Presented by:

ahmed khaled

Ahmed yakout

marwan tammam

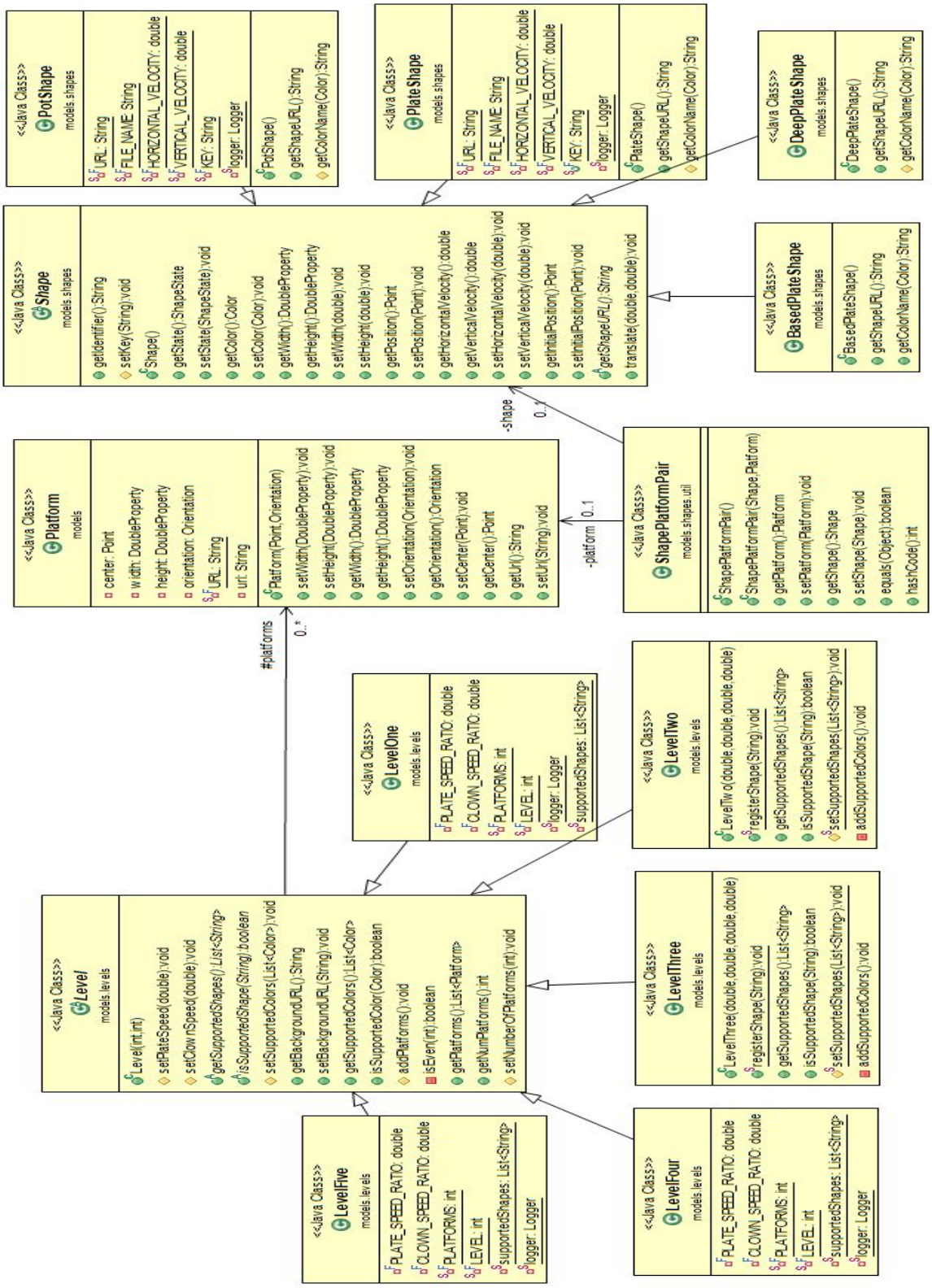
MOHAMMED ABD EL BARRY

Circus of plates game report

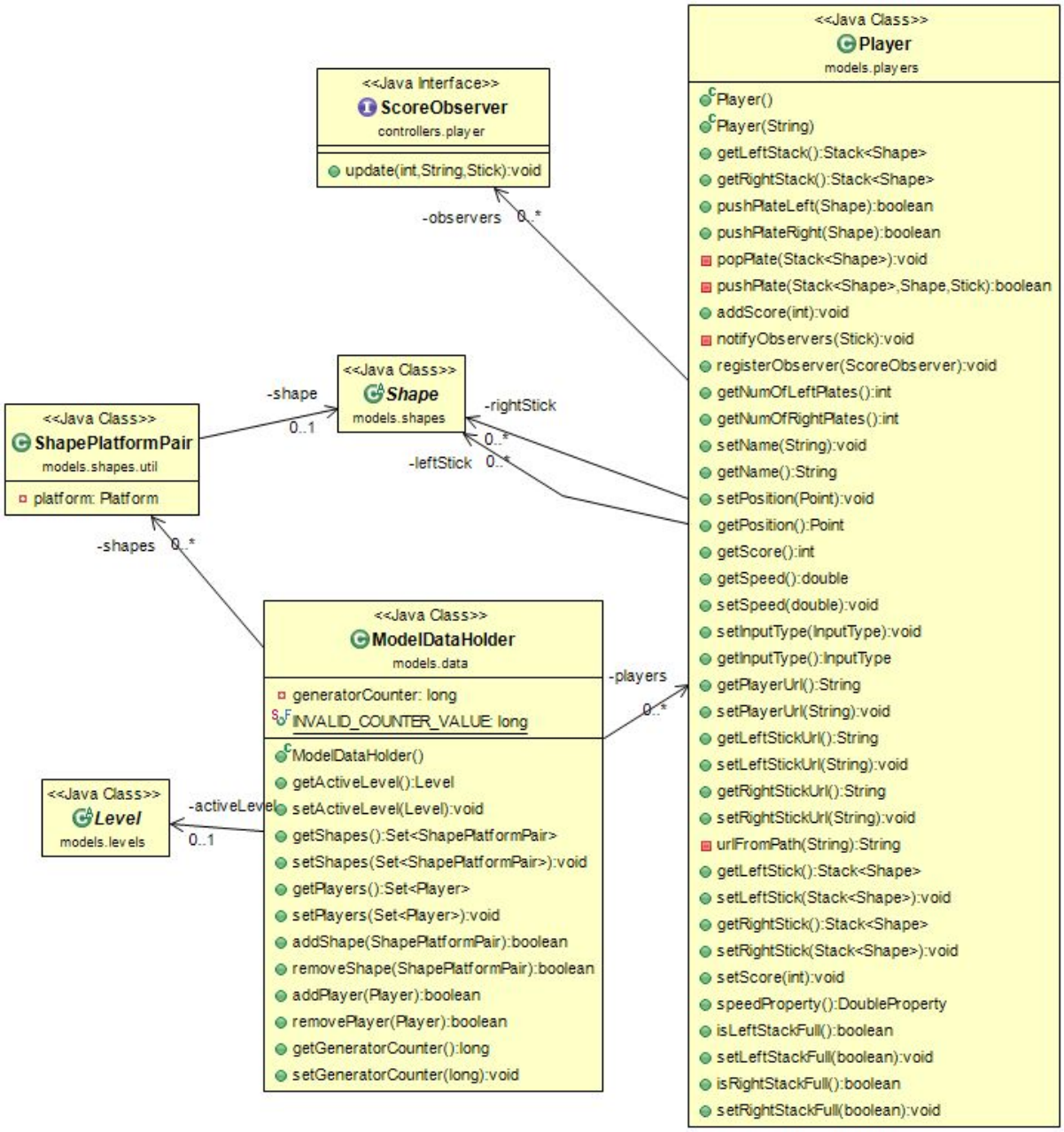
