Group 7

Assignment 1

Authors:
Matthijs Verzijl - 4282604
Emilie de Bree - 4247558
Jim Hommes - 4306090
Toine Hartman - 4305655
Jeffrey Helgers - 4318749

Supervisor:
Alberto Bacchelli

Teaching Assistant:
Aaron Ang

TU Delft Delft University of Technology

October 6, 2015

Contents

1 The Core **1**
1.1 Question 1 . . . 1
1.2 Question 2 . . . 4
1.3 Question 3 . . . 5
1.4 Question 4 . . . 6
1.5 Question 5 . . . 7

2 UML in practice **8**
2.1 Question 1 . . . 8
2.2 Question 2 . . . 8
2.3 Question 3 . . . 9

3 Simple logging **11**
3.1 Question 1 . . . 11
3.2 Question 2 . . . 11

1. The Core

1.1 Question 1

"Following the Responsibility Driven Design, start from your requirements (without considering your implementation) and derive classes, responsibilities, and collaborations (use CRC cards). Describe each step you make. Compare the result with your actual implementation and discuss any difference (e.g., additional and missing classes)."

As a recap we will start by providing our requirements:

- The game shall have a single level to play.
- The game shall show a game over screen when the player dies.
- The game shall have a field where the character and monsters can move around.
- The player shall have the ability to win the game when they have killed all the monsters.
- The player shall lose the game when a monster touches the character.
- The player's character must be able to move around horizontally.
- The player's character must be able to jump, both straight up and diagonally.
- The player's character shall be able to shoot bubbles.
- The monsters shall walk around horizontally, and when they hit a wall they will change direction.
- A monster shall be trapped in a bubble when the player shoots a bubble at it.
- A monsters shall die when it is caught in a bubble and the character collides with that bubble.
- The game shall have a start up screen.
- The game shall have multiple levels to play on.
- The game shall keep a track of the player's score.
- The game shall have the ability to pause.
- The game shall have the ability to unpause after the pause button have been pressed.
- The game shall have the ability to stop, which closes the game.
- The game shall have a help screen that will inform the player about the controls and powerups.
- The player shall have the ability to start a new game.
- The player shall have multiple lives.
- The player shall lose a life, when a monster touches the character.
- The monsters shall be able to drop items when they die.
- The player shall have the ability to pick up items which will increase their score.
- The powerups shall increase the speed of the bubble, the distance the bubble travels or the walking speed of the character.

Next step is to derive all necessary classes from the requirements. Useful to do this is by selecting every noun in the requirements.

- LevelController
- Level
- Wall
- Character
- Sprite
- Monster
- Input
- Bubble
- PowerUp

Since we now have all classes present, we can start constructing our CRC cards

LevelController	
Superclass(es):	
Subclass(es): Level, Player	
createLvl	Level
createPlayer	Player
startLvl	Level
winLvl	Level

Table 1.1: CRC card: LevelController

Level	
Superclass(es):	
Subclass(es): Wall	
initiateLevel	Level
initiateWalls	Wall
initiateSprite	Sprite
initiateMonsters	Monster

Table 1.2: CRC card: Level

Wall	
Superclass(es):	
Subclass(es):	
setX	Integer
setY	Integer

Table 1.3: CRC card: Wall

Sprite	
Superclass(es):	
Subclass(es): Character, Monster	
get/setX	Integer
get/setY	Integer
get/setR	Integer
get/setDx	Integer
get/setDy	Integer
get/setDr	Integer
get/setCanMove	Boolean
get/setImagePath	String
get/setWidth	Integer
get/setHeight	Integer

Table 1.4: CRC card: Sprite

Character	
Superclass(es): Sprite	
Subclass(es): Bubble, Input	
moveLeft	Input
moveRight	Input
fireWeapon	Input
die	-
isDead	Boolean

Table 1.5: CRC card: Character

Monster	
Superclass(es): Sprite	
Subclass(es):	
get/setSpeed	Integer
checkCollision	-
get/setDirection	String
isCaught	Boolean
isDead	Boolean

Table 1.6: CRC card: Monster

Input	
Superclass(es):	
Subclass(es):	
moveUp	-
moveDown	-
moveLeft	-
moveRight	-
fireWeapon	-

