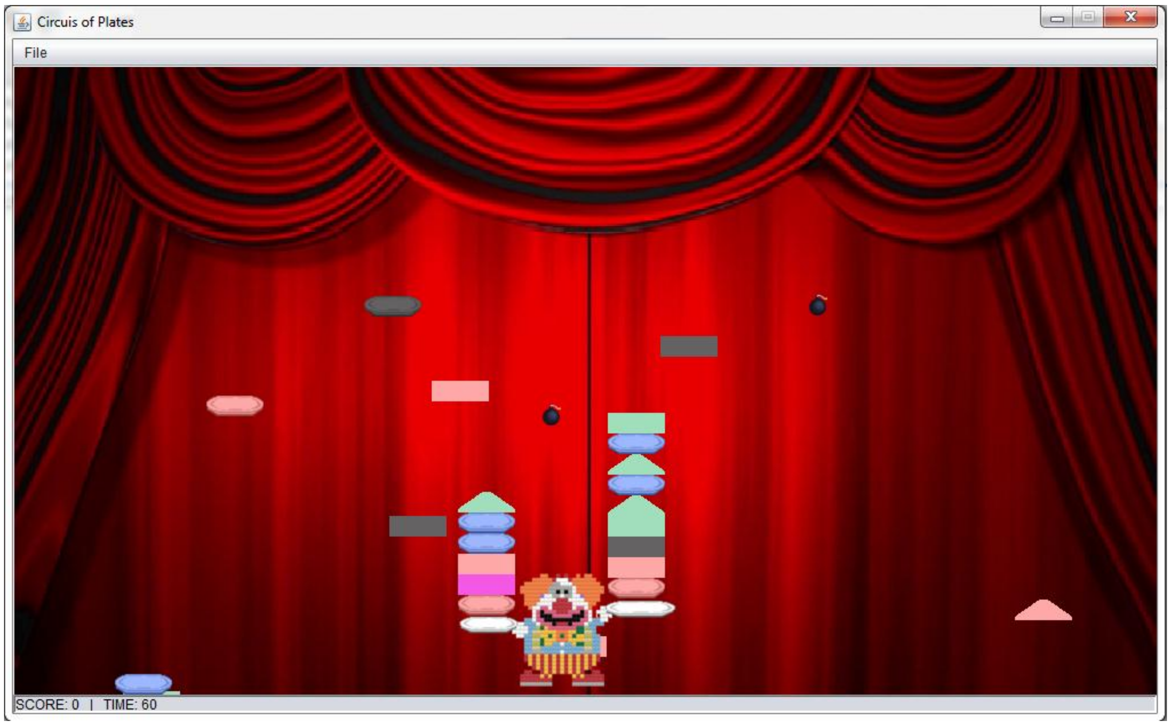


Fig. 06: Level Three

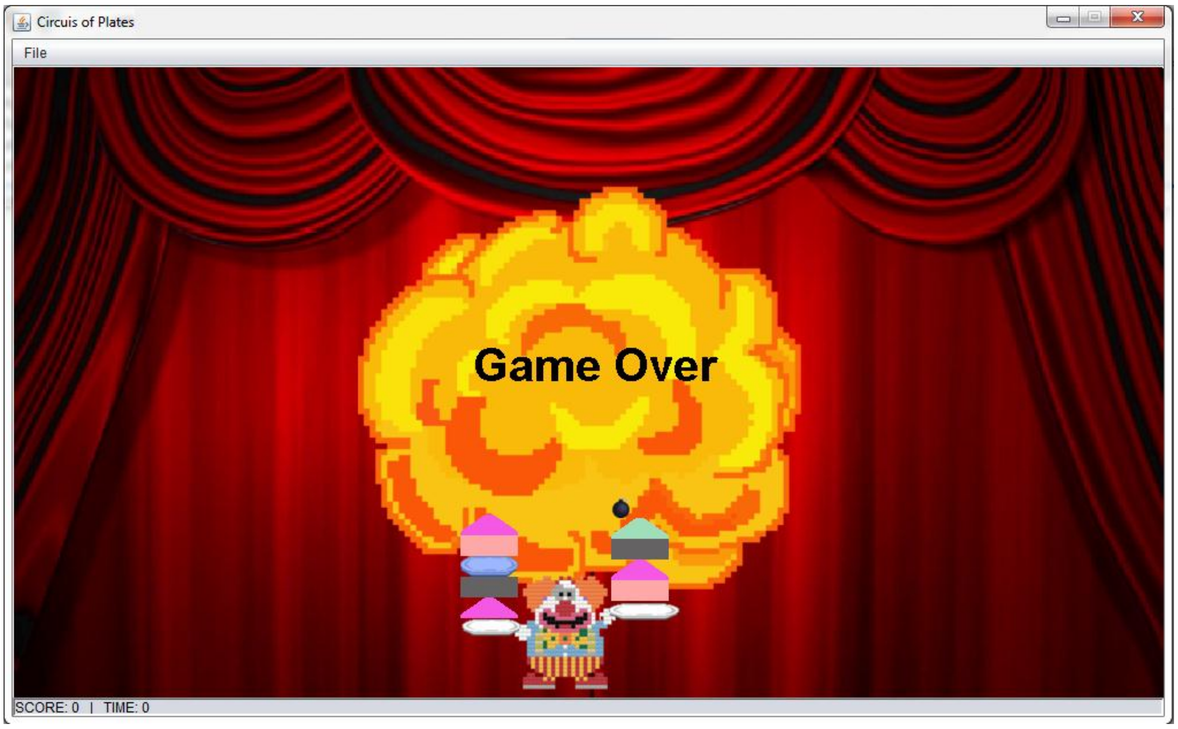


Fig. 07: Level Three (Game Over due to collecting bomb)