January 25, 2017

CLASS DIAGRAMS

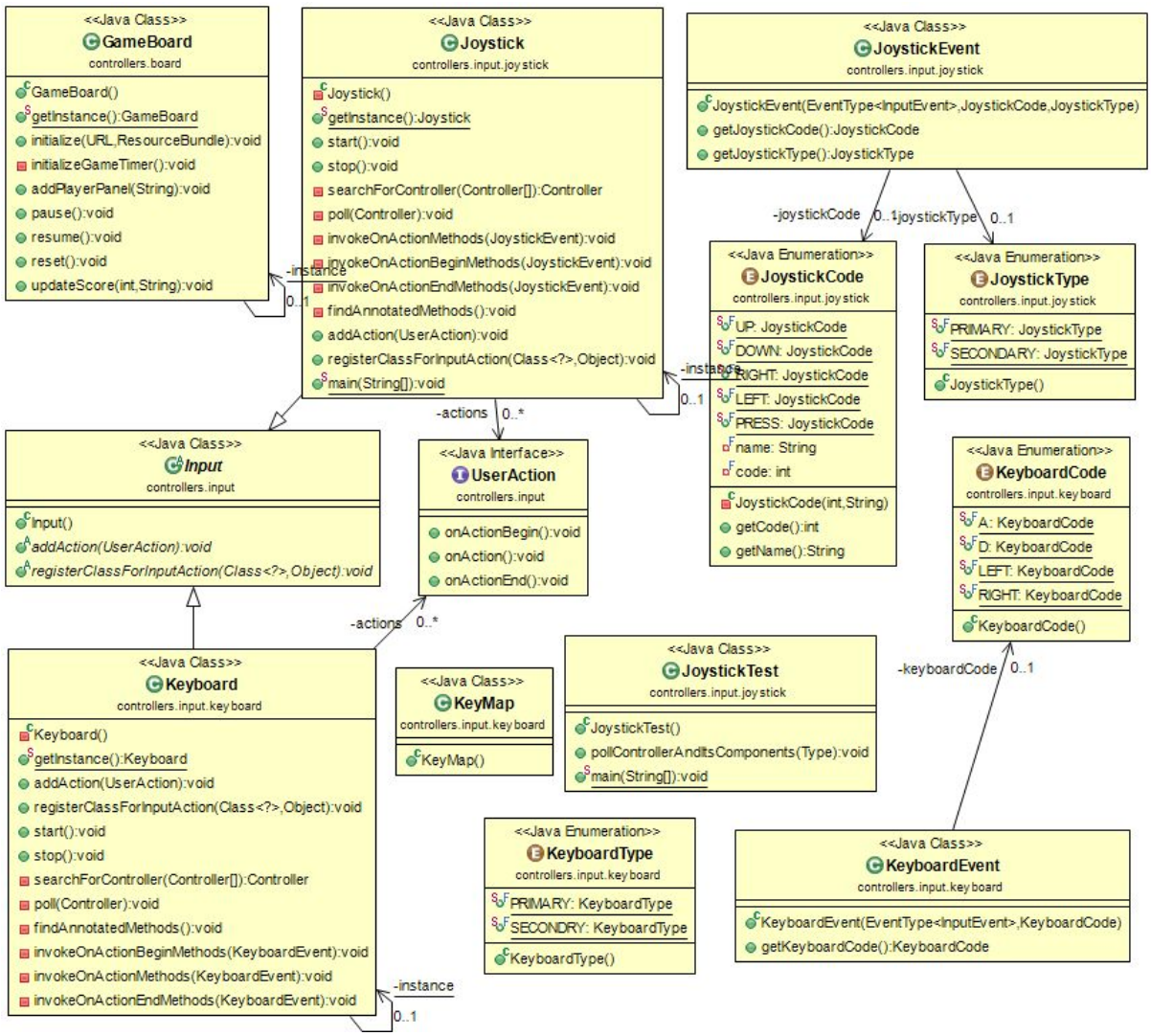
Model:



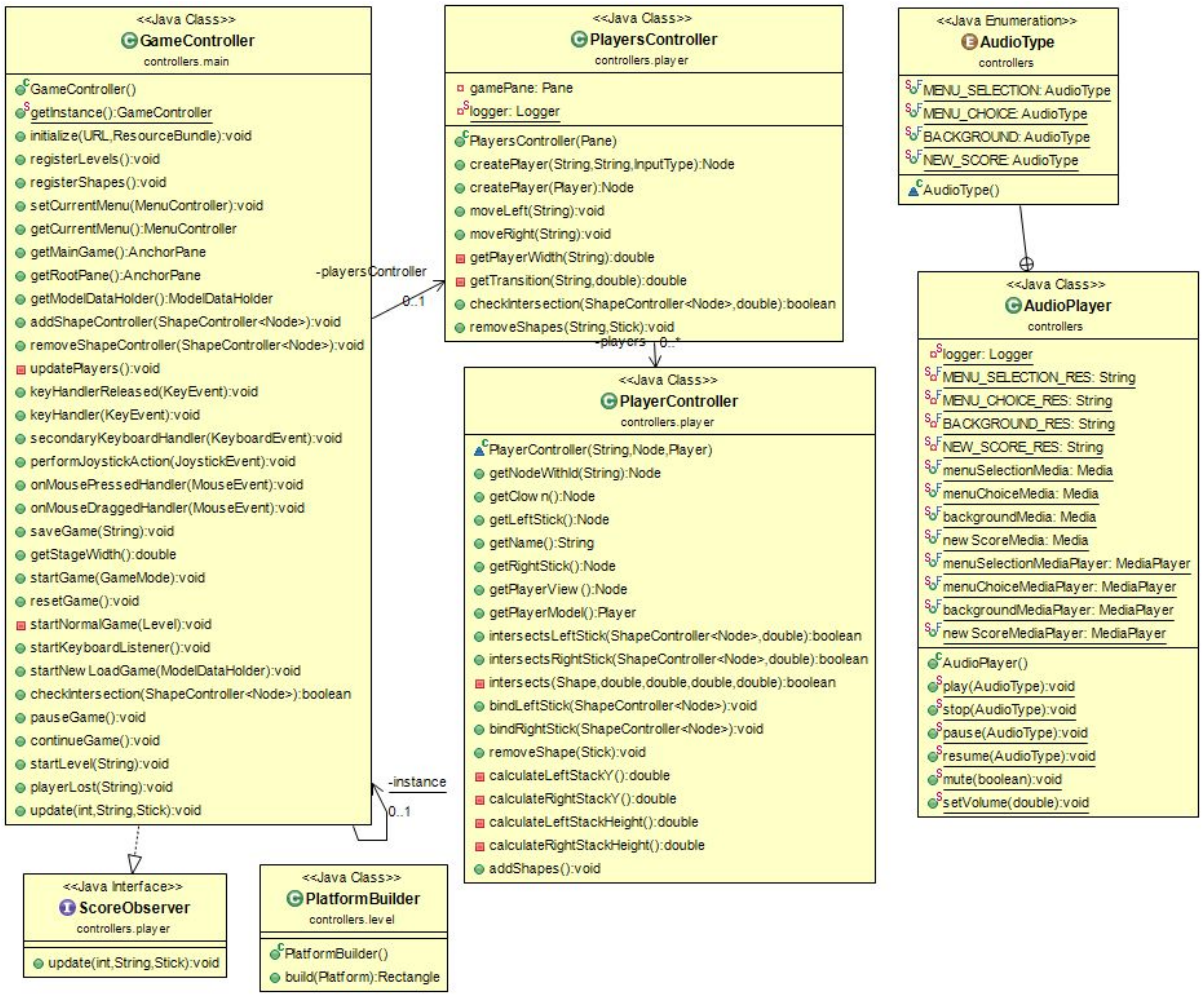
Model data holder and player:



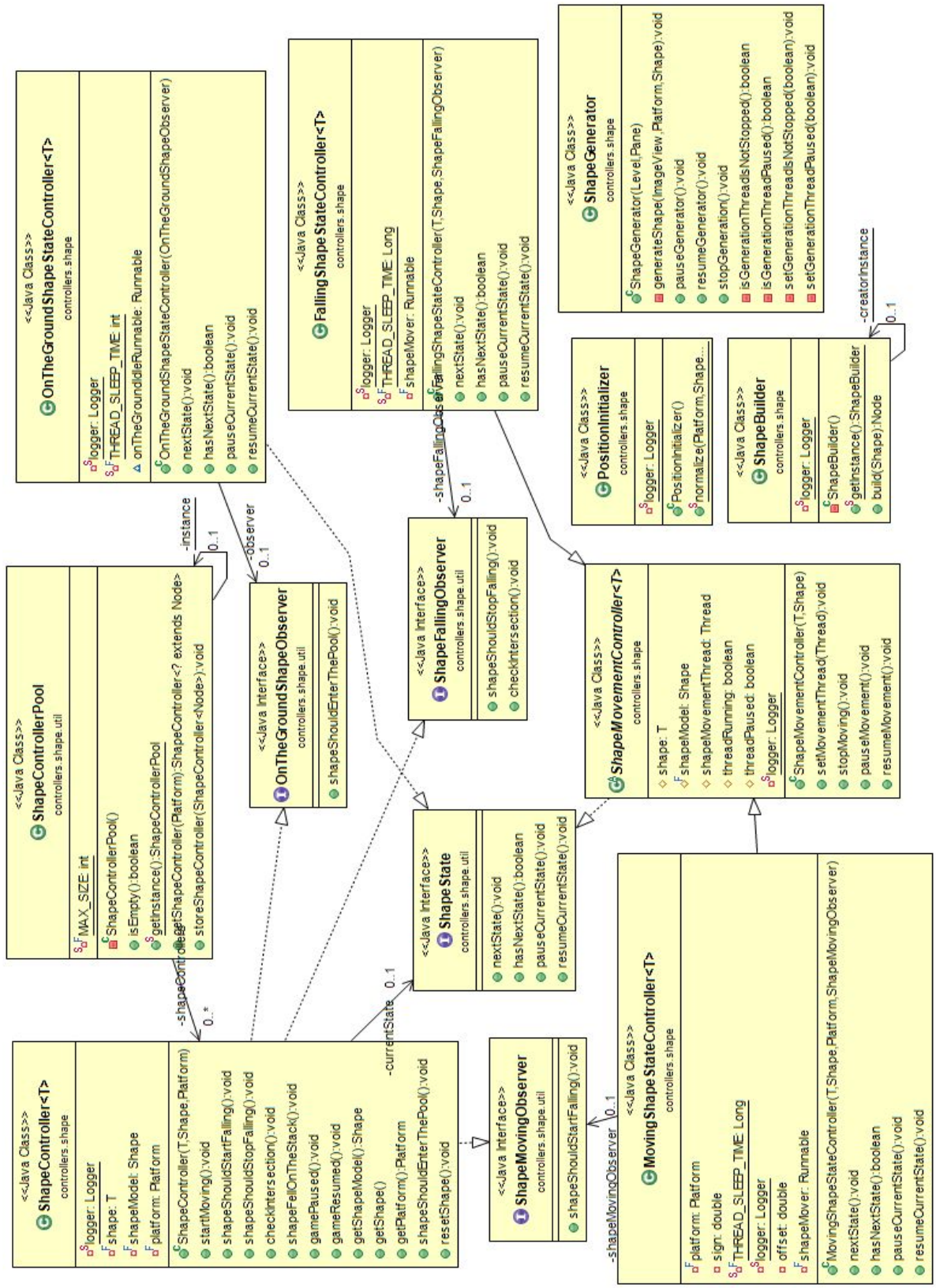
Controller (input and board):



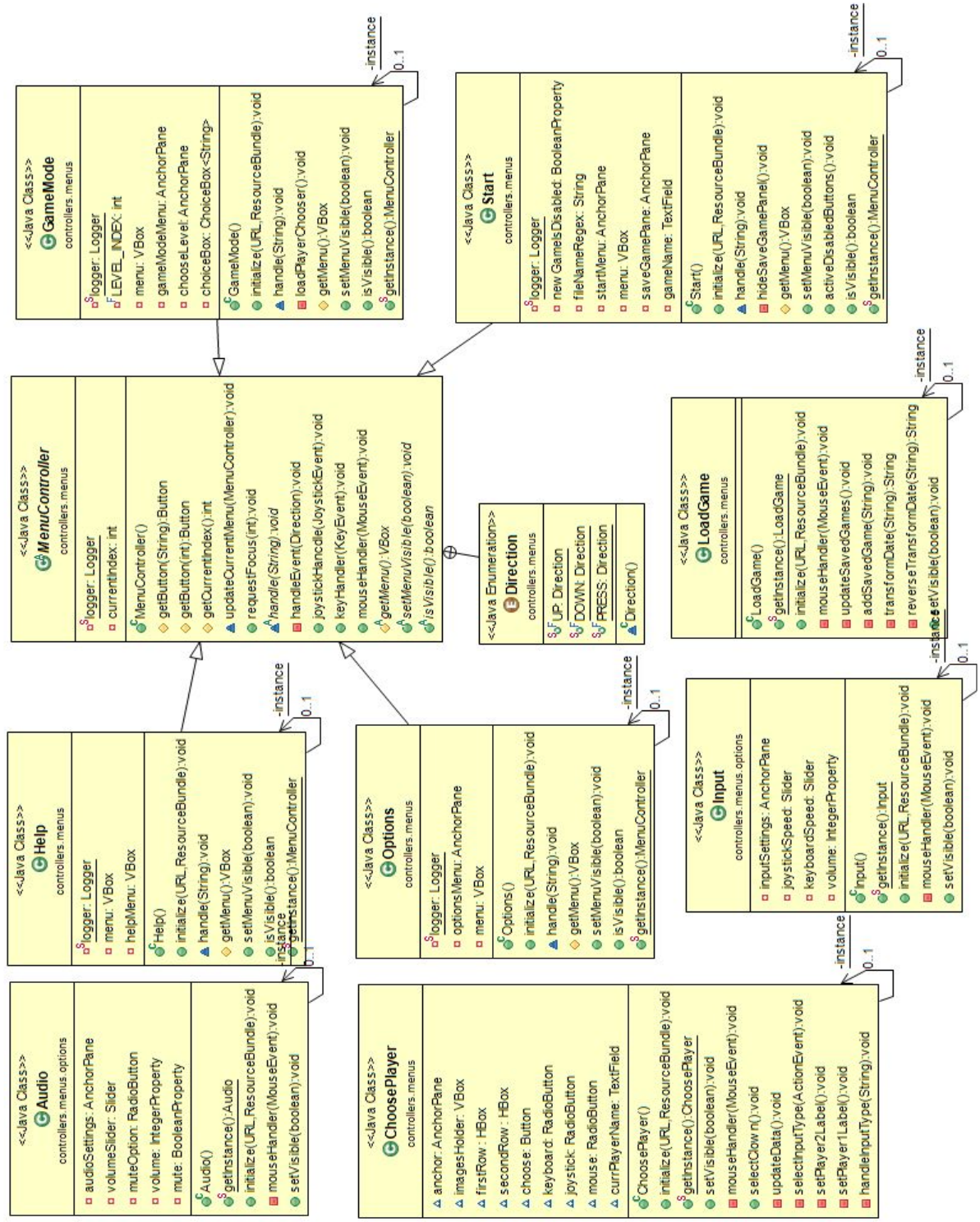
Platform, player and game controllers:



Shape controllers:

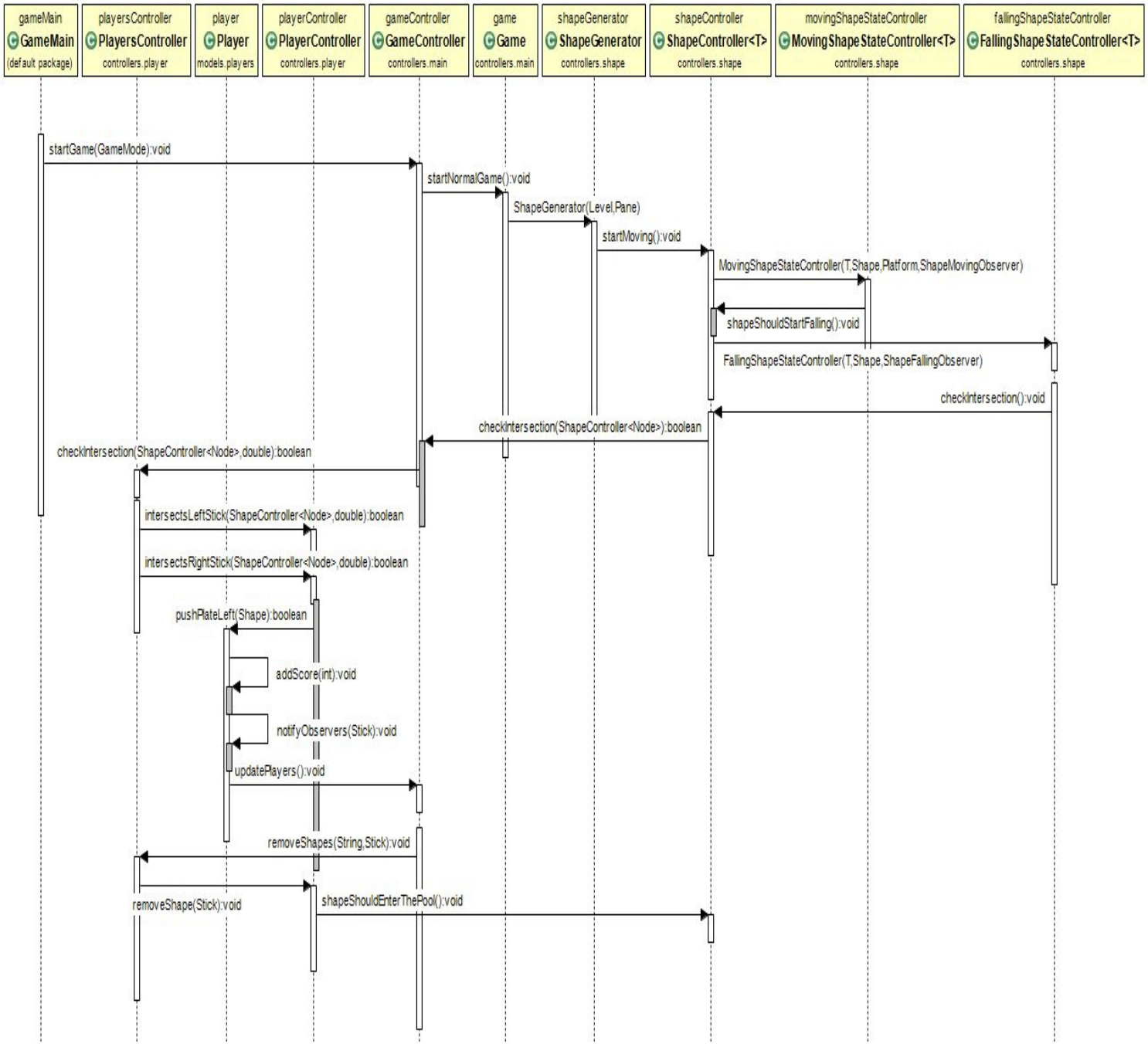


Menu Controllers:

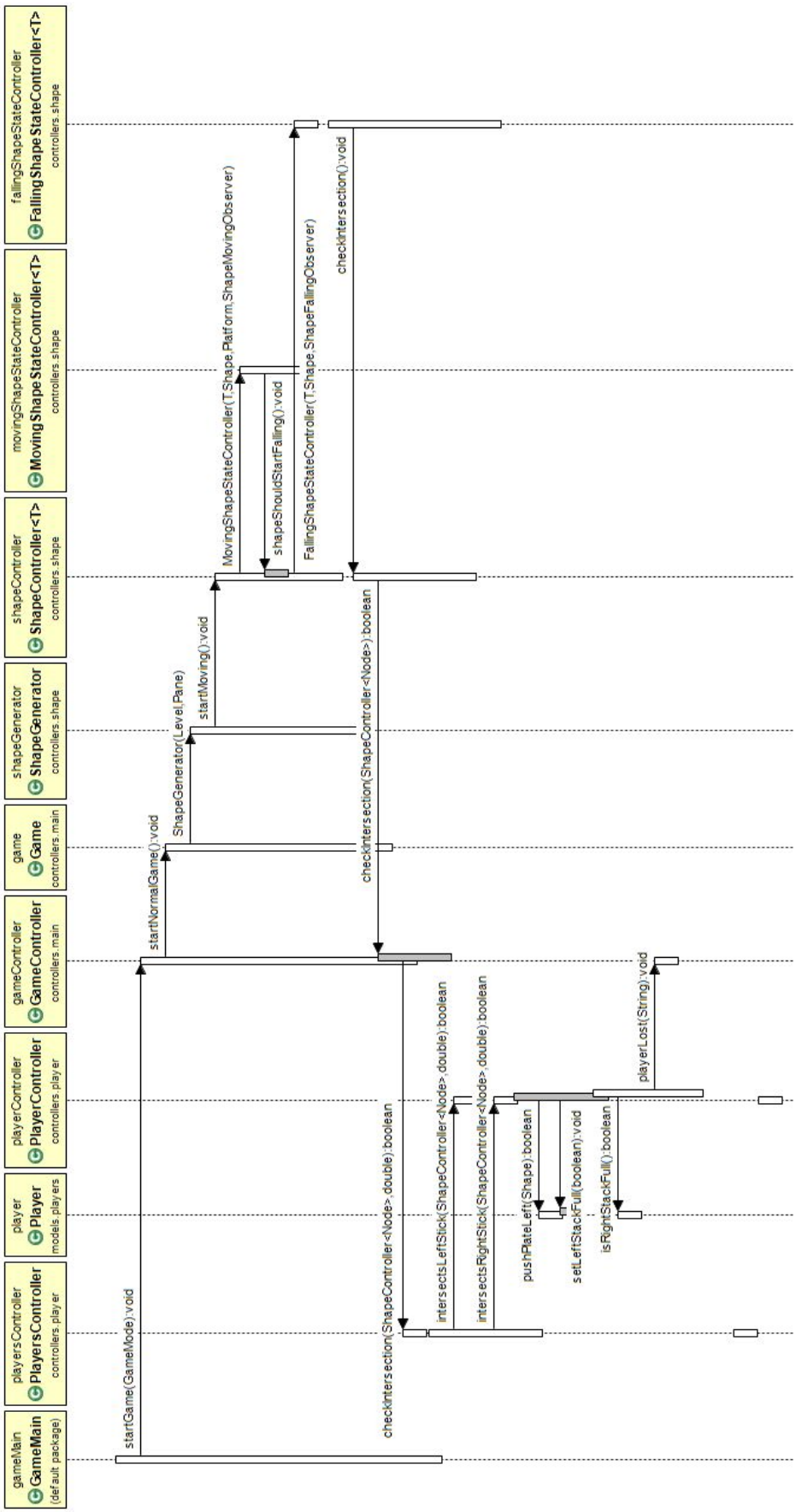


SEQUENCE DIAGRAM

Scenario 1: Shape life cycle when it falls and intersects with a player's left stick which has 2 other plates of the same color, where the shape is removed, score updated and the shape added to the pool.


gameMain
GameMain
(default package)
playersController
PlayersController
controllers.player
player
Player
models.players
playerController
PlayerController
controllers.player
gameController
GameController
controllers.main
game
Game
controllers.main
shapeGenerator
ShapeGenerator
controllers.shape
shapeController
ShapeController<T>
controllers.shape
movingShapeStateController
MovingShapeStateController<T>
controllers.shape
fallingShapeStateController
FallingShapeStateController<T>
controllers.shape
startGame(GameMode):void
startNormalGame():void
ShapeGenerator(Level,Pane)
startMoving():void
MovingShapeStateController(T,Shape,Platform,ShapeMovingObserver)
shapeShouldStartFalling():void
FallingShapeStateController(T,Shape,ShapeFallingObserver)
checkIntersection():void
checkIntersection(ShapeController<Node>):boolean
checkIntersection(ShapeController<Node>,double):boolean
intersectsLeftStick(ShapeController<Node>,double):boolean
intersectsRightStick(ShapeController<Node>,double):boolean
pushPlateLeft(Shape):boolean
addScore(int):void
notifyObservers(Stick):void
updatePlayers():void
removeShapes(String,Stick):void
removeShape(Stick):void
shapeShouldEnterThePool():void