Table 1.7: CRC card: Input

Bubble	
Superclass(es):	
Subclass(es):	
ableToCatch	-
move	-

Table 1.8: CRC card: Bubble

PowerUp	
Superclass(es):	
Subclass(es):	
get/setAbleToPickup	Boolean

Table 1.9: CRC card: PowerUp

When comparing this with our actual implementation we notice a couple things:

- In our actual implementation, the classes Monster and Sprite have their own superclasses.
- The class PowerUp hasnt́ been implemented yet.
- Some of the classes are named differently, which can make easy explaining of classes harder.
- The class Sprite (our called Player in the implementation), has more methods, cause there are multiple variables defined which we didn't think of in the CRC cards. Examples are a counter variable to count the bubbles, an array of all the bubbles a player fired, a variable to check whether the player can jump.

1.2 Question 2

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

Launcher

Responsibility:
The Launcher launches the game by loading the StartorEndController.
Collaborations:
The Launcher collaborates with the StartorEndController.

StartorEndController

Responsibility:
The StartorEndController starts the game, or ends it.
Collaborations:
The StartorEndController collaborates with LevelController toe load the Level which will be placed.

LevelController

Responsibility:
The LevelController-class is responsible for handling interaction between a Level and Spritebases. It is also runs the gameLoop which renders the game every frame. therefore it is also able to pause the game, win a game and go to a next Level.
Collaborations:
The LevelController-class collaborates with a Level and the SpriteBase.

ScreenController

Responsibility:
The ScreenController is responsible for drawing everything on a screen.
Collaborations:
The ScreenController works together with the LevelController as well as SpriteBase to draw every sprite on the screen.

Level

Responsibility:
The Level-class is responsible for constructing the level a player plays. A level has to be created from loading a text-file in which is defined where to place the walls and the monsters.

Collaborations:
The Level-class collaborates with SpriteBase to display every element in the Level.

SpriteBase

Responsibility:
Every element in Level has certain standard attributes such as an x,y coordinate, but also the speed in every direction and an image. All of these attributes are defined in SpriteBase.
Collaborations:
Depending on the kind of SpriteBase, the player must be able to perform input. therefore the SpriteBase collaborates with Input.

Input

Responsibility:
The Input class catches all input performed by the player. The class makes sure that when a player presses one of the keys allowed the desired action is performed and when the key is released the action stops.
Collaborations:
The only object which can be moved by the player is the Spritebase which has the property of being the played character. therefore this is the only class with which the Input class collaborates.

Non-main classes

The classes GravityObject, Bubble, Wall, Monster, Player and Walker are all (direct or indirect) extensions of SpriteBase. therefore they can be removed. Settings is a class with only static information and can therefore also be merged in a controller.

1.3 Question 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 are considered less important are:

- GameOverController
- WinController
- Settings
- Player
- Monster
- Walker
- GravityObject
- Bubble
- Wall

The GameOverController and WinController are important because they control the screen when you win or lose, But these classes are almost the same. They both create a screen with a ”Play Again” and a ”Quit button”. The only differences between these classes are the picture on the screen and the way they are called. The WinController is called when all the monsters are killed, and the GameOverController when the character touches a monster. We have removed these two classes, and created one class new class: StartOrEndController. This controller creates a screen with the two buttons, with a different image when you win or lose.

The class Settings is a class that contains information about the game, like speed of monsters, bubbles and the player. This class makes the programming more clear.

The other classes are all extensions from SpriteBase. All these classes are necessary to run the program correctly. The classes GravityObject, Bubble and Wall are direct extensions from SpriteBase.

The GravityObjects are all the objects that can move in different directions in the level.

