Tommy Tran & Alonzo Contreras

December 21 2017

CSC 413-02 Team 7

GitHub Repos: https://github.com/CSC-413-SFSU-02/csc413-tankgame-team07

https://github.com/CSC-413-SFSU-02/csc413-secondgame-team07

Term Project

Tank Game & Lazarus

Introduction:

The term project assigned was to have students group up into pairs to design and create two Java game. Each group would build a Tank Wars style game and later create a second game based on the design and code of the previous. Doing so would teach each group about the process of designing and implementing a program that would need to be reused to build other programs. This would also be the first time that most students would be working with a partner while also using git. We were given about two months to finish the games and documentation.

Overview:

Each group was provided with game assets and the sample code from a similar game to base their work off of. During the design phase of building our game, we were highly advised to keep the concepts of polymorphism and inheritance in mind, while also basing our class hierarchy off of design patterns we learned in class. For both projects, we implemented the Observer pattern that was talked about in class. This involves Observer and Observable classes, with the Observables notifying the Observers when changes are made. We also used the pattern of the Model View Controller, which is when the view is the middle man between the model and the controller. This allows for the view to take in all the information from both and show updates/changes accordingly. Throughout the project we had six milestones:

Milestone One:

Create a class diagram for the tank game. Using both the Observer and MVC design pattern we turned in a diagram in which all the gameobjects would inherit a parent GameObject class. Info about the design is discussed further in this document.

Milestone Two:

The second milestone was to finish the tank game and submit the code to GitHub. We did not fulfill this milestone on time and received an N. However, we did pick up the pace and finished the tank game by the last milestone.

Milestone Three:

The third milestone that we fulfilled was making the class diagram for the second game. We chose to do Lazarus and turned in a class diagram that was similar to our tank game.

Milestone Four:

The fourth milestone was present our game(s) to the class. Presentations were cancelled.

Milestone Five:

Complete both games by 12/19/2017.

Milestone Six:

Complete documentation for the games.

Instructions:

Clone the repositories from GitHub and open in your preferred Java IDE. After successfully opening the repo as a project, press the play button.

For the tank game, WASD and space control player 1 on the left side of the screen. The arrow keys and the enter key control player 2 on the right side of the screen. Each tank has a health bar and three squares that display how many lives each tank has. When a player runs out of lives, a game over screen will appear to signify that the game is over.

For Getting Lazarus Out of the Pit, Left and Right on the arrow keys allows Lazarus to move about the game. Lazarus is only allowed to move to an open spot, on top of a box of height one; only allowed to jump one box high, not two. If a box lands on Lazarus, Lazarus is smashed and killed, ending the game. Boxes randomly spawn and fall directly over Lazarus, so the player has to strategically move about the map. The objective is to get the boxes to stack in order for Lazarus to escape the pit by pressing the stop button on either end of the map. Careful though, heavier boxes break lighter ones, so try to not get trapped, mwahahaha!

Scope of Work:

Tank Game

1.) GameWorld

The GameWorld class is responsible for creating instances of each game object, two controllers, and painting everything to the screen.

2.) Core Gameplay

The tanks need to be able to shoot each other until one explodes. Several game collisions must be handled in order for this to happen.

3.) Tank Rotation/Movement

Both tanks should be able to move 360 degrees and at the same time to provide a pleasant user experience. The movement should be responsible and smooth to reflect player input.

4.)MiniMap

A minimap of the gameworld was to be drawn in order for both players to get the scope of the game map.

5.)Destructible/Indestructible Walls

The Map has walls that are both breakable and unbreakable to deploy player strategy and to protect spawns.

6.)Animations

The tank rotations and game explosions should be shown smoothly and in such a way as to not slow down gameplay.

7.) Audio

Game music was to be implemented, and explosion sounds as well.

8.) Power Ups (Not completed)

Power Ups should populate the map to provide both players with a possible edge over the other.

Lazarus

1.) Game Map

Each object in the map was created and drawn to fit the size of the background.

2.) Game Walls

Lazarus and the random boxes dropping cannot go through these walls.

3.) Four boxes of different weights generated at random

There are four boxes in increasing weight: cardboard, wood, metal, and stone which are generated randomly and dropped at Lazarus X-location at the time of box creation.

4.) Player Controls

Only one player in this game and the controls are left and right.

5.) Lazarus rendered across map

When Lazarus moves he appears where he needs to be within the confines of the rules of the game. There is a bug where he sometimes jumps down from a box and sometimes he doesn't and just hovers in the air. Because of time constraints, the bug went unfixed.

6.) Animations

Animation of the boxes move smoothly as they are dropping down. Lazarus has getting squashed animation. Lazarus jumping left and jumping right was sped up to fast to be able to see, therefore it was left commented out.

7.) Audio

The game has main game music as well as sound fx for boxes being crushed, Lazarus being squished, button being pressed, game over, and Lazarus jumping.

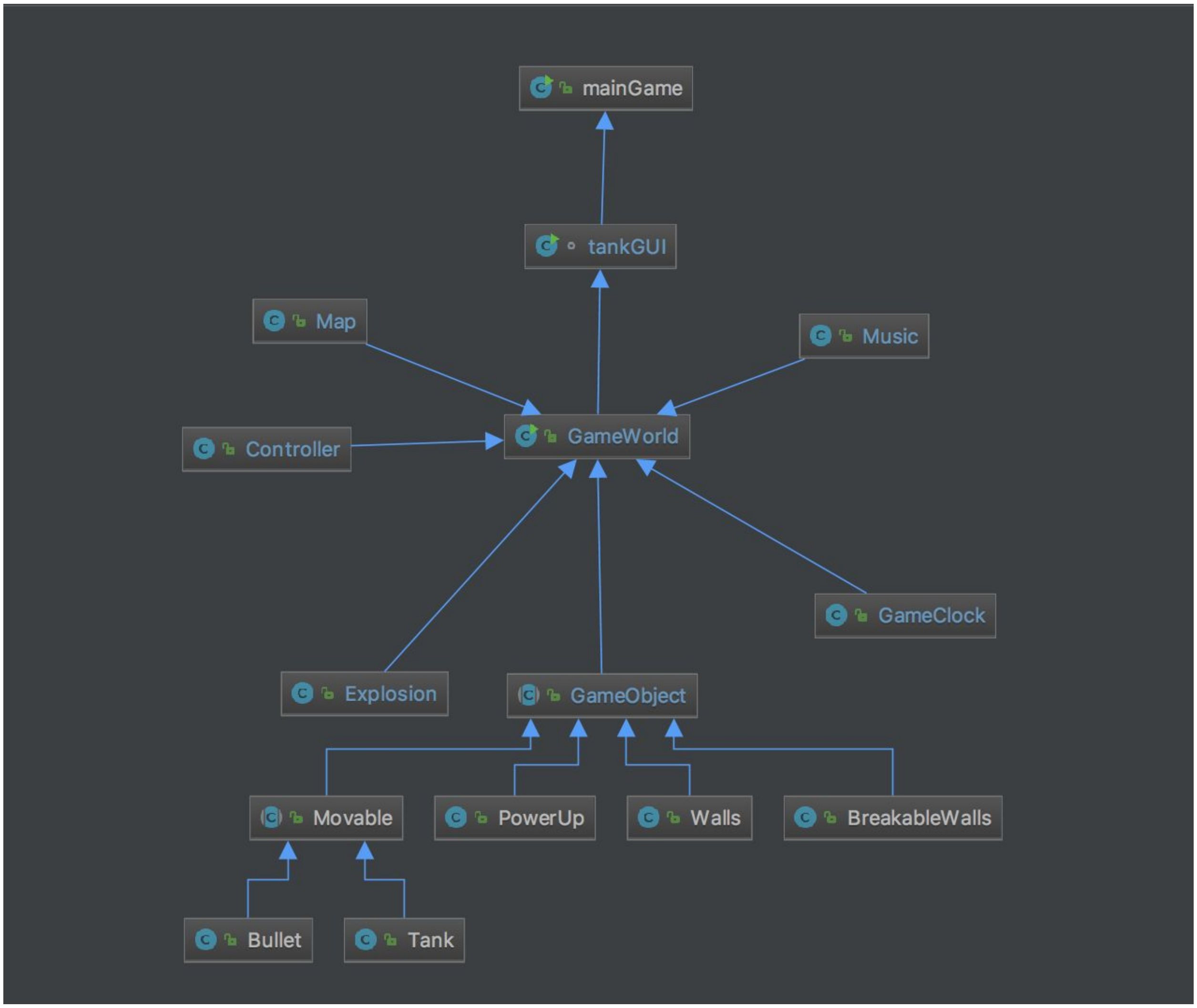
Assumptions:

The assumptions we made when making the tank game were that we were given to freedom over what the games would be so long as they met the criteria given on ilearn. We would not get much in terms of guidance or skeleton code unlike previous assignments, so a lot of mistakes are going to be made and plenty of redesigns implemented. We knew that the design of the class structure was incredibly important in order to easily reuse code for another game. Looking at the sample code for the plane game, we had the impression that the project was going to take several hours to complete and that we should plan our schedules accordingly.

After analyzing the code used in the plane game, we assumed the observer pattern was key when designing the game. We would need classes that are observable and classes that observe these observables. Right off the bat we knew that having all the game objects inherit from one parent class was crucial to the design, since all gameobjects share similar data fields and must have their own reactions to the collisions of one another.

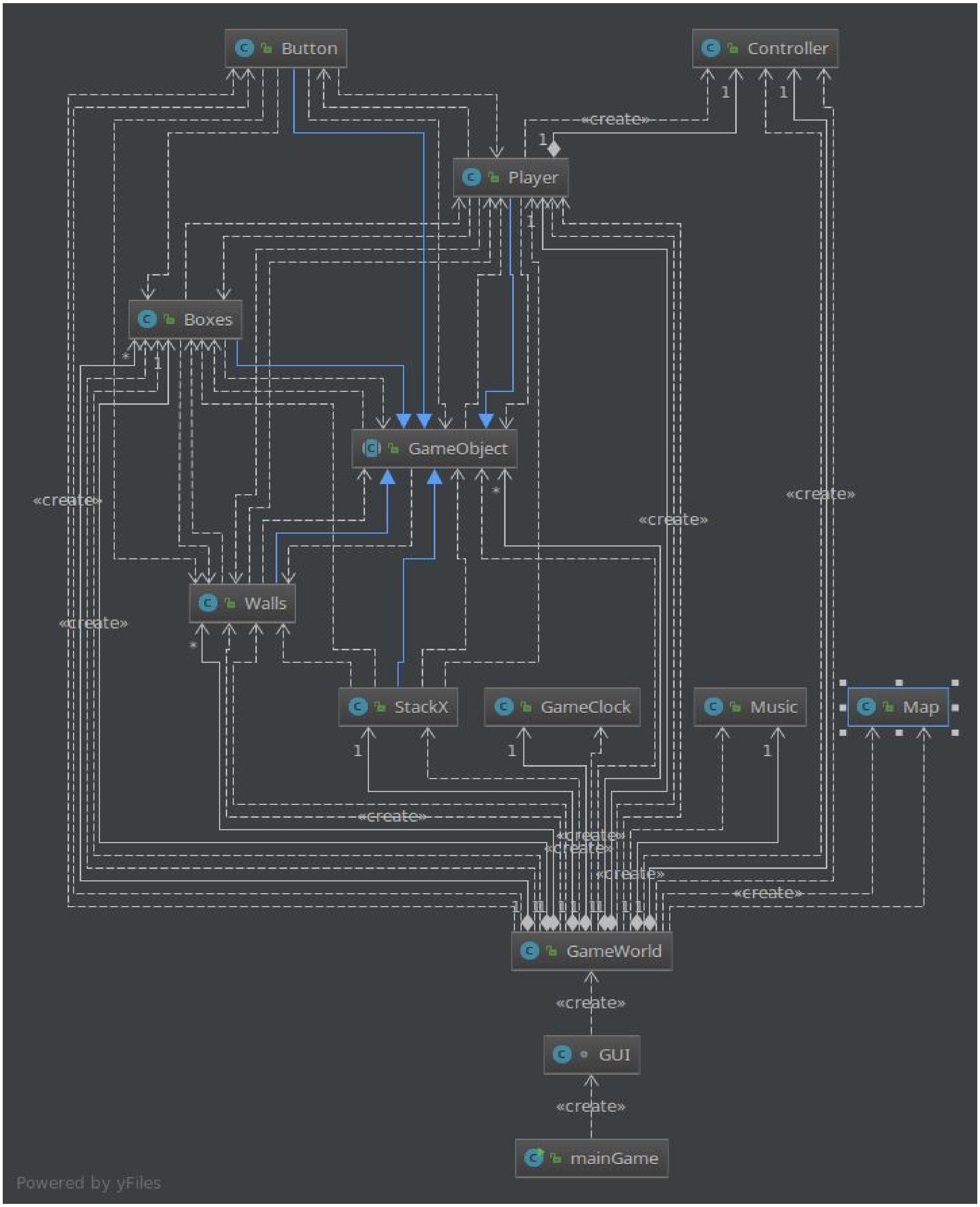
To anchor the game, we settled on a single GameWorld class that would handle all game logic. This means using the Singleton pattern to organize and implement the rest of the classes of the game.

Tank Game Class Diagram:



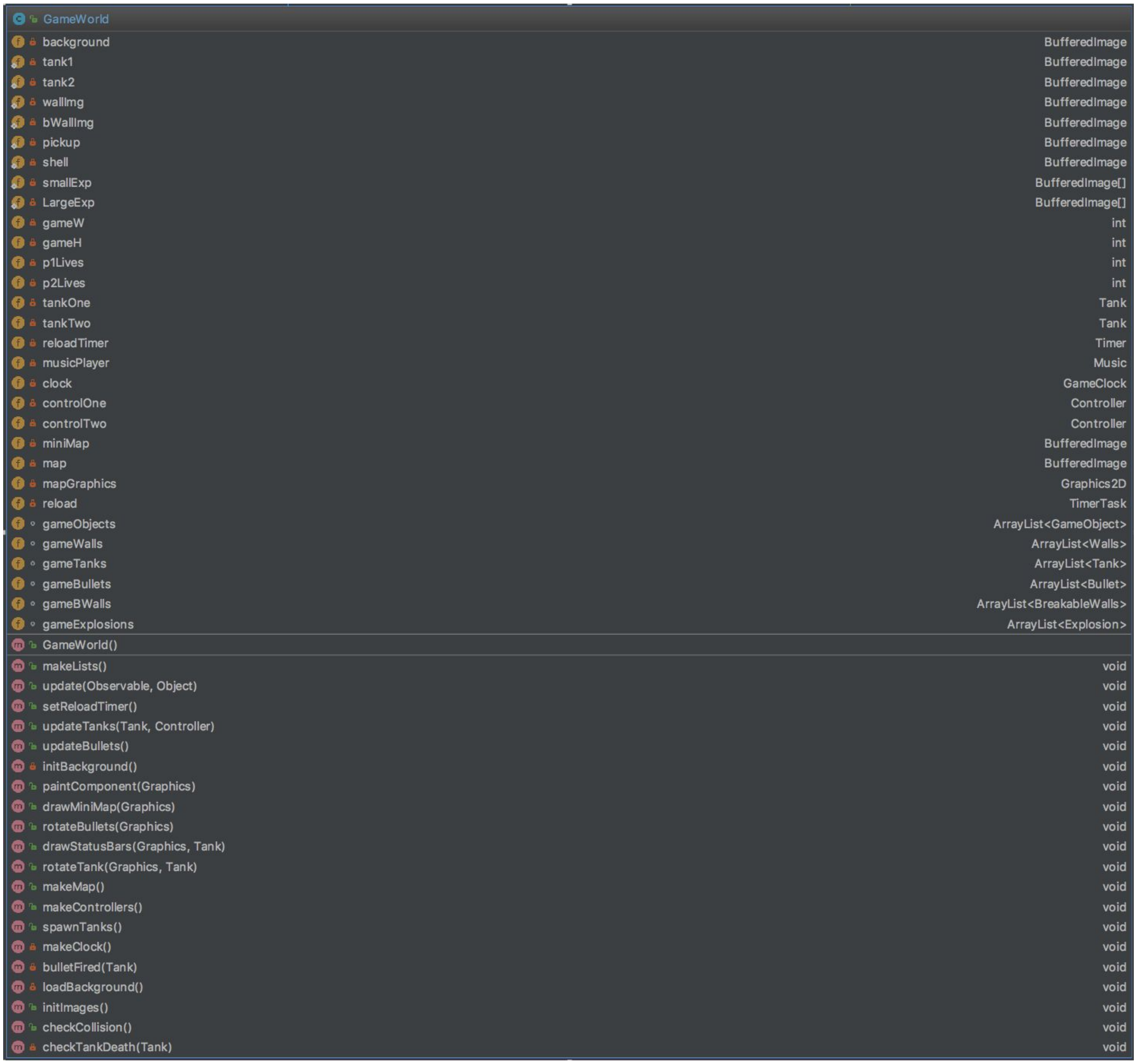
This is the class final class diagram for the tank game. All instances of the game objects are created in the GameWorld class, along with pretty much everything else. This would allow for the GameWorld class to contain all the logic for the game, while all the more broad/specific aspects are taken care of by the others. This also allows for some classes to be more easily interchangeable for better code reusability.

Lazarus Class Diagram:



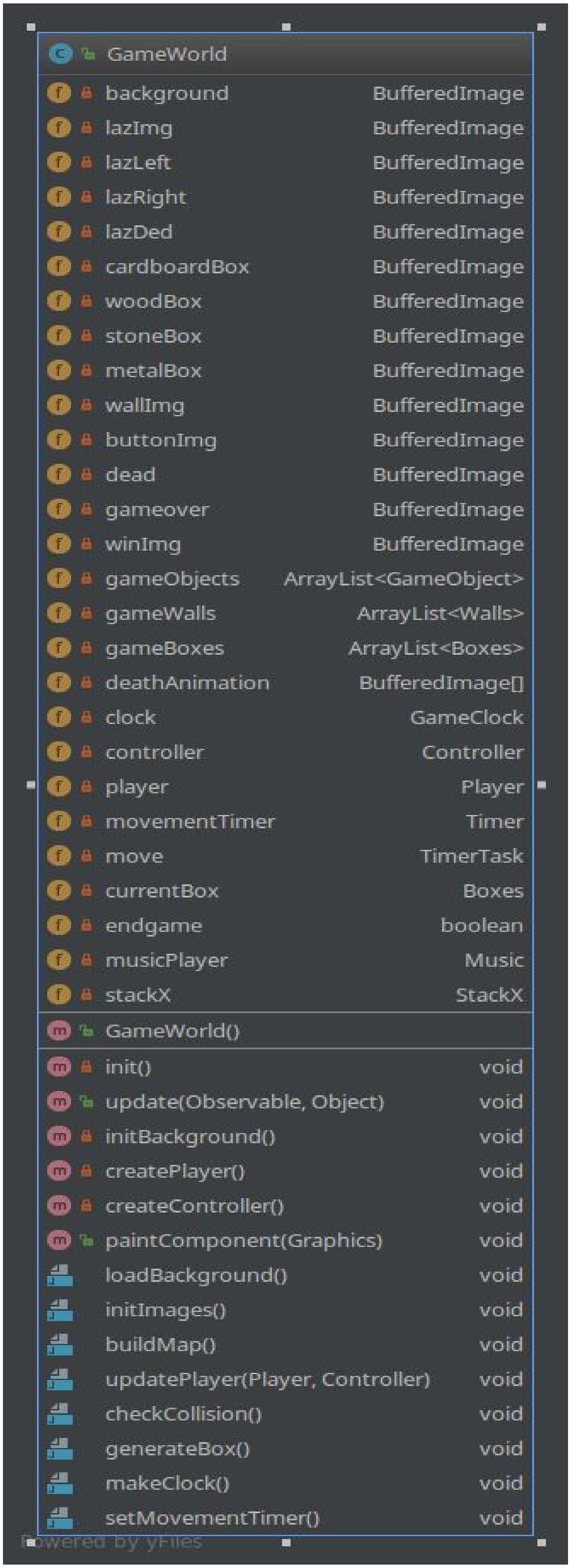
Final class diagram of Lazarus Pit. All classes are classes are dependent GameWorld and all game object instances are created in GameWorld. GameWorld is the logic holder of the game and distributes the work to individual classes that do specific tasks. This allows for code reusability and allows the classes to be interchangeable at any given time.

GameWorld Class (Tank Game):



GameWorld	
background	BufferedImage
tank1	BufferedImage
tank2	BufferedImage
wallImg	BufferedImage
bWallImg	BufferedImage
pickup	BufferedImage
shell	BufferedImage
smallExp	BufferedImage[]
LargeExp	BufferedImage[]
gameW	int
gameH	int
p1Lives	int
p2Lives	int
tankOne	Tank
tankTwo	Tank
reloadTimer	Timer
musicPlayer	Music
clock	GameClock
controlOne	Controller
controlTwo	Controller
miniMap	BufferedImage
map	BufferedImage
mapGraphics	Graphics2D
reload	TimerTask
gameObjects	ArrayList<GameObject>
gameWalls	ArrayList<Walls>
gameTanks	ArrayList<Tank>
gameBullets	ArrayList<Bullet>
gameBWalls	ArrayList<BreakableWalls>
gameExplosions	ArrayList<Explosion>
GameWorld()	
makeLists()	void
update(Observable, Object)	void
setReloadTimer()	void
updateTanks(Tank, Controller)	void
updateBullets()	void
initBackground()	void
paintComponent(Graphics)	void
drawMiniMap(Graphics)	void
rotateBullets(Graphics)	void
drawStatusBars(Graphics, Tank)	void
rotateTank(Graphics, Tank)	void
makeMap()	void
makeControllers()	void
spawnTanks()	void
makeClock()	void
bulletFired(Tank)	void
loadBackground()	void
initImages()	void
checkCollision()	void
checkTankDeath(Tank)	void

GameWorld Class (Lazarus Game):