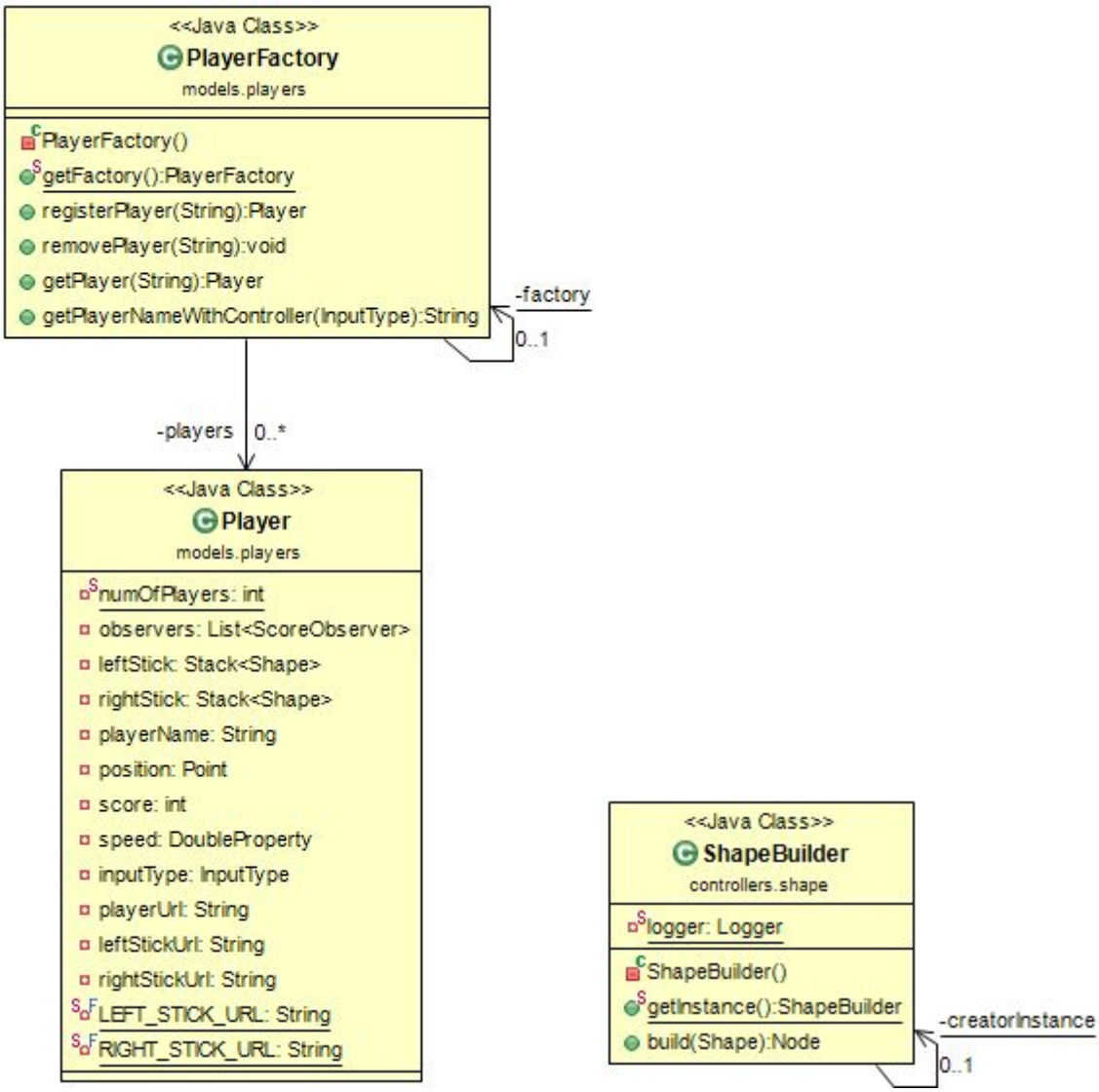
Scenario 2: Shape life cycle when both the stacks get filled and the player loses.


gameMain
GameMain
(default package)
playersController
PlayersController
controllers.player
player
Player
models.players
playerController
PlayerController
controllers.player
gameController
GameController
controllers.main
game
Game
controllers.main
shapeGenerator
ShapeGenerator
controllers.shape
shapeController
ShapeController<T>
controllers.shape
movingShapeStateController
MovingShapeStateController<T>
controllers.shape
fallingShapeStateController
FallingShapeStateController<T>
controllers.shape
startGame(GameMode):void
startNormalGame():void
ShapeGenerator(Level.Pane)
startMoving():void
MovingShapeStateController(T.Shape,Platform,ShapeMovingObserver)
shapeShouldStartFalling():void
FallingShapeStateController(T.Shape,ShapeFallingObserver)
checkIntersection():void
checkIntersection(ShapeController<Node>):boolean
checkIntersection(ShapeController<Node>,double):boolean
intersectsLeftStick(ShapeController<Node>,double):boolean
intersectsRightStick(ShapeController<Node>,double):boolean
pushPlateLeft(Shape):boolean
setLeftStackFull(boolean):void
isRightStackFull():boolean
playerLost(String):void

Design patterns

Singelton

Used with ShapeBuilder and PlayerFactory classes to ensure that no more than one instance can be created. That's because there's no need to create may builders, and also this builder is synchronized. Additionally, the player factory registers the players and keeps track of them, so having multiple factories is not only meaningless but can lead to errors like registering a player twice or registering one player in one factory and the other in another instance of the factory. So having a single class ensures all players are registered in one place.



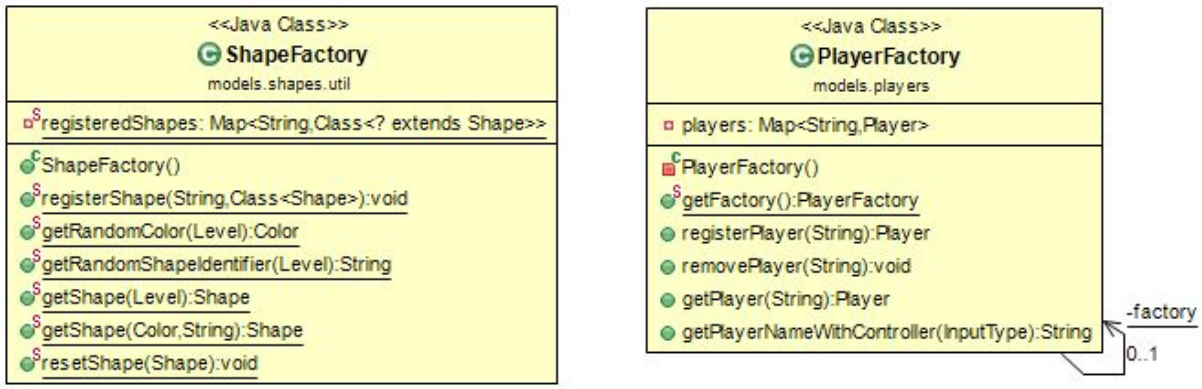
FACTORY

Used in places. First the ShapeFactory:

> The ShapeFactory's purpose is given a specific level, generates a random shape and color from the level's supported shapes and color if this shape identifier is registered in the factory. If not then this class was not loaded by the class loader, and thus not registered. Then this shape instance is returned by the factory. The factory also has a method to reset a given shape's properties. This method is called when the object is first created or by the pool before it adds it to its pool list.

Second the PlayerFactory:

> The player factory registers players and given an InputType object returns the user using this input from the map of registered users. It also enables removing players by name from the list of registered players or gets the player instance by name.

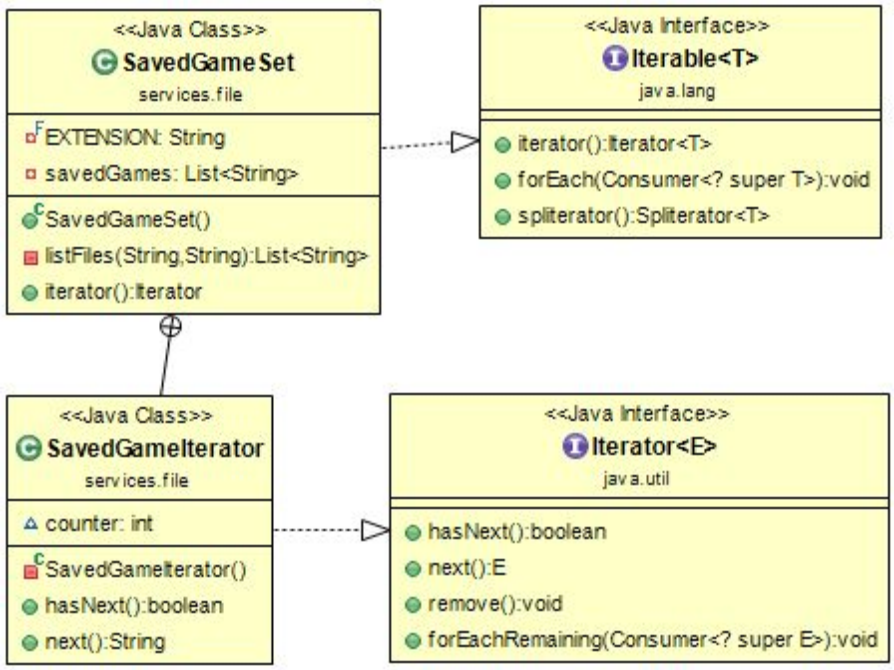


BUILDER

Builds an image view for a given shape object by copying the shape's properties and setting the position of the image view and other properties. Then returns this image view as a Node object which it the parent class of the image view. That allows for extensibility where the image view could later be replaced with any other class as long as it's a child/descendent of the node class.

ITERATOR

The SavedGameSet class implements the java.util.Iterable and contains an inner class SavedGameIterator implementing java.util.Iterator. They are used to iterate over the saved game files stored on disk. The listFiles method takes a path and a string representing file extension and returns a list of strings of the names of files with this extension in the given path. The SavedGameIterator iterates over this list.



DYNAMIC LINKAGE

Dynamic class loading is used in the ShapeLoader class where its sole method, loadShapes, given a path of any folder, iterates through the files in this directory, and each class file which is also extends the Shape class, is dynamically loaded and registered at the ShapeFactory. This method actually copies the directory to where the game is running so that the class files are in a folder called model. That's because it's required that the class to be loaded to be in the same directory as the game running and in directories with the name of its package. Finally after loading the classes, it deletes those copied files and the created directory.