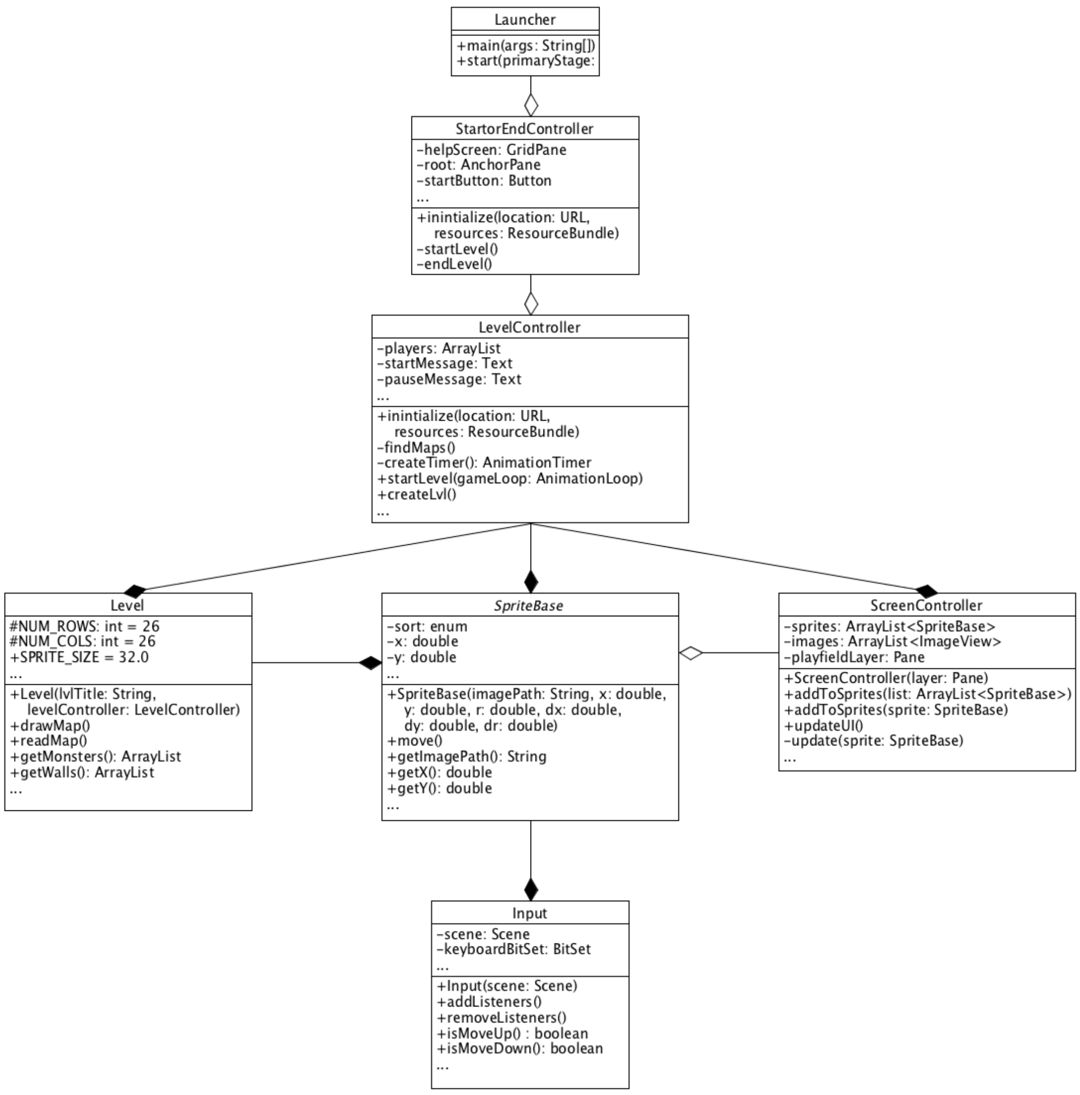
The Walls are the objects that can't collide with other sprites.

The Bubbles are the bubbles that move in a specific direction, a few steps to the left or right and then they move up. These bubbles are shot by a player and can catch the monsters.

The classes Monster and Player are extensions from GravityObject. The difference between these classes is that the monsters move around freely, and the player moves around by key presses. At the moment is it possible to remove the class Monster, because there is only one extension from this class: Walker, which represent the monsters that walk around the field. But we won't remove the class Monster because in the future we may want to create different types of monster, for instance monsters that fly or can shoot.