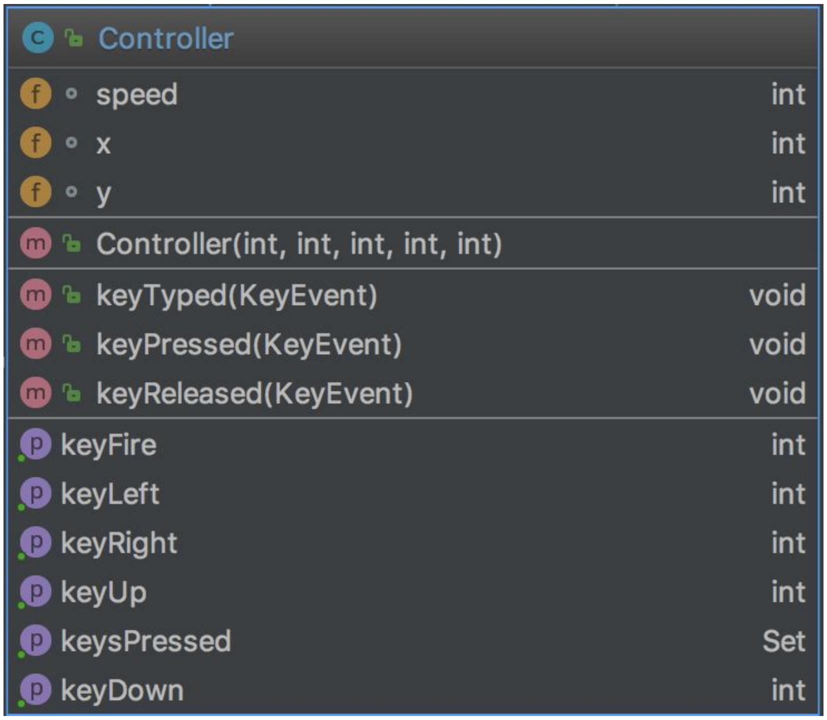
GameWorld	
background	BufferedImage
lazImg	BufferedImage
lazLeft	BufferedImage
lazRight	BufferedImage
lazDed	BufferedImage
cardboardBox	BufferedImage
woodBox	BufferedImage
stoneBox	BufferedImage
metalBox	BufferedImage
wallImg	BufferedImage
buttonImg	BufferedImage
dead	BufferedImage
gameover	BufferedImage
winImg	BufferedImage
gameObjects	ArrayList<GameObject>
gameWalls	ArrayList<Walls>
gameBoxes	ArrayList<Boxes>
deathAnimation	BufferedImage[]
clock	GameClock
controller	Controller
player	Player
movementTimer	Timer
move	TimerTask
currentBox	Boxes
endgame	boolean
musicPlayer	Music
stackX	StackX
GameWorld()	
init()	void
update(Observable, Object)	void
initBackground()	void
createPlayer()	void
createController()	void
paintComponent(Graphics)	void
loadBackground()	void
initImages()	void
buildMap()	void
updatePlayer(Player, Controller)	void
checkCollision()	void
generateBox()	void
makeClock()	void
setMovementTimer()	void

Both GameWorld classes are responsible for all the logic of the game. They populate the world with instances of gameobjects, controllers, and paint the game in accordance of player input and an in game clock. They extend JPanel which allows for painting and they implement the Observer interface that updates only when player input or a clock tick is present. They are singletons due there being only one instance.

The tank games resources are also loaded in through this class so they can be easily assigned to the gameobjects. In order for tanks to move independently of each other, two controllers were made and a hashset was used to record several key presses at a time. Each game object is stored and organized through arraylists. This makes it easy to iterate over things like wall, bullets, tanks to check for collisions and to draw them.

Much like in tank agame, in Lazarus, the resources are loaded through the game world class to be assigned to gameobjects. For the player to move, one controller was created passing in two keys to a hash set to record which key was pressed. To check what objects are being drawn and compared in collision, an arraylist of gameobjects is used to iterate through.

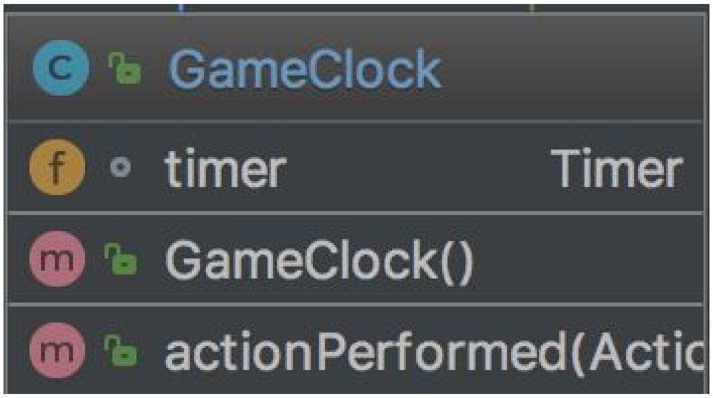
Controller Class:



c	Controller	
f	speed	int
f	x	int
f	y	int
m	Controller(int, int, int, int, int)	
m	keyTyped(KeyEvent)	void
m	keyPressed(KeyEvent)	void
m	keyReleased(KeyEvent)	void
p	keyFire	int
p	keyLeft	int
p	keyRight	int
p	keyUp	int
p	keysPressed	Set
p	keyDown	int

The Controller class handles and stores key presses and communicates that information to the GameWorld. Both Controller classes are essentially same since the class only records key presses and send a hashset over to GameWorld. They implement the KeyListener interface and extend the Observable class. For the tank game, this class requires five key codes that are the controls for each tank. When one of these keys is pressed, it is added to the hashset and the GameWorld is notified, and a similar pattern follows when a key is released.

Game Clock Class:



c GameClock

f timer Timer

m GameClock()

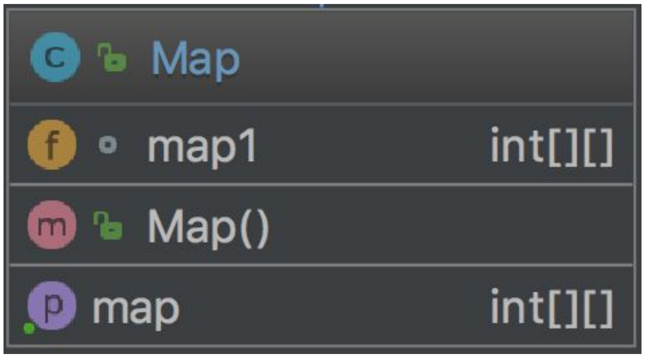
m actionPerformed(Actic

(Tank Game and Lazarus Pit)

The Game Clock class is a timer that is set off every fifteen milliseconds. After testing timer delays, we decided this was the sweet spot for game smoothness and performance, however your experience may vary depending on your hardware. For every fifteen milliseconds the GameWorld is notified to update. This allows for things like bullets and explosions to play out when both players are not performing any movements. This also allows intervals between Lazarus' movements in jump around boxes, keeping him from flying around too fast.

Game Clock is exactly the same for both games since they do not need any modifications to cater to each.