This design pattern makes the game extensible and allows adding new shapes at runtime.

SNAPSHOT

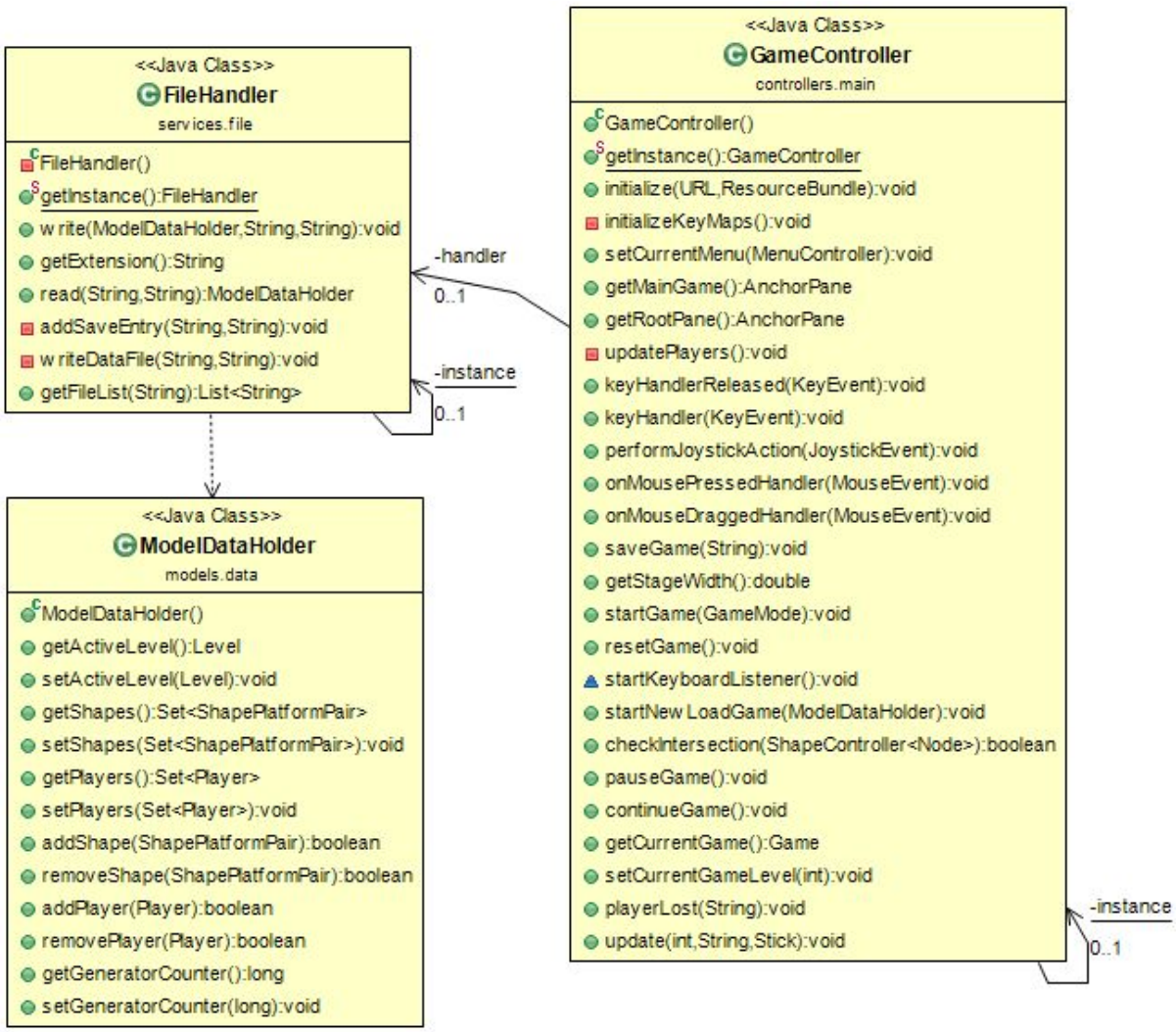
The ModelDataHolder class is continuously updated to hold the state of the game while running. When the user wishes to save the game, the controller gets an object of the ModelDataHolder containing the last state and sends it to the FileHandler to be saved. When the user loads, the ModelDataHolder's state is reset to be the same as the saved object on disk.

GameController asks the FileHandler for the ModelDataHolder.

Memento: ModelDataHolder

Caretaker: GameController

Originator: FileHandler



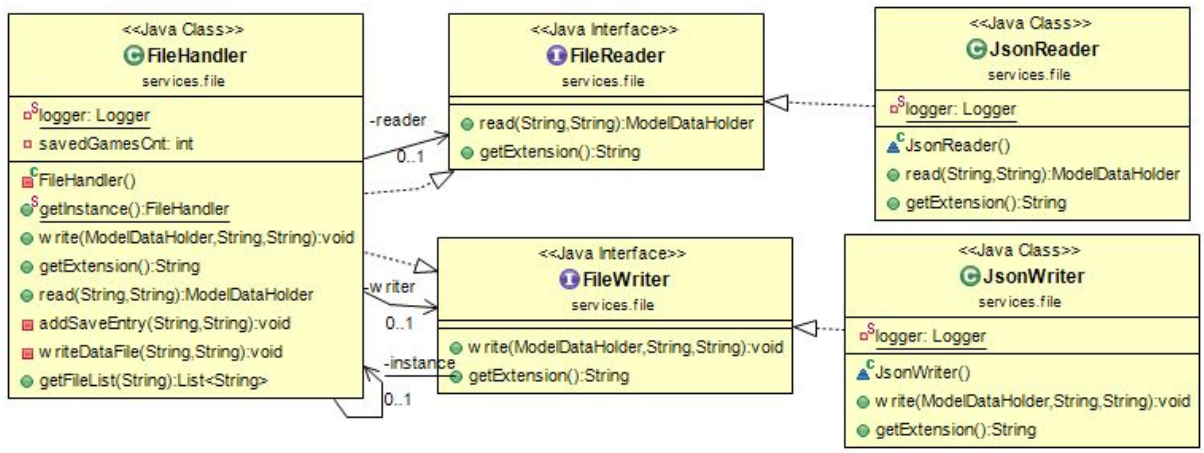
STATE

ShapeState interface defines 4 main methods; nextState, hasNextState, pauseCurrentState, resumeCurrentState. The interface is implemented by three classes representing the following states for the shape:

- OnTheGround: keeps the shape stationary after it has reached the ground and notifies the observer to add this shape to the shape pool. Has no next state.
- MovingShapeState: defines the initial movement of the shape when it's on the platforms and notifies the observer when the whole shape exits the platform where it should start moving vertically downwards. Has next state as FallingShapeState.
- FallingShapeState: defines the movement of the shape while falling vertically and has the next state as OnTheGround. Also notifies the observer when the shape intersects the upper part of any of the players' sticks where it should stop falling and attach to the stick.