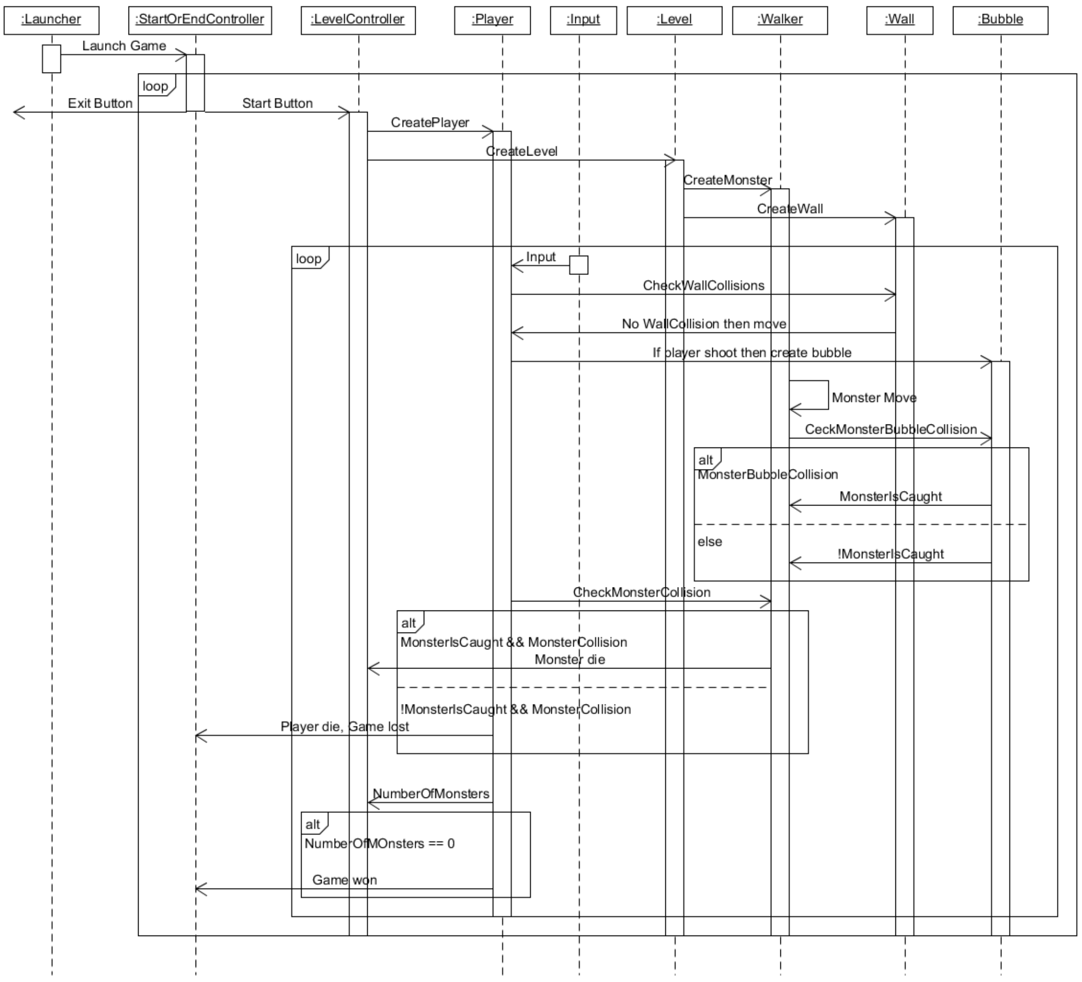
1.4 Question 4

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



1.5 Question 5

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



2. UML in practice

2.1 Question 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."
The difference between aggregation and composition is that in composition class A owns class B (or contains). While in aggregation, class A uses class B (or consists of), so the classes are independent of each other.