Map Class

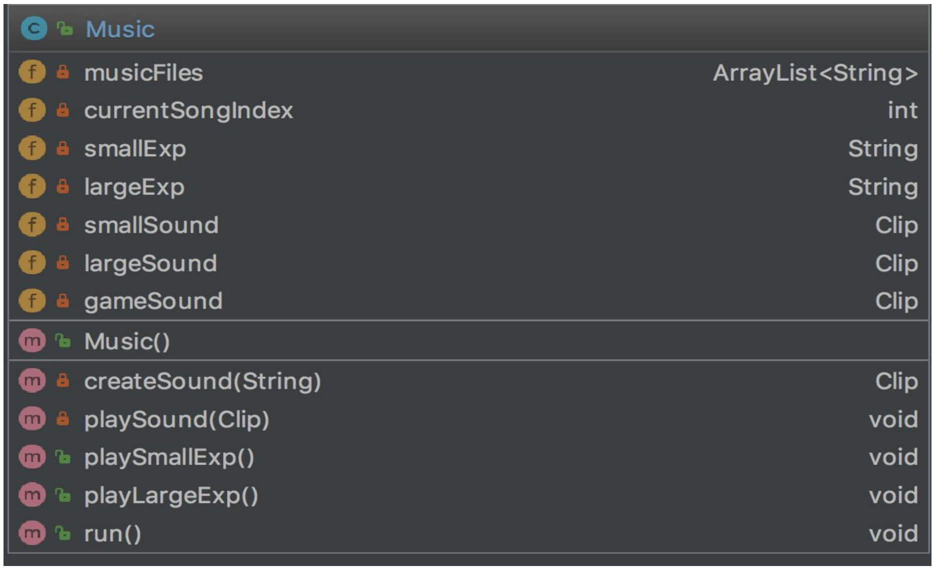


c	Map	
f	map1	int[][]
m	Map()	
p	map	int[][]

(Tank Game and Lazarus Pit)

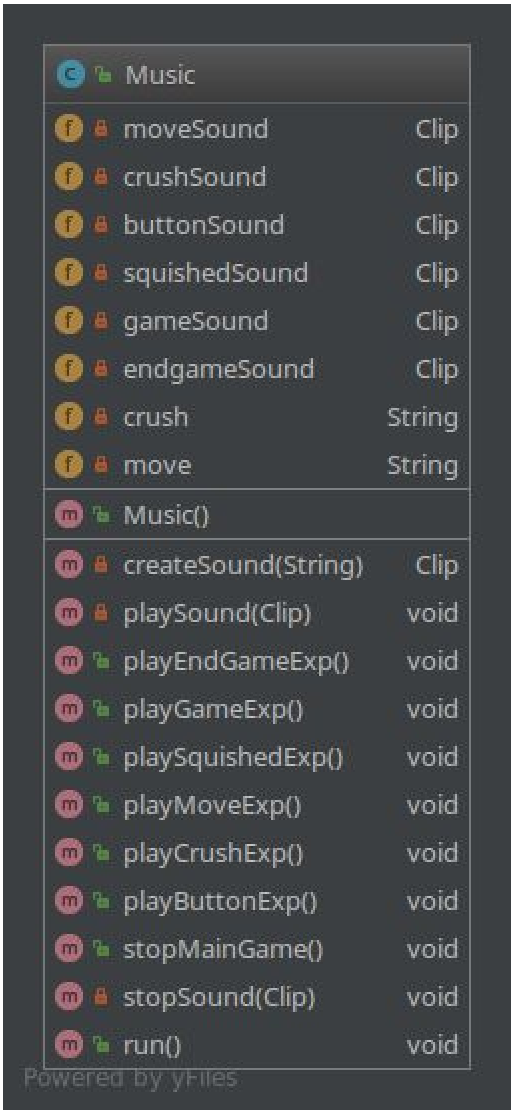
The Map class holds a 2D array that is then passed to the GameWorld. Changing this array changes the design of the map. We found that this is the easiest solution to making the game map since we can visualize the map before runtime.

Music Class:



c	Music	
f	musicFiles	ArrayList<String>
f	currentSongIndex	int
f	smallExp	String
f	largeExp	String
f	smallSound	Clip
f	largeSound	Clip
f	gameSound	Clip
m	Music()	
m	createSound(String)	Clip
m	playSound(Clip)	void
m	playSmallExp()	void
m	playLargeExp()	void
m	run()	void

(Tank Game)



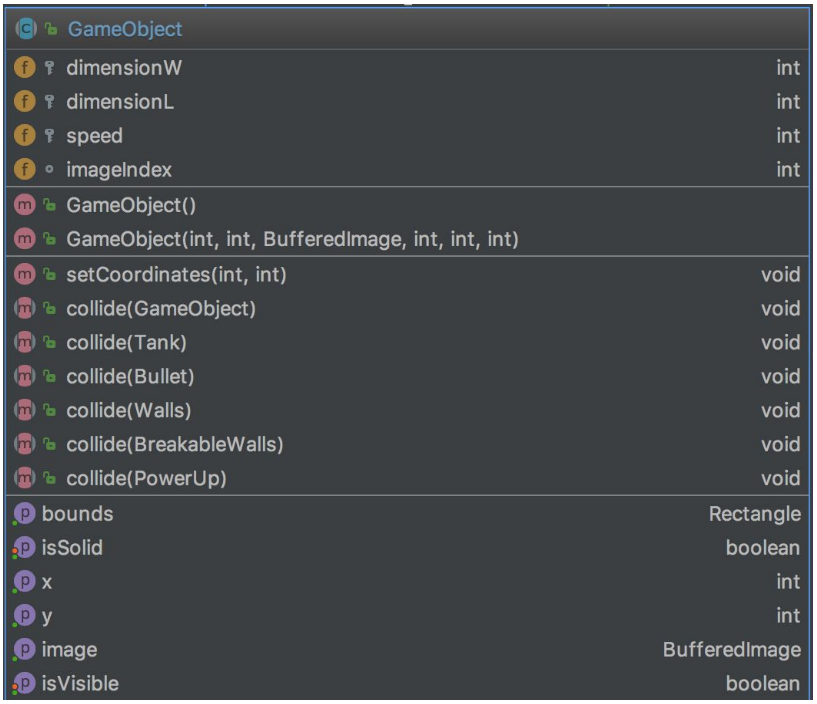
Music

moveSound	Clip
crushSound	Clip
buttonSound	Clip
squishedSound	Clip
gameSound	Clip
endgameSound	Clip
crush	String
move	String
Music()	
createSound(String)	Clip
playSound(Clip)	void
playEndGameExp()	void
playGameExp()	void
playSquishedExp()	void
playMoveExp()	void
playCrushExp()	void
playButtonExp()	void
stopMainGame()	void
stopSound(Clip)	void
run()	void

(Lazarus Pit)

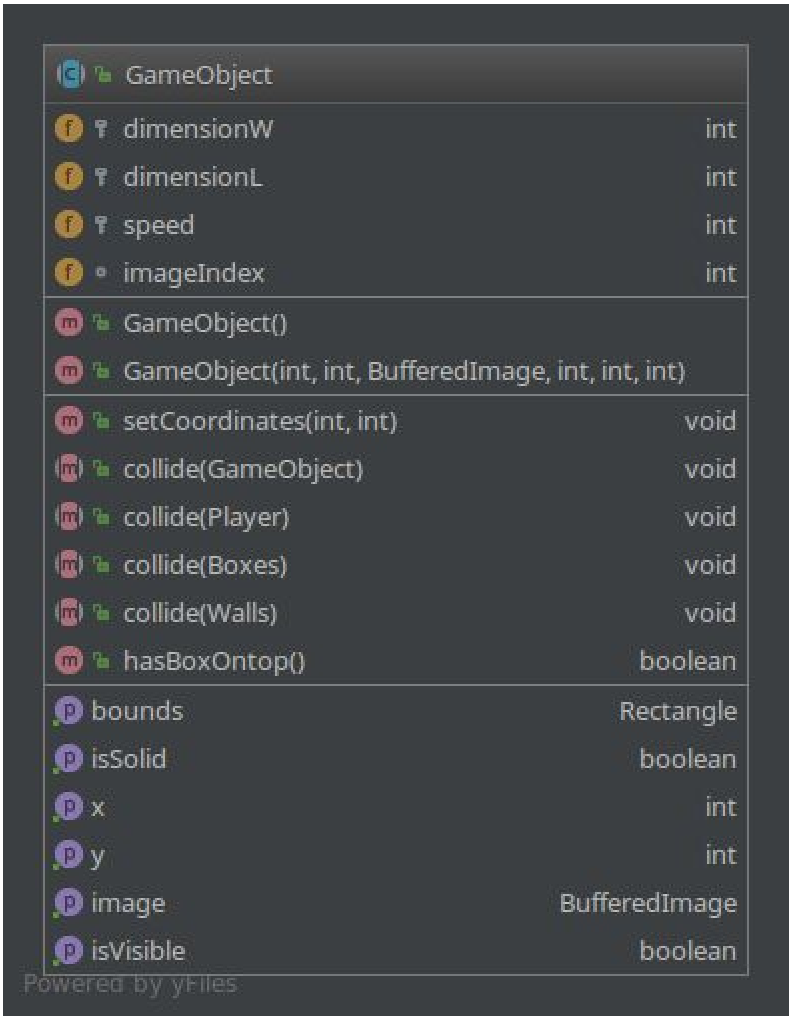
The Music Class loads the .wav and .mp3 files and plays the sounds through it specific methods. The GameWorld will call these methods when explosions happen or when either Lazarus moves, gets squished, or a box is crushed. These sounds are in the resources folder.