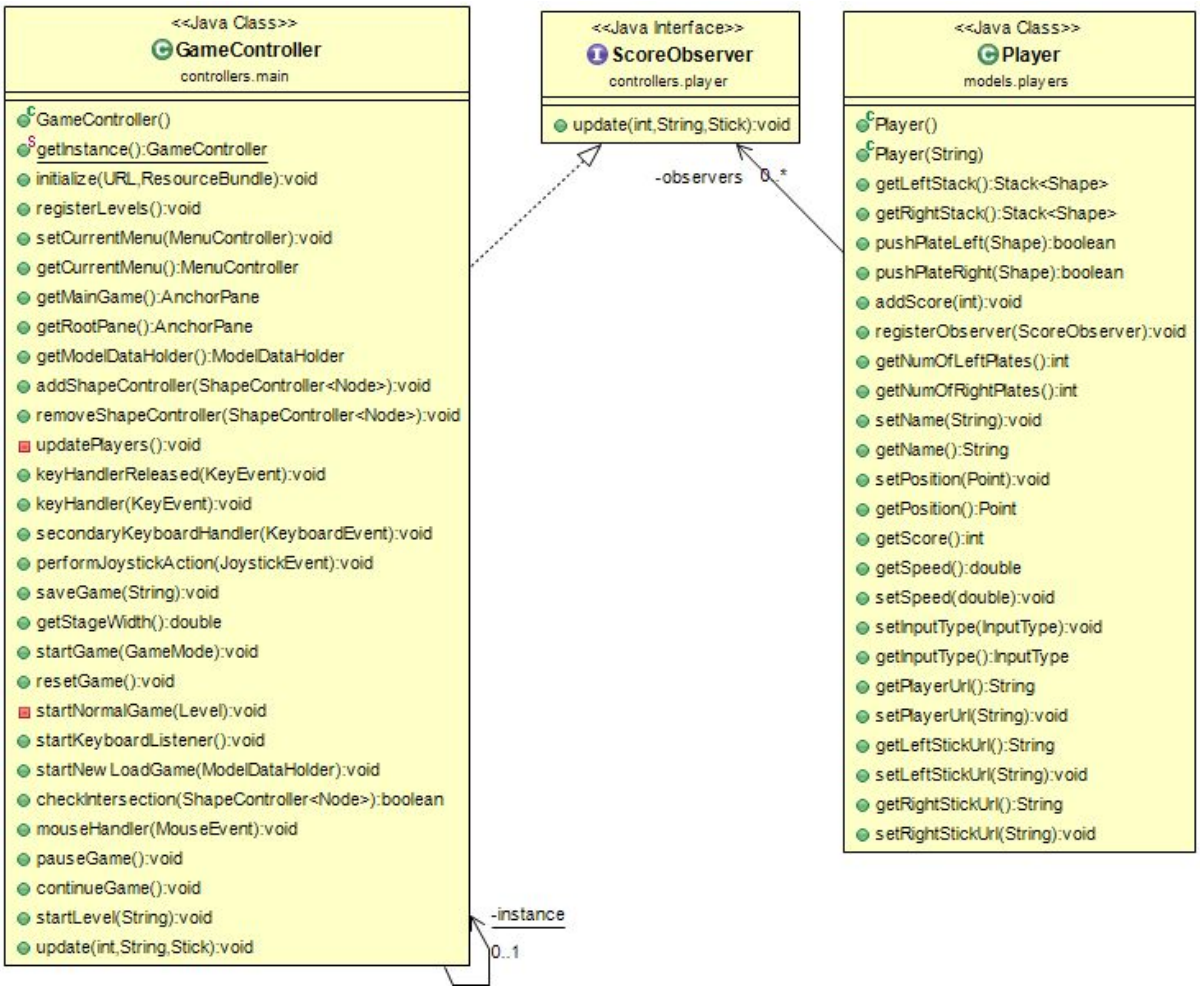
STRATEGY

Used in saving loading the game. Where there are two interfaces for file reading and file writing. Any classes implementing those interfaces can save and load the game. This enables the game to be easily saved in a different format by just implementing the interfaces.



OBSERVER

The observable is the Player class in the model. And the observers, which implement the ScoreObserver interface are the GameController objects. When three consecutive shapes of the same color fall on the stick the Player object pops the 3 shapes from the stick stack, increases the score and notify its observers by calling the update method passing the player name, score and stick object. The update method implemented in the GameController calls two methods. First the updateScore method of the game board, so that the new score is displayed. Second, the removeShapes method in the PlayersController which removes the plates from the actual view of the game.



MVC

The project is based on the MVC pattern where:-

1) The model:
    a. Player – holds data about the player like the name, score, speed, input type, position, etc. In addition, it keeps track of the plates on each of the left and right sticks.
    b. Settings – holds different game settings including audio, graphics, controller and general game settings.
    c. Shape – represents the shape falling and holds data about it such as the color, position, velocity, state, dimensions, etc.
    d. Platform
    e. Point – acts like the java.awt version but modified to store the x and y as DoubleProperty which is compatible and easier to use along with JavaFX instead of using integers or doubles in case of Point.Double2D
    f. ShapeFactory
    g. ShapeLoader
    h. ShapePoool
    i. ModelDataHolder – holds the necessary data in the model for saving and loading the game.
2) The View: using JavaFx for the view with fxml files
    a. Main menu: the default menu of the where the player can choose whether he needs to start a game or wants to set some settings or load a saved game, etc.
    b. StartGame menu: the menu appears on selecting NewGame from the main menu. In this menu the player can choose whether to set his clown, input controllers, volume of the music in game, etc.
    c. The background, plates, clowns, platforms, scores, timer(for time attack mode) and the sticks for clowns are set depending on each level.
    d. The movement of shapes on the platform and when falling corresponds the changes done in the ShapeController threads for movements.
    e. The movement of clowns is handled to the view according the user key-action.

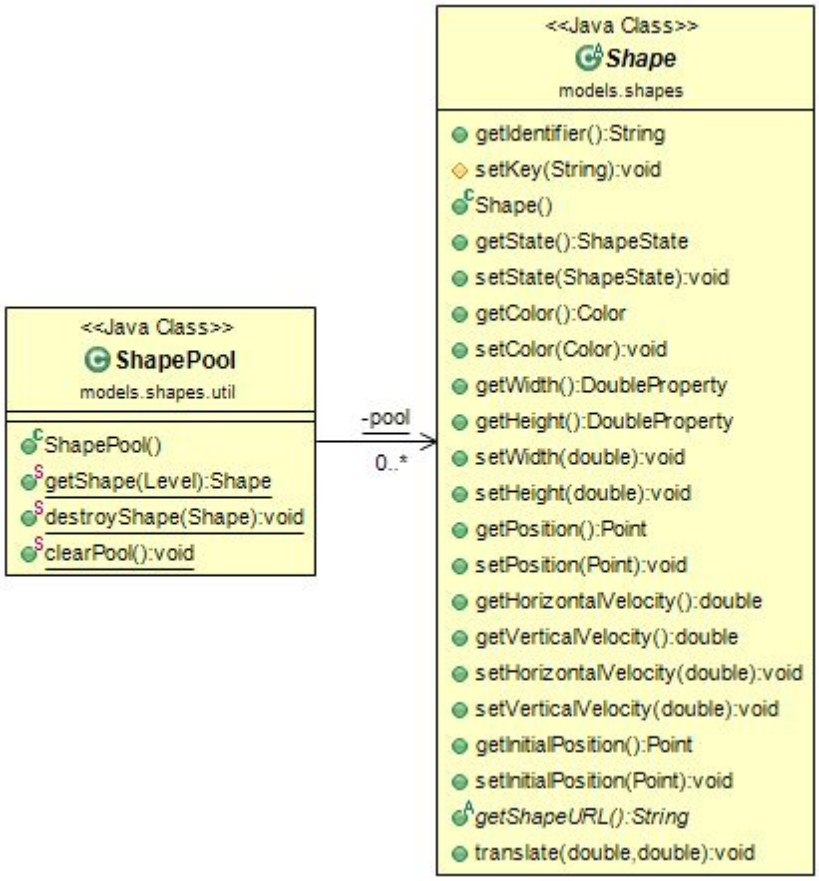
3) The controller:

OBJECT POOL

The ShapePool is used for reuse of shapes throughout the game. It's used to get a shape object which it does by generating a random color and shape identifier with the help of the methods already in the ShapeFactory. Then using the generated color and shape identifier, searches its list of shapes (the pool) for a shape with this color and identifier, and in case it exists remove and return it. If not found, the shape pool returns a new object by passing the color and shape identifier to the factory.

The shape pool also supports destroying shapes when they are no longer needed by resetting their states and adding them to the list of shapes. Lastly, it supports clearing the pool which is needed when moving to another level during the game or starting a new one where the current

shape in the pool may not be applicable.



