Exercise 1 - The core

1. Responsibility driven design

Based on the Must Haves in the requirements documents V2.1, and using responsibility driven design, the game would have the following classes (described in a CRC fashion):

Class	**Responsibility**	**Collaborations**
GameScreen	Manages the execution order of other classes, defines the play area	All other classes
Ball	Contains and execute the basic parameters and functionality related to balls	-
BallGraph	Stores the location and color of each ball, detects what balls need to pop/remain.	Ball, LevelLoading
BallManager	Manages ball actions: bouncing, attaching to the top, popping animation.	Ball, GameScreen
Cannon	Manages and executes canon-related functionality and parameters	Ball, InputHandler
ScoreKeeping	Manages the score related issues: score reset, incrementation, and possibly high score storage later on.	BallGraph
TimeKeeping	Keeps track of the game time, specifically to satisfy the ball shooting within 10 seconds requirement.	GameScreen, Ball.
LevelLoading	Generate the random gameplay levels.	Ball
InputHandler	Detects the user Input, initiates performing the required task	-
AssetLoader	Responsible for loading the sprites used for game animation	-

Deriving classes involved following the processes Dr. Alberto explained in the first class that included writing a description of the game, highlighting important characteristics and deriving classes accordingly. This was for sure done while examining the requirements documents and ensuring the coverage of all key points.

The classes derived above are quite similar to the classes used in the actual implementation of the game, as that exact process was the one followed at the beginning of designing this game. However, extra classes exist as an extension to the key functionality described in the CRC sheet above. This is due to also having extra add-on functionalities in addition to the main required ones. Extra classes are:

- MainMenuScreen: a main menu screen.
- YouLoseScreen: displays when the player loses.
- YouWinScreen: displays when the user wins.
- Config: used to hold key parameters for settings in the game e.g. screen size, ball size.
- AudioManager: a class that control the audio features running in the game.

2. *Following the Responsibility Driven Design, describe the main classes you implemented in your project in terms of responsibilities and collaborations*

- Responsibilities : A responsibility is an obligation to perform a task or know information. These are further categorized according to their usage scenario.
- Collaborations : A collaboration is defined as an interaction of objects or roles (or both)

Main Class	**Responsibility**	**Collaborations**
BustaMove	1. Initializes sprite batch	
GameScreen	1. Manages all the game related components	1. BallManager 2. Cannon 3. InputHandler
BallManager	1. Manages the balls in the game	1. Ball 2. Cannon 3. BallGraph
Ball	1. Represent a ball 2. Save all the properties like speed, color and position	
Cannon	1. Manages the angle of the cannon. 2. Manages the angle the ball should be shot in	
BallGraph	Keep track of all the connections between the balls	