In our project there are both examples of aggregation as well as composition, as listed below:

- Aggregation can be found between:
  - **Launcher** and **StartController**; Launcher uses the StartController to be able to start the game.
  - **StartOrEndController** and **StartController**; StartOrEndController uses the StartController to be able to start the game again.
  - **StartController** and **LevelController**; StartController uses the LevelController to be able to create the level.
  - **LevelController** and **StartOrEndCOntroller**; LevelController uses the StartOrEndController to be able to restart the game after the player has won or lost.
  - **ScreenController** and **SpriteBase**; ScreenController uses the SpriteBase to be able to create the sprites on the screen, when playing the game.
  - **Level** and **Settings**; Level uses the Settings to get measurements for the level.
  - **Bubble** and **Settings**; Bubble uses the Settings to get the measurements for the bubbles that will be shot.
  - **Player** and **Settings**; Player uses the Settings to get the measurements for the character that the player will be playing.
- Composition can be found between:
  - **LevelController** and **ScreenController**; LevelController can not exist without the use of the ScreenController as the ScreenController controls what happens on the screen during a level.
  - **LevelController** and **SpriteBase**; LevelController owns SpriteBase because without SpriteBase the levels can not be created to be controller by the controller.
  - **LevelController** and **Level**; LevelController owns Level because without the Level class the LevelController can not create the levels for the game to be played on.
  - **Level** and **SpriteBase**; Level contains SpriteBase because without SpriteBase the levels can not be created as the SpriteBase is used to create all the sprites in a level.
  - **Player** and **Input**; Player owns Input because without Input the player can not exist to be played.

2.2 Question 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 parametrized classes in your UML diagrams."

In our project we haven't implemented any parameterised classes, but we do use a parameterised class from java.util, the ArrayList. With the parameterised class "ArrayList" from java, you can create a list of the object that is defined in in the parameter.

When you write a class definition and you don't know with which types the class will be used, then you should use a parameterised class. With the parameter you can define the type on which the class will operate.

A parameterised class should be in the UML diagram when you have implemented a parameterised class in your source code. The paramaterised class from java.util that we used is also visible in the UML diagram, but not as a parameterised class, as a attribute from a different class.

The UML diagram shows a parameterised class with a dotted box over the right corner of a normal class. In this box goes the parameter for that specific class, it is possible to have more than one parameter.

2.3 Question 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."

The hierarchies that can be seen in the class diagram in figure 2.1, have been chosen to stop the code from becoming too complex, and to allow both the programmers and developers, as well as anyone who would be looking at the code to have a better understanding of how the code is set up. The hierarchies were also created in this way, to insure that no class would get too long.

The following hierarchies can be seen in the class diagram:

- There is Polymorphism between SpriteBase (the super class) and its three sub classes GravityObject, Bubble and Wall.
- There is Polymorphism between GravityObject (the super class) and its two sub classes Monster and Player.
- There is Is-a between Monster (the super class) and its sub class Walker.

The hierarchies work well because they aren't too complex and because they allow each of the components of the game to be created without having to rewrite code multiple times.