Game Rules

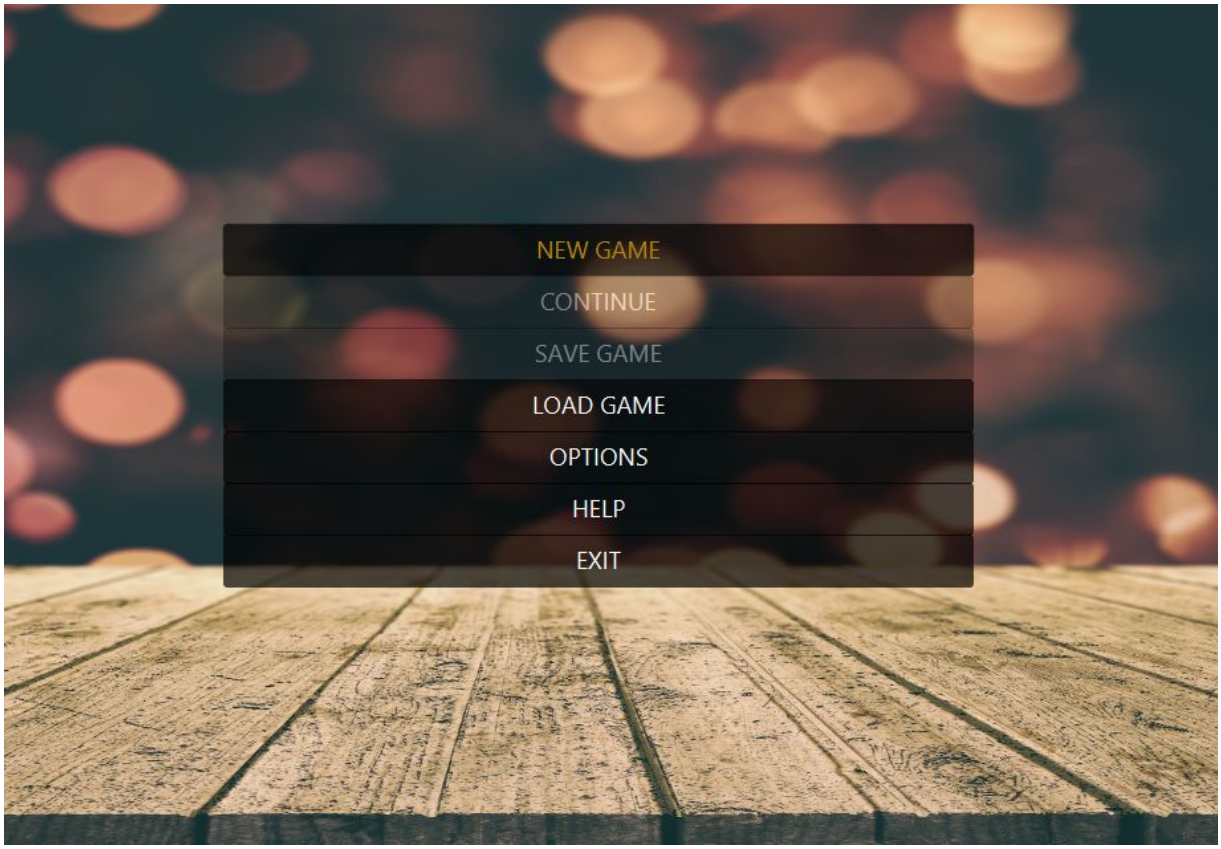
In the Normal mode, the game ends when both of a players' stacks are full. When that happens the player whose stacks full have his score halved and then the 2 scores are compared to determine the winner.

In the Time Attack mode, the same rule applies when a player's stacks become full before the time runs out. However, if the time runs out without any of the players' stacks getting filled, the game ends the scores are compared to determine the winner.

The stack is considered full if its height exceeds the highest platform in the game.

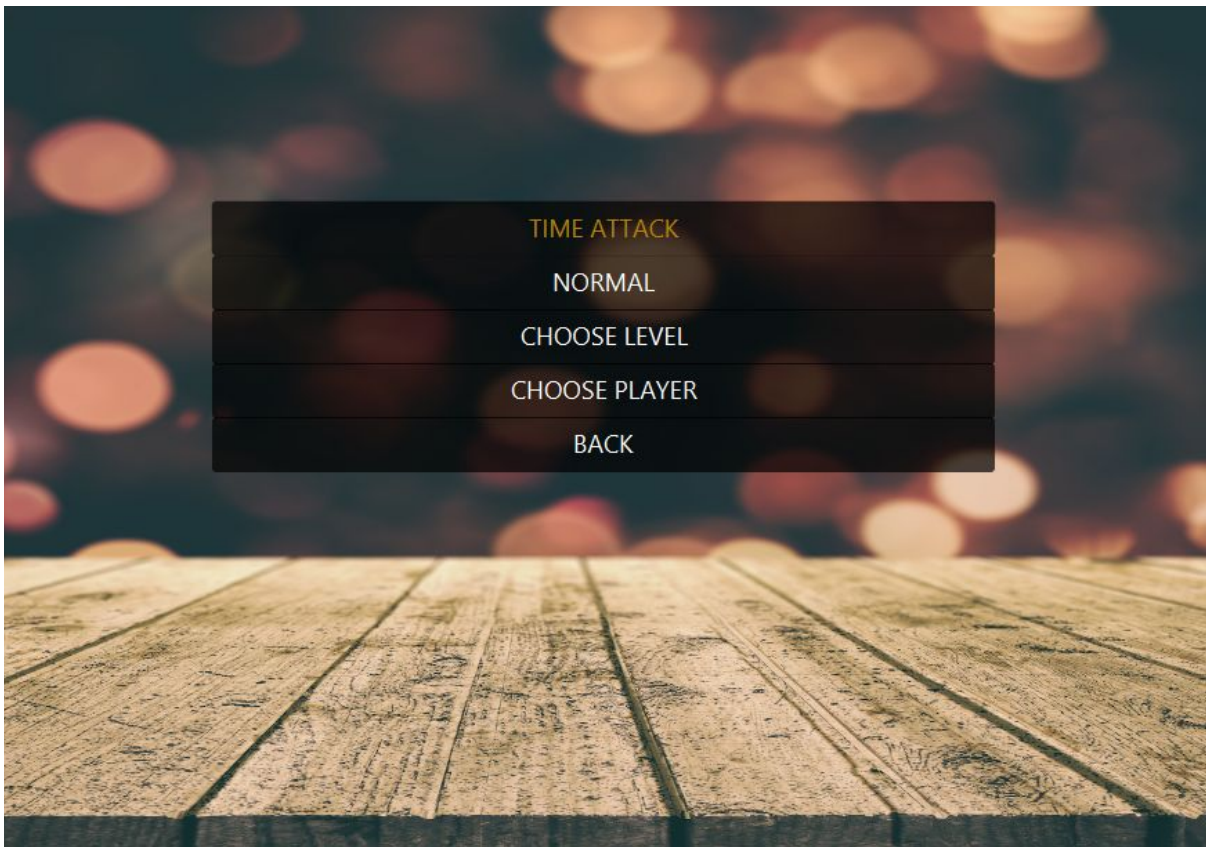
SNAPSHOTS

Main menu:

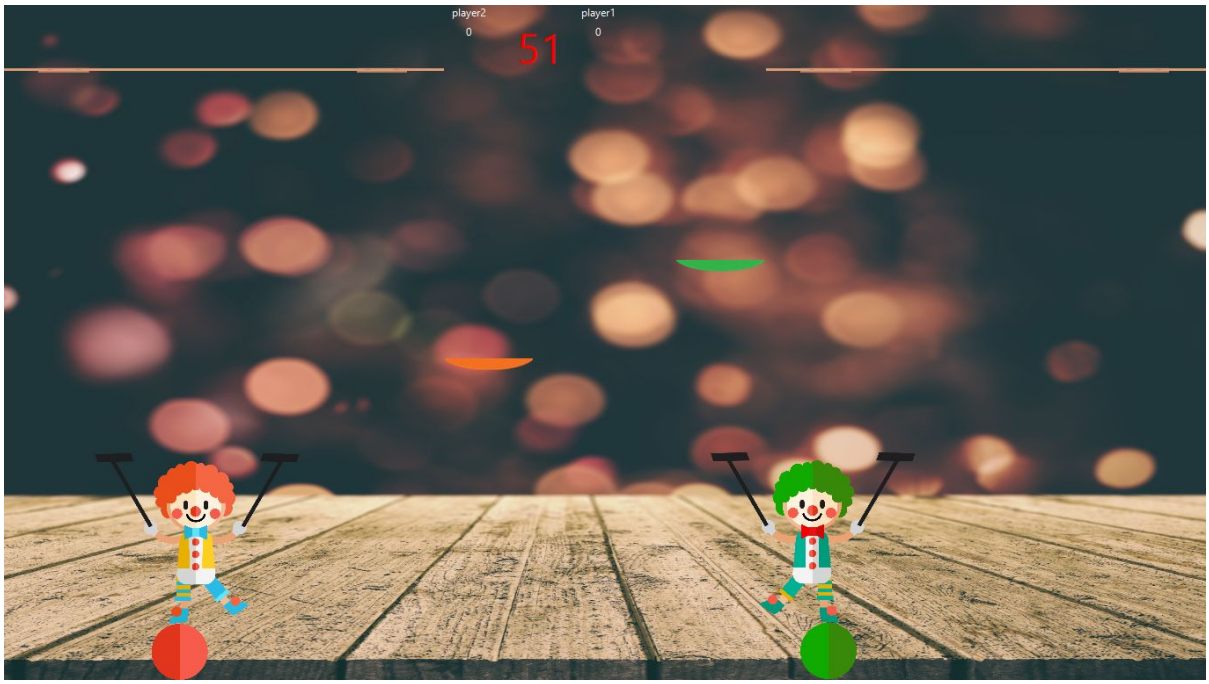


PlayerChooser:

Game Modes:



Gameplay:


player2
0
player1
0
51


player2
0
player1
0
45