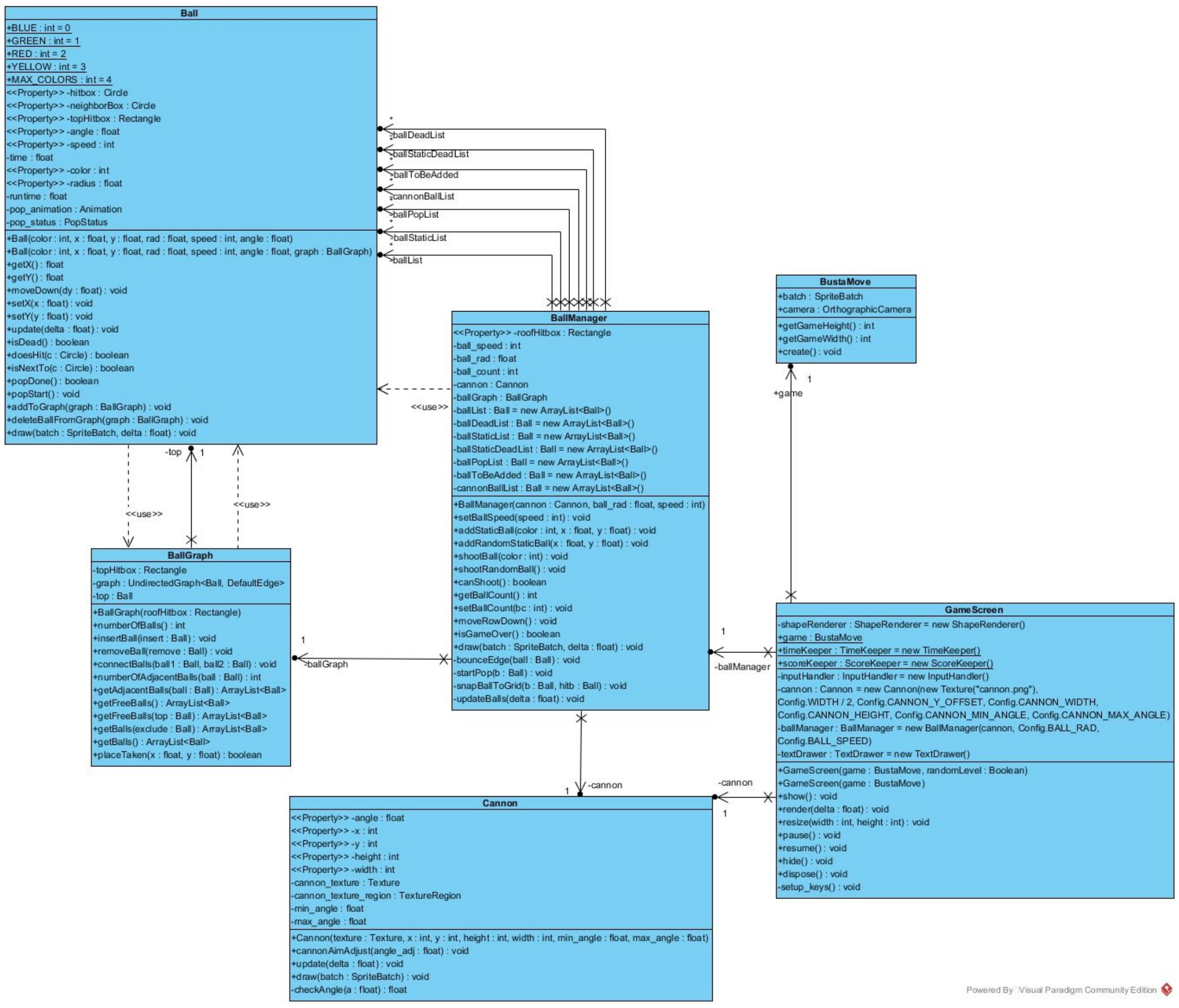
3. Why do you consider the other classes as less important? Following the Responsibility Driven Design, reflect if some of those non-main classes have similar/little responsibility and could be changed, merged, or removed. If so, perform the code changes; if not, explain why you need them.

The classes that we consider less important than the classes listed in the table above, are located in the package helpers. We consider them less important because they just perform tasks for the main classes. The helper classes do not really determine *how* the game is played.

All of the helper classes have their own responsibility. Their responsibility can be derived from their names. Some have little responsibility like the TextDrawer, but its responsibility is totally different from the other classes. Therefore we think it is not a good idea to merge the classes. We thought the responsibilities of the classes well trough when creating them.

Some classes might be a candidate to merge if we look at their responsibility, like the BallManager and the BallGraph. They both keep track of all the balls, but if we look at the size of the classes it’s better to keep them separate otherwise the BallManager class would become very large which will make the class less maintainable. Based on the previous analysis, no classes have been merged or changed.