GameObject Class:



c	GameObject	
f	dimensionW	int
f	dimensionL	int
f	speed	int
f	imageIndex	int
m	GameObject()	
m	GameObject(int, int, BufferedImage, int, int, int)	
m	setCoordinates(int, int)	void
m	collide(GameObject)	void
m	collide(Tank)	void
m	collide(Bullet)	void
m	collide(Walls)	void
m	collide(BreakableWalls)	void
m	collide(PowerUp)	void
p	bounds	Rectangle
p	isSolid	boolean
p	x	int
p	y	int
p	image	BufferedImage
p	isVisible	boolean

(Tank Game)

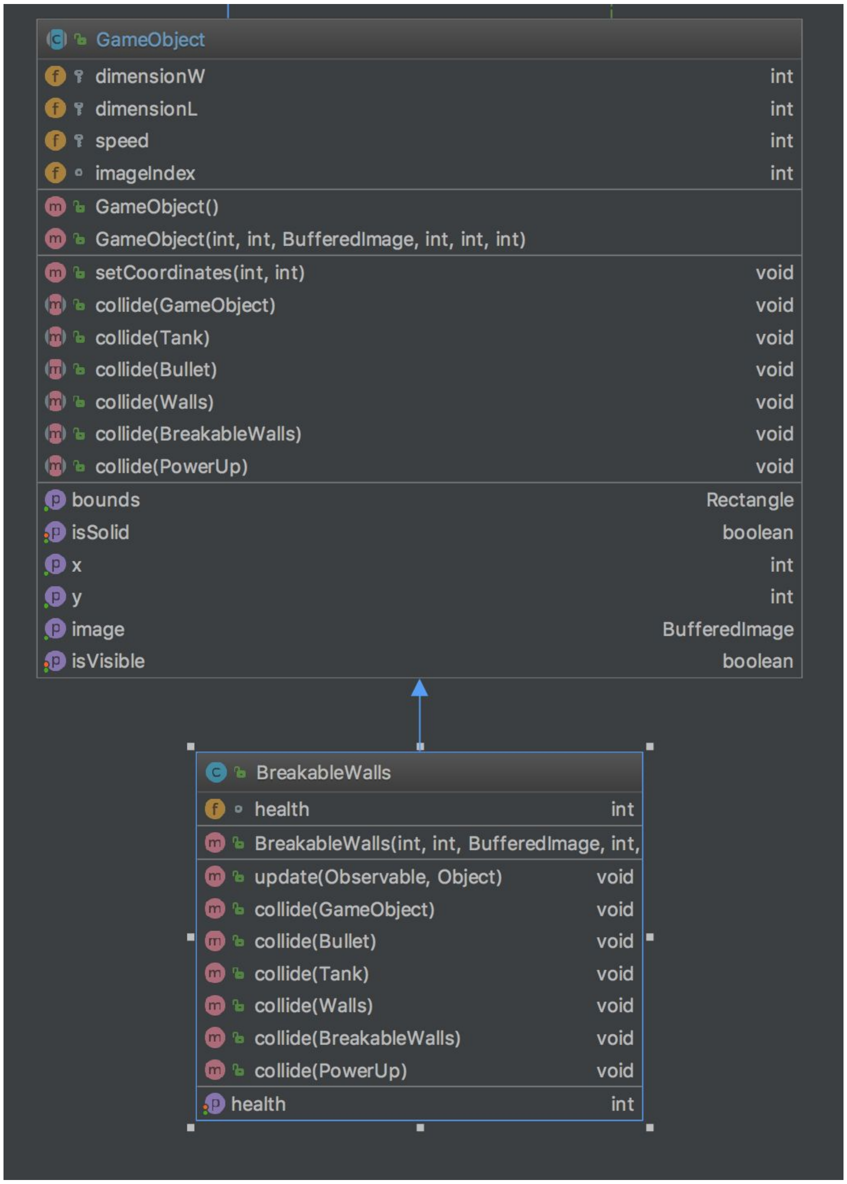


GameObject	
dimensionW	int
dimensionL	int
speed	int
imageIndex	int
GameObject()	
GameObject(int, int, BufferedImage, int, int, int)	
setCoordinates(int, int)	void
collide(GameObject)	void
collide(Player)	void
collide(Boxes)	void
collide(Walls)	void
hasBoxOntop()	boolean
bounds	Rectangle
isSolid	boolean
x	int
y	int
image	BufferedImage
isVisible	boolean

(Lazarus Pit)

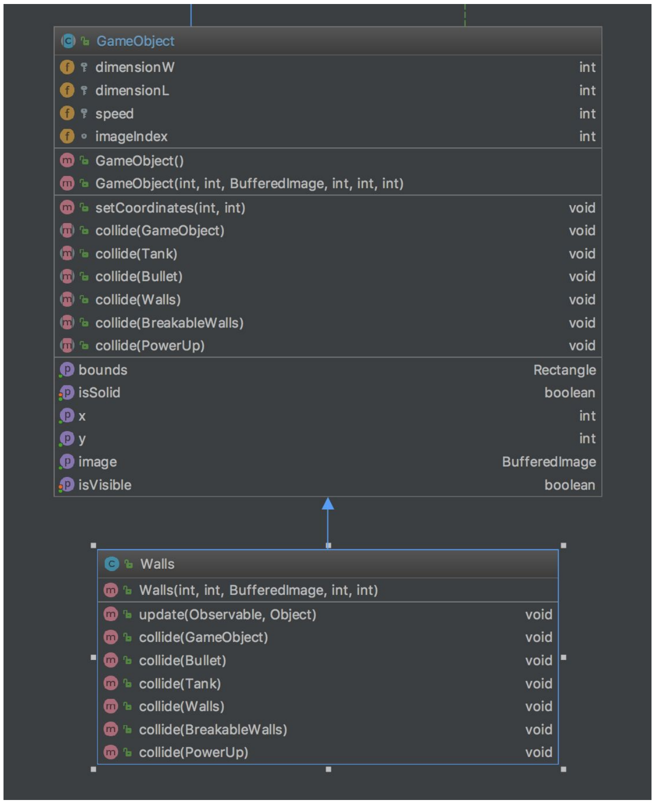
The GameObject class is the parent class of all of the objects in the game. Each method and data field is common among all game objects. This allows for easy collisions and creations of game objects. GameObjects are both Observers and Observables since the tanks and bullets as well as Player and Boxes are observed by the game clock while also be observables to the GameWorld. Since this the parent class of all game objects, this class is abstract and is never instantiated.

Breakable Walls Class



The Breakable Walls Class is a subclass of GameObject. These are the walls that players can destroy for a competitive advantage. Each wall has health and will destroy itself when it's health reaches zero. Once the wall is destroyed it stops getting drawn and becomes free space for the tanks to move through.