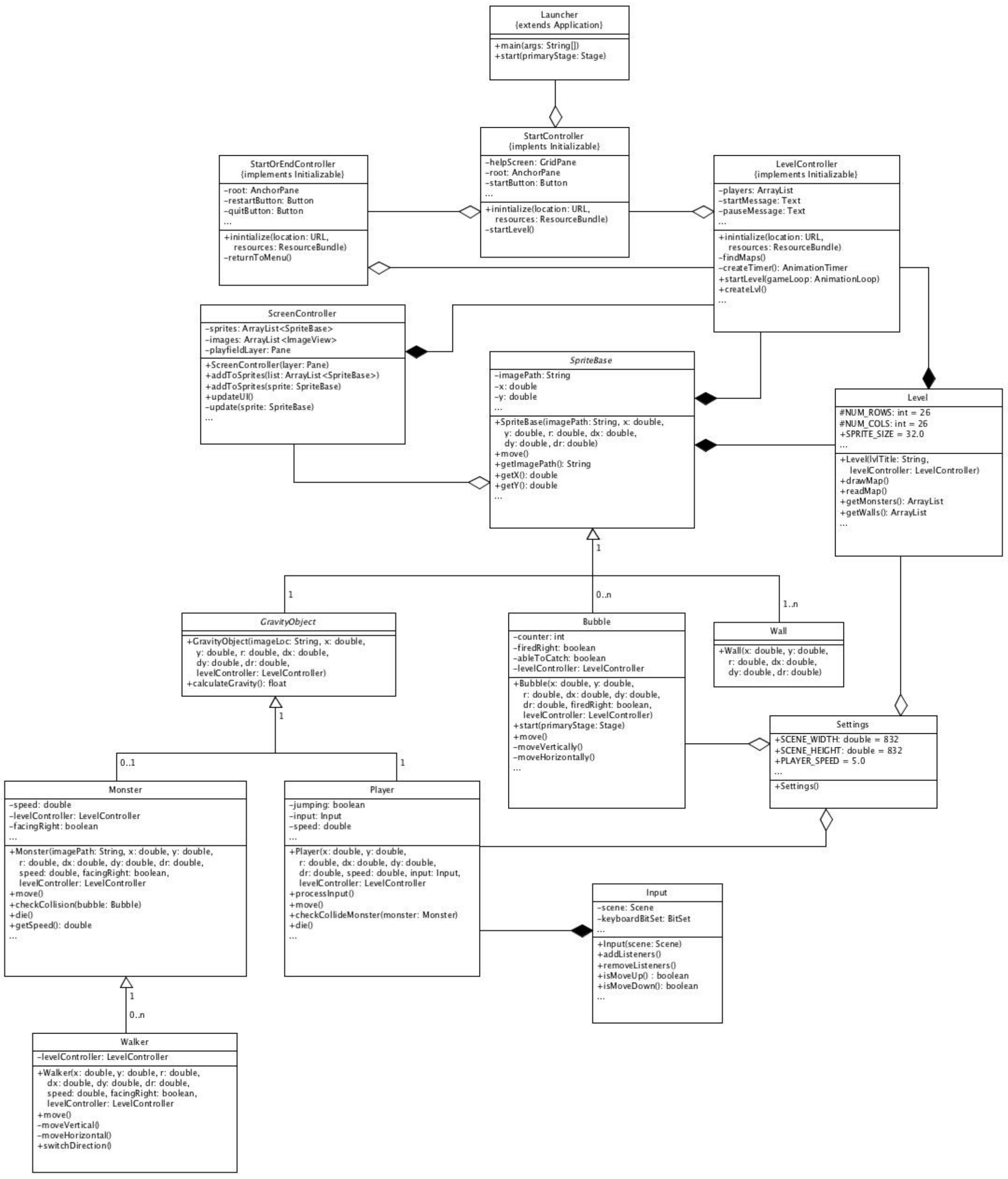

Launcher
{extends Application}
+main(args: String[])
+start(primaryStage: Stage)
StartController
{implents Initializable}
–helpScreen: GridPane
–root: AnchorPane
–startButton: Button
...
+inintialize(location: URL, resources: ResourceBundle)
–startLevel()
StartOrEndController
{implements Initializable}
–root: AnchorPane
–restartButton: Button
–quitButton: Button
...
+inintialize(location: URL, resources: ResourceBundle)
–returnToMenu()
LevelController
{implements Initializable}
–players: ArrayList
–startMessage: Text
–pauseMessage: Text
...
+inintialize(location: URL, resources: ResourceBundle)
–findMaps()
–createTimer(): AnimationTimer
+startLevel(gameLoop: AnimationLoop)
+createLvl()
...
ScreenController
–sprites: ArrayList<SpriteBase>
–images: ArrayList<ImageView>
–playfieldLayer: Pane
+ScreenController(layer: Pane)
+addToSprites(list: ArrayList<SpriteBase>)
+addToSprites(sprite: SpriteBase)
+updateUI()
–update(sprite: SpriteBase)
...
SpriteBase
–imagePath: String
–x: double
–y: double
...
+SpriteBase(imagePath: String, x: double, y: double, r: double, dx: double, dy: double, dr: double)
+move()
+getImagePath(): String
+getX(): double
+getY(): double
...
Level
#NUM_ROWS: int = 26
#NUM_COLS: int = 26
+SPRITE_SIZE = 32.0
...
+Level(lvlTitle: String, levelController: LevelController)
+drawMap()
+readMap()
+getMonsters(): ArrayList
+getWalls(): ArrayList
...
1
1
0..n
1..n
GravityObject
+GravityObject(imageLoc: String, x: double, y: double, r: double, dx: double, dy: double, dr: double, levelController: LevelController)
+calculateGravity(): float
Bubble
–counter: int
–firedRight: boolean
–ableToCatch: boolean
–levelController: LevelController
+Bubble(x: double, y: double, r: double, dx: double, dy: double, dr: double, firedRight: boolean, levelController: LevelController)
+start(primaryStage: Stage)
+move()
–moveVertically()
–moveHorizontally()
...
Wall
+Wall(x: double, y: double, r: double, dx: double, dy: double, dr: double)
Settings
+SCENE_WIDTH: double = 832
+SCENE_HEIGHT: double = 832
+PLAYER_SPEED = 5.0
...
+Settings()
1
0..1
1
Monster
–speed: double
–levelController: LevelController
–facingRight: boolean
...
+Monster(imagePath: String, x: double, y: double, r: double, dx: double, dy: double, dr: double, speed: double, facingRight: boolean, levelController: LevelController