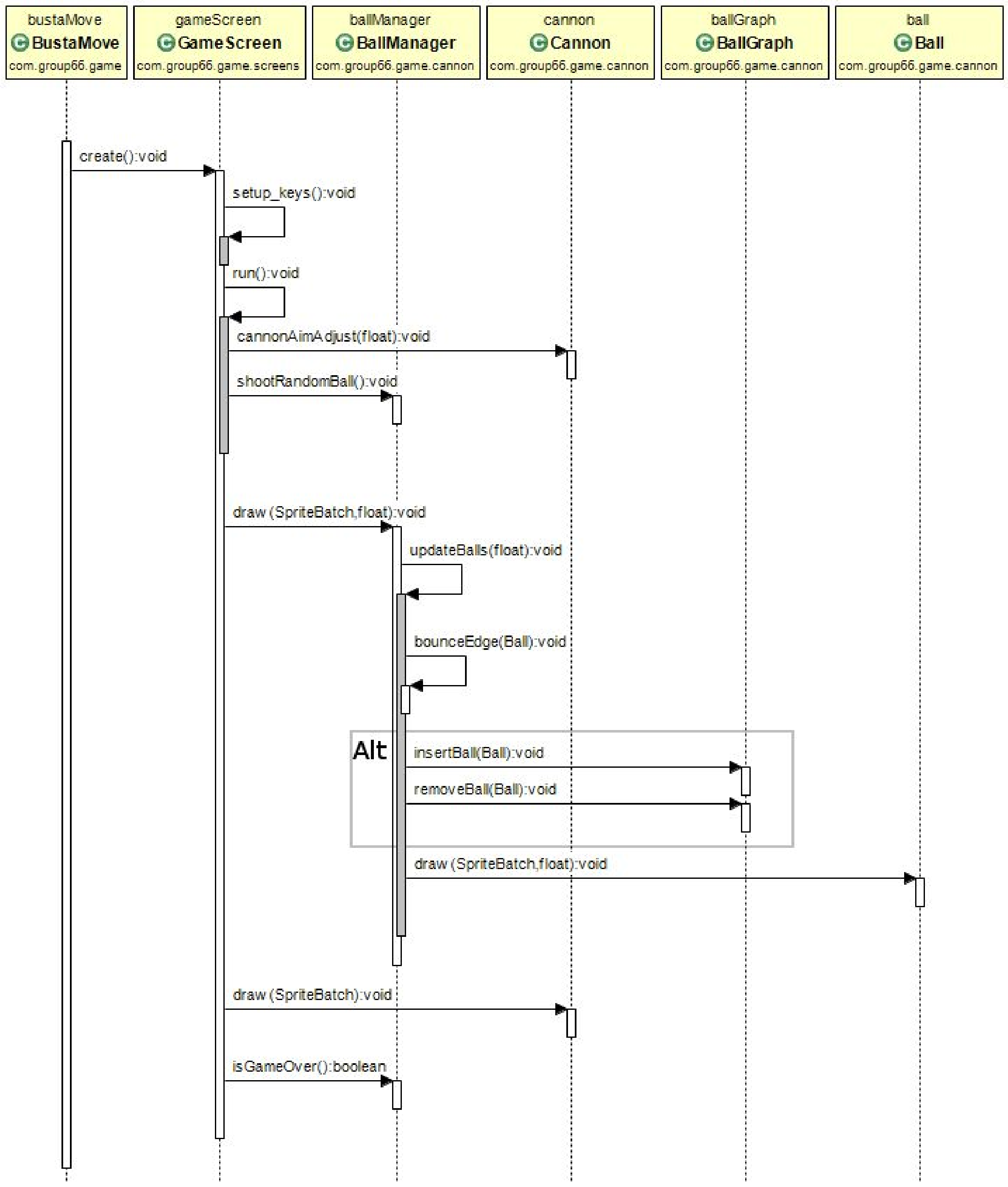
4. Draw the class diagram of the aforementioned main elements of your game (do not forget to use elements such as parametrized classes or association constraints, if necessary).