Wall Class:



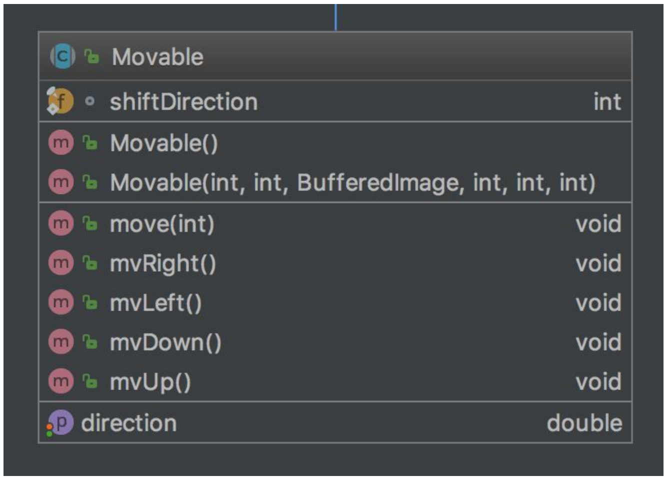
GameObject	
dimensionW	int
dimensionL	int
speed	int
imageIndex	int
GameObject()	
GameObject(int, int, BufferedImage, int, int, int)	
setCoordinates(int, int)	void
collide(GameObject)	void
collide(Tank)	void
collide(Bullet)	void
collide(Walls)	void
collide(BreakableWalls)	void
collide(PowerUp)	void
bounds	Rectangle
isSolid	boolean
x	int
y	int
image	BufferedImage
isVisible	boolean

Walls	
Walls(int, int, BufferedImage, int, int)	
update(Observable, Object)	void
collide(GameObject)	void
collide(Bullet)	void
collide(Tank)	void
collide(Walls)	void
collide(BreakableWalls)	void
collide(PowerUp)	void

(Tank Game and Lazarus)

The Wall class provides a barrier to set the confines of where the Tanks can move. It is a subclass of the GameObject class. We spawn these walls by reading the 2D array of Map Class and instiating the Wall objects in accordance to the layout of the 2D array. We found this to be the easiest solution to designing and implementing map ideas.

Movable Class:

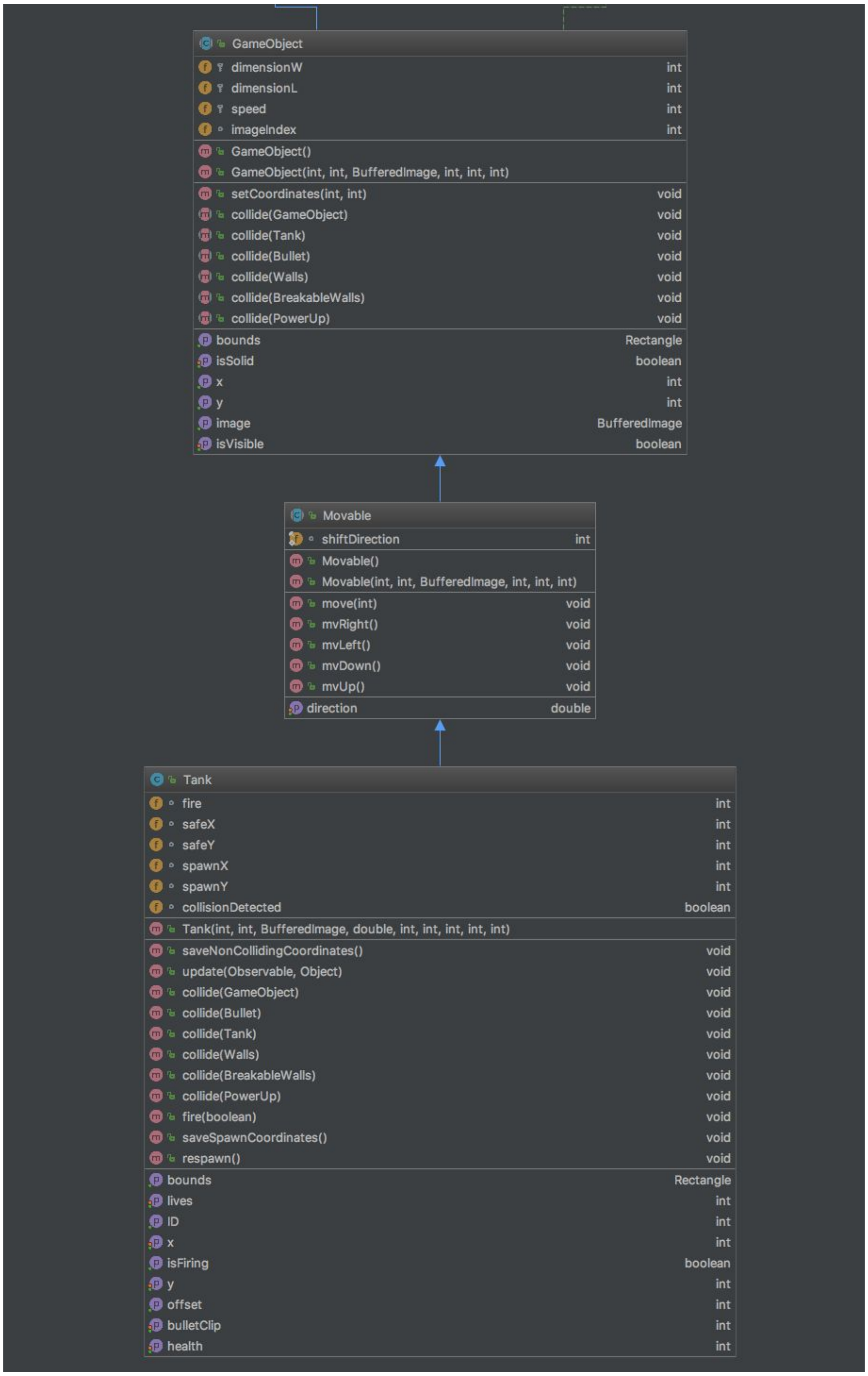


Movable	
shiftDirection	int
Movable()	
Movable(int, int, BufferedImage, int, int, int)	
move(int)	void
mvRight()	void
mvLeft()	void
mvDown()	void
mvUp()	void
direction	double

(Tank Game)

The Movable Class is a child of the GameObject Class, but is a parent of the Tank and Bullet Class. This is because the methods that account for movement and direction can shared between the tanks and the bullets. Moving diagonal for the tanks is done by using the sine and cosine functions, and the same is done for the bullets.

Tank Class:



GameObject	
dimensionW	int
dimensionL	int
speed	int
imageIndex	int
GameObject()	
GameObject(int, int, BufferedImage, int, int, int)	
setCoordinates(int, int)	void
collide(GameObject)	void
collide(Tank)	void
collide(Bullet)	void
collide(Walls)	void
collide(BreakableWalls)	void
collide(PowerUp)	void
bounds	Rectangle
isSolid	boolean
x	int
y	int
image	BufferedImage
isVisible	boolean

Movable	
shiftDirection	int
Movable()	
Movable(int, int, BufferedImage, int, int, int)	
move(int)	void
mvRight()	void
mvLeft()	void
mvDown()	void
mvUp()	void
direction	double

Tank	
fire	int
safeX	int
safeY	int
spawnX	int
spawnY	int
collisionDetected	boolean
Tank(int, int, BufferedImage, double, int, int, int, int, int)	
saveNonCollidingCoordinates()	void
update(Observable, Object)	void
collide(GameObject)	void
collide(Bullet)	void
collide(Tank)	void
collide(Walls)	void
collide(BreakableWalls)	void
collide(PowerUp)	void
fire(boolean)	void
saveSpawnCoordinates()	void
respawn()	void
bounds	Rectangle
lives	int
ID	int
x	int
isFiring	boolean
y	int
offset	int
bulletClip	int
health	int

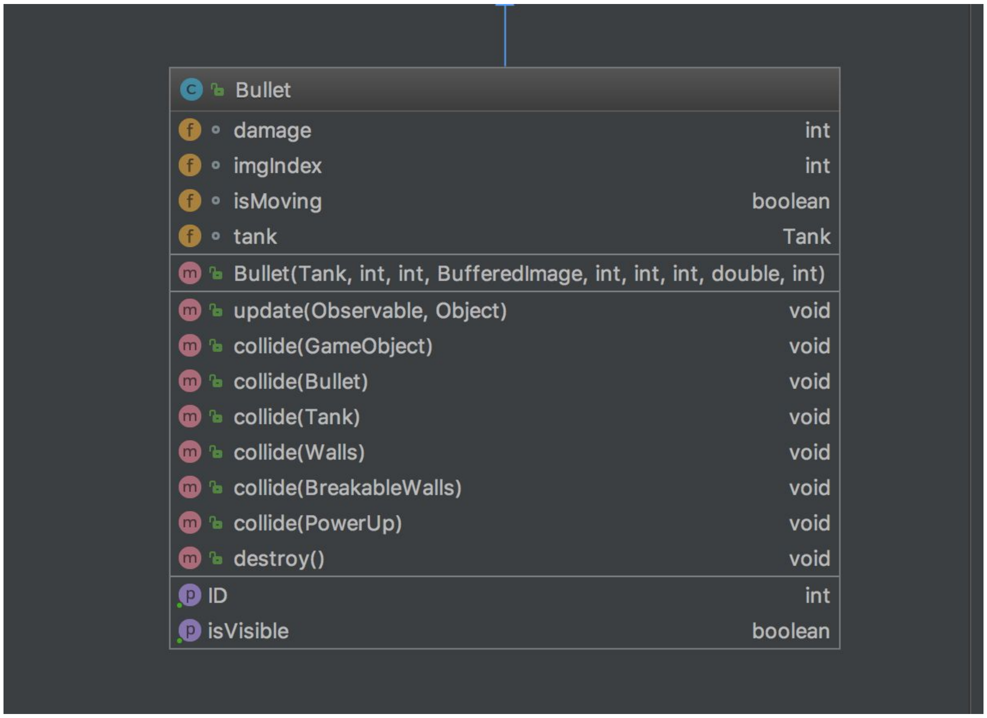
The Tank Class holds all the methods and data fields for the game tanks. It extends Movable which extends GameObject. The tanks can rotate 360 degrees and shoot bullets in the direction they are facing.

This is where the collision methods from the GameObject Class are filled in to implement the collision logic for the game. The Collision for the tanks against walls was implemented by saving safe, non colliding coordinates until a collision is detected. When a collision is detected, the tanks are taken back to the last safe non colliding coordinate saved in the game. I found this to be a more effective way to handle collisions rather than having the tanks bounce off the wall.

Collision of bullets create and explosion and damage tanks if the ID of the bullet does not match the ID of the tank it came from. Bullets damage is 5 and Tank health is 100.

Tanks shoot by activating a boolean within them that then spawns a bullet in the GameWorld. Since creating a bullet at the tanks coordinates would spawn the bullet at the upper right of the tank, we pass in an offset to create the illusion that the bullet spawns in the middle of the tank.

Bullet Class:

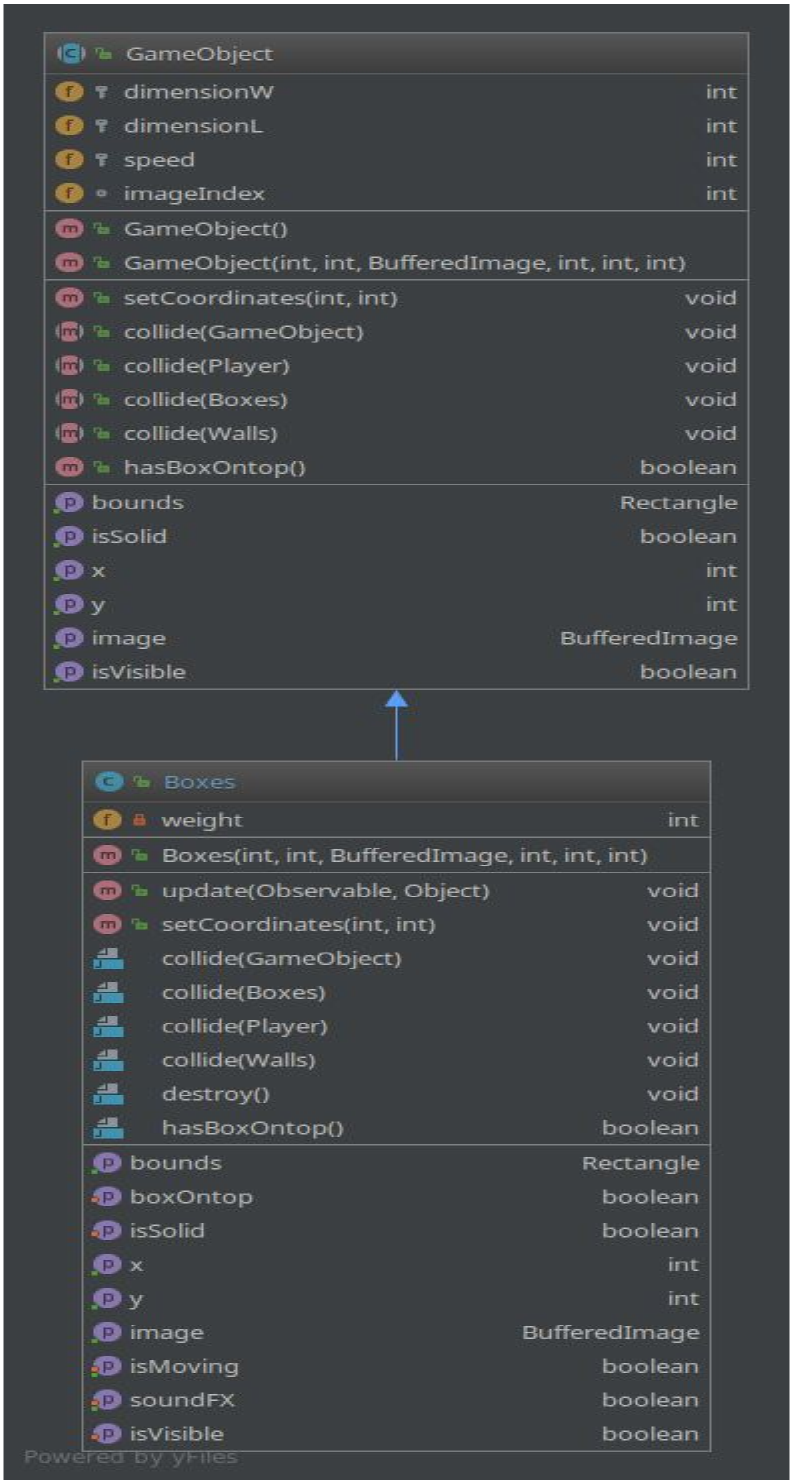


c	Bullet	
f	damage	int
f	imgIndex	int
f	isMoving	boolean
f	tank	Tank
m	Bullet(Tank, int, int, BufferedImage, int, int, int, double, int)	
m	update(Observable, Object)	void
m	collide(GameObject