+move()
+checkCollision(bubble: Bubble)
+die()
+getSpeed(): double
...
Player
–jumping: boolean
–input: Input
–speed: double
...
+Player(x: double, y: double, r: double, dx: double, dy: double, dr: double, speed: double, input: Input, levelController: LevelController
+processInput()
+move()
+checkCollideMonster(monster: Monster)
+die()
...
Input
–scene: Scene
–keyboardBitSet: BitSet
...
+Input(scene: Scene)
+addListeners()
+removeListeners()
+isMoveUp() : boolean
+isMoveDown(): boolean
...
1
0..n
Walker
–levelController: LevelController
+Walker(x: double, y: double, r: double, dx: double, dy: double, dr: double, speed: double, facingRight: boolean, levelController: LevelController
+move()
–moveVertical()
–moveHorizontal()
+switchDirection()

Figure 2.1: The class diagram for the whole project.

3. Simple logging

3.1 Question 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."

See the Logger Requirements document for the logger requirements.

3.2 Question 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)."

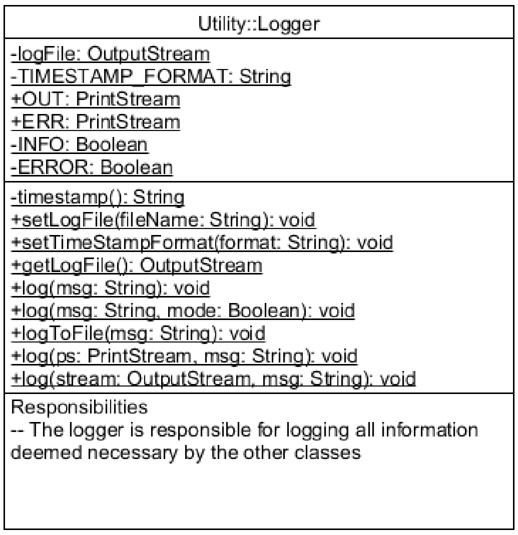
Introduction

Using Responsibility Driven Design and UML this project, it is only logical to use the documentation for such an addition. That is exactly what we did. To the Responsibility Driven Design document, the following should be added:

The Logger

Responsibility:
This class will make sure that other classes are able to output information through the logger.
Collaborations:
It should be flexible so that all the classes can collaborate with the Logger class.

In addition to the UML, the following class should be added:



```
Utility::Logger
-logFile: OutputStream
-TIMESTAMP_FORMAT: String
+OUT: PrintStream
+ERR: PrintStream
-INFO: Boolean
-ERROR: Boolean
-timestamp(): String
+setLogFile(fileName: String): void
+setTimeStampFormat(format: String): void
+getLogFile(): OutputStream
+log(msg: String): void
+log(msg: String, mode: Boolean): void
+logToFile(msg: String): void
+log(ps: PrintStream, msg: String): void
+log(stream: OutputStream, msg: String): void
Responsibilities
-- The logger is responsible for logging all information
deemed necessary by the other classes
```

All the classes which deem it necessary to log will aggregate (filled in diamond arrow) the Logger class.