Ball
+BLUE : int = 0
+GREEN : int = 1
+RED : int = 2
+YELLOW : int = 3
+MAX_COLORS : int = 4
<<Property>> -hitbox : Circle
<<Property>> -neighborBox : Circle
<<Property>> -topHitbox : Rectangle
<<Property>> -angle : float
<<Property>> -speed : int
-time : float
<<Property>> -color : int
<<Property>> -radius : float
-runtime : float
-pop_animation : Animation
-pop_status : PopStatus
+Ball(color : int, x : float, y : float, rad : float, speed : int, angle : float)
+Ball(color : int, x : float, y : float, rad : float, speed : int, angle : float, graph : BallGraph)
+getX() : float
+getY() : float
+moveDown(dy : float) : void
+setX(x : float) : void
+setY(y : float) : void
+update(delta : float) : void
+isDead() : boolean
+doesHit(c : Circle) : boolean
+isNextTo(c : Circle) : boolean
+popDone() : boolean
+popStart() : void
+addToGraph(graph : BallGraph) : void
+deleteBallFromGraph(graph : BallGraph) : void
+draw(batch : SpriteBatch, delta : float) : void
-ballDeadList
-ballStaticDeadList
-ballToBeAdded
-cannonBallList
-ballPopList
-ballStaticList
-ballList
<<use>>
-top
1
<<use>>
<<use>>
BallManager
<<Property>> -roofHitbox : Rectangle
-ball_speed : int
-ball_rad : float
-ball_count : int
-cannon : Cannon
-ballGraph : BallGraph
-ballList : Ball = new ArrayList<Ball>()
-ballDeadList : Ball = new ArrayList<Ball>()
-ballStaticList : Ball = new ArrayList<Ball>()
-ballStaticDeadList : Ball = new ArrayList<Ball>()
-ballPopList : Ball = new ArrayList<Ball>()
-ballToBeAdded : Ball = new ArrayList<Ball>()
-cannonBallList : Ball = new ArrayList<Ball>()
+BallManager(cannon : Cannon, ball_rad : float, speed : int)
+setBallSpeed(speed : int) : void
+addStaticBall(color : int, x : float, y : float) : void
+addRandomStaticBall(x : float, y : float) : void
+shootBall(color : int) : void
+shootRandomBall() : void
+canShoot() : boolean
+getBallCount() : int
+setBallCount(bc : int) : void
+moveRowDown() : void
+isGameOver() : boolean
+draw(batch : SpriteBatch, delta : float) : void
-bounceEdge(ball : Ball) : void
-startPop(b : Ball) : void
-snapBallToGrid(b : Ball, hitb : Ball) : void
-updateBalls(delta : float) : void
BustaMove
+batch : SpriteBatch
+camera : OrthographicCamera
+getGameHeight() : int
+getGameWidth() : int
+create() : void
1
+game
BallGraph
-topHitbox : Rectangle
-graph : UndirectedGraph<Ball, DefaultEdge>
-top : Ball
+BallGraph(roofHitbox : Rectangle)
+numberOfBalls() : int
+insertBall(insert : Ball) : void
+removeBall(remove : Ball) : void
+connectBalls(ball1 : Ball, ball2 : Ball) : void
+numberOfAdjacentBalls(ball : Ball) : int
+getAdjacentBalls(ball : Ball) : ArrayList<Ball>
+getFreeBalls() : ArrayList<Ball>
+getFreeBalls(top : Ball) : ArrayList<Ball>
+getBalls(exclude : Ball) : ArrayList<Ball>
+getBalls() : ArrayList<Ball>
+placeTaken(x : float, y : float) : boolean
1
-ballGraph
1
-ballManager
GameScreen
-shapeRenderer : ShapeRenderer = new ShapeRenderer()
+game : BustaMove
+timeKeeper : TimeKeeper = new TimeKeeper()
+scoreKeeper : ScoreKeeper = new ScoreKeeper()
-inputHandler : InputHandler = new InputHandler()
-cannon : Cannon = new Cannon(new Texture("cannon.png"), Config.WIDTH / 2, Config.CANNON_Y_OFFSET, Config.CANNON_WIDTH, Config.CANNON_HEIGHT, Config.CANNON_MIN_ANGLE, Config.CANNON_MAX_ANGLE)
-ballManager : BallManager = new BallManager(cannon, Config.BALL_RAD, Config.BALL_SPEED)
-textDrawer : TextDrawer = new TextDrawer()
+GameScreen(game : BustaMove, randomLevel : Boolean)
+GameScreen(game : BustaMove)
+show() : void
+render(delta : float) : void
+resize(width : int, height : int) : void
+pause() : void
+resume() : void
+hide() : void
+dispose() : void
-setup_keys() : void
1
-cannon
-cannon
1
Cannon
<<Property>> -angle : float
<<Property>> -x : int
<<Property>> -y : int
<<Property>> -height : int
<<Property>> -width : int
-cannon_texture : Texture
-cannon_texture_region : TextureRegion
-min_angle : float
-max_angle : float
+Cannon(texture : Texture, x : int, y : int, height : int, width : int, min_angle : float, max_angle : float)
+cannonAimAdjust(angle_adj : float) : void
+update(delta : float) : void
+draw(batch : SpriteBatch) : void
-checkAngle(a : float) : float
Powered By Visual Paradigm Community Edition

5. Draw the sequence diagram to describe how the main elements of your game interact (consider asynchrony and constraints, if necessary).

The sequence diagram depicted below shows the sequence of the main classes. First the BustaMove Class initiates the game. The main Bust-A-Move game is run and managed within the *GameScreen*. Here game related classes such as the *BallManager* and *Cannon* are created. Within the *GameScreen* the function *setup_keys()* is called, which will map the keys used in the game. After this, *run()* is called, which will check the keys and adjust the angle of the *Cannon* or shoot a ball using the *BallManager* if needed.
The *draw()* function is called within *GameScreen* by the underlying graphics library LibGDX. This is called whenever a new frame has to be rendered.
When this is called, we in turn call the *draw()* function of the *BallManager*. This will update the ball locations using *updateBalls()*. Within *updateBalls()* it's checked if a ball which is moving is hitting the edge and should bounce. Next it checks whether a ball should be inserted or not using *insertBall* or *removeBall* which are present within the *BallGraph* class. Lastly the *BallManager* calls the *draw()* of the managed *Ball* classes, this draws the balls. After we are done running the *draw()* of the *BallManager*, we call the *draw()* of the *Cannon* class, this will draw the cannon.

The final check is whether the game is over using the *isGameOver* function present in the *BallManager* class.



The sequence diagram for the main components of the game.

Exercise 2 - UML in practice

1. What is the difference between aggregation and composition? Where are composition and aggregation used in your project? Describe the classes and explain how these associations work

Aggregation:
An aggregation link is a relation between objects that means that *Class-A*, although used by *Class-B*, is not the owner or exclusive user of *Class-B*. This also means that deletion of *Class-A*, does not imply the deletion of *Class-B* which is linked by the aggregation link. There is no parent-child relation.

Composition:
A composition link between classes states that *Class-A* has a strong lifecycle dependency with *Class-B*. If *Class-A* is deleted, *Class-B* will also be deleted. It also shows that *Class-A* has exclusive ownership over (at least parts) of *Class-B*. This is a parent-child relationship.

Example of aggregation in the project:

- Objects of typeBall are aggregated to create the ball grid that the player shoots at, and that is managed by the BallManager. In this relation, every Ball is an objectthat's capable of existing on its own, but the aggregation of many balls creates the grid.

Examples of composition in the project:

- The *GameScreen* class creates the *BallManager* and *Cannon* classes. When the *GameScreen* is destroyed, there is no need for the *BallManager* and *Cannon*, so they are also destroyed.
- The BallManager class creates array lists -which are objects of the ArrayList class embedded in Java- to manage the condition of the balls. The composition of these lists is what creates the BallManager and allows it to have its functionality.

2. Is there any parametrized class in your source code? If so, describe which classes, why they are parametrized, and the benefits of the parametrization. If not, describe when and why you should use parameterized classes in your UML diagrams.

In our source code there are no parametrized classes. We do make use of some parameterized classes like the ArrayList, but we have not implemented one ourselves in the source code.

We could implement the BallGraph as an parameterized class in our project in that way we are also able the make the game with other objects than balls, for example Doritos chips. The rest of the classes do not take attributes of that type. Also functionalities are not implemented twice because it is applied to another datatype. So there are no advantages of parameterizing classes.

We do not have the ambition to make the game also with other objects like squares or Doritos Chips, so we don’t think we should parameterize any of our classes.

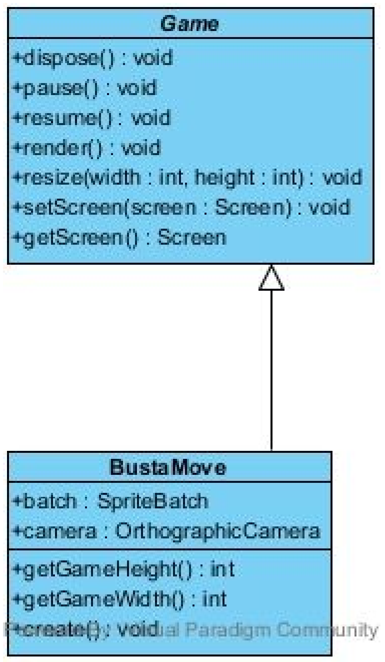
3. Draw the class diagrams for all the hierarchies in your source code. Explain why you created these hierarchies and classify their type (e.g., “Is-a” and “Polymorphism”). Considering the lectures, are there hierarchies that should be removed? Explain and implement any necessary change.

Since there is currently no hierarchies in the source code, there is no UML diagram that can be drawn. We have all classes separated from each other based on their responsibilities.

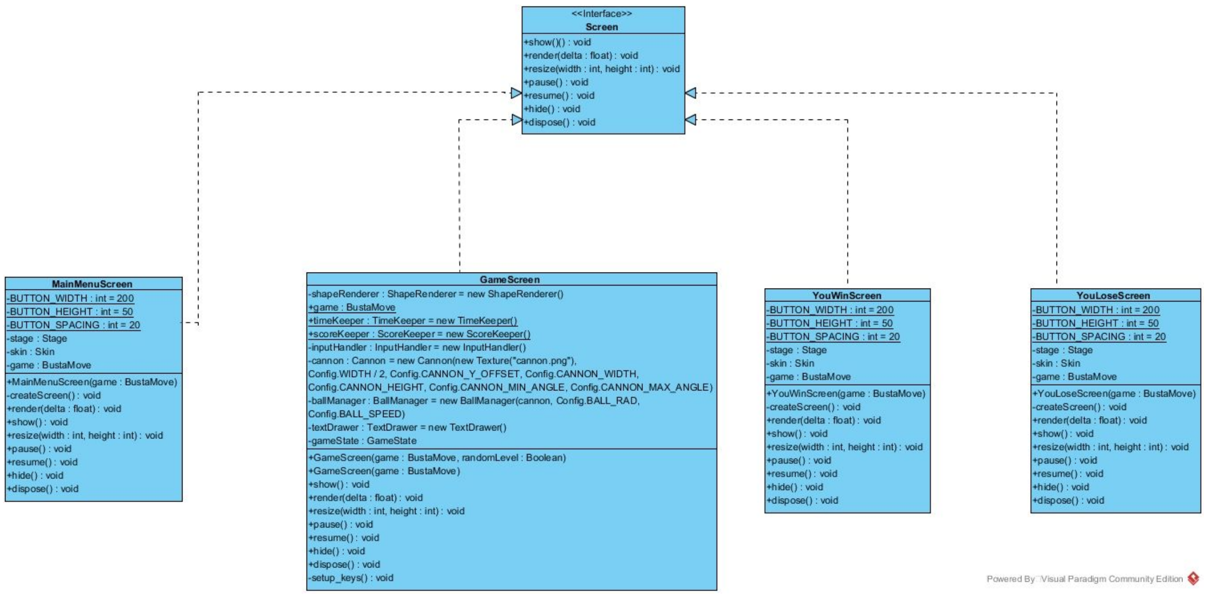
The reason we do not have any case of hierarchy in the code is that there are not many objects in the game that are similar. The only object that is the same is the object ball. The only difference between the balls is the color, that is currently implemented as a variable in the class instead of making an hierarchy with an interface or an abstract class.

Although there is no hierarchy in the written code, there is some kind of hierarchy in the whole project. By using the libgdx library for making the game, the source code implements and extends some interfaces and abstract classes from the library.

These relations are shown in the following UML diagram:



This diagram shows the relation between the abstract class Game of the LibGDX library and the implementation in the source code BustaMove. This has been done so the library will work correctly with the game.



<<Interface>>
Screen
+show()() : void
+render(delta : float) : void
+resize(width : int, height : int) : void
+pause() : void
+resume() : void
+hide() : void
+dispose() : void
MainMenuScreen
-BUTTON_WIDTH : int = 200
-BUTTON_HEIGHT : int = 50
-BUTTON_SPACING : int = 20
-stage : Stage
-skin : Skin
-game : BustaMove
+MainMenuScreen(game : BustaMove)
-createScreen() : void
+render(delta : float) : void
+show() : void
+resize(width : int, height : int) : void
+pause() : void
+resume() : void
+hide() : void
+dispose() : void
GameScreen
-shapeRenderer : ShapeRenderer = new ShapeRenderer()
+game : BustaMove
+timeKeeper : TimeKeeper = new TimeKeeper()
+scoreKeeper : ScoreKeeper = new ScoreKeeper()
-inputHandler : InputHandler = new InputHandler()
-cannon : Cannon = new Cannon(new Texture("cannon.png"), Config.WIDTH / 2, Config.CANNON_Y_OFFSET, Config.CANNON_WIDTH, Config.CANNON_HEIGHT, Config.CANNON_MIN_ANGLE, Config.CANNON_MAX_ANGLE)
-ballManager : BallManager = new BallManager(cannon, Config.BALL_RAD, Config.BALL_SPEED)
-textDrawer : TextDrawer = new TextDrawer()
-gameState : GameState
+GameScreen(game : BustaMove, randomLevel : Boolean)
+GameScreen(game : BustaMove)
+show() : void
+render(delta : float) : void
+resize(width : int, height : int) : void
+pause() : void
+resume() : void
+hide() : void
+dispose() : void
-setup_keys() : void
YouWinScreen
-BUTTON_WIDTH : int = 200
-BUTTON_HEIGHT : int = 50
-BUTTON_SPACING : int = 20
-stage : Stage
-skin : Skin
-game : BustaMove
+YouWinScreen(game : BustaMove)
-createScreen() : void
+render(delta : float) : void
+show() : void
+resize(width : int, height : int) : void
+pause() : void
+resume() : void
+hide() : void
+dispose() : void
YouLoseScreen
-BUTTON_WIDTH : int = 200
-BUTTON_HEIGHT : int = 50
-BUTTON_SPACING : int = 20
-stage : Stage
-skin : Skin
-game : BustaMove
+YouLoseScreen(game : BustaMove)
-createScreen() : void
+render(delta : float) : void
+show() : void
+resize(width : int, height : int) : void
+pause() : void
+resume() : void
+hide() : void
+dispose() : void
Powered By Visual Paradigm Community Edition

In this diagram the screens of the game are shown. They are all implemented according to the interface the LibGDX library provides for creating screens. In this way it is possible to switch between the screens by calling library functions.

Exercise 3 - Simple logging

1. Extend your implementation of the game to support logging. The game has to log all the actions happened during the game (e.g., player moved Tetris piece from position X to position Y). The logging has to be implemented from scratch without using any existing logging library. Define your requirements and get them approved by your teaching assistant.

Logger related requirements can be found within the logger directory.

2. During the analysis and design phases of this extension use responsibility driven design and UML (push to the repository a single PDF file including all the documents produced)

Logger related design can be found within the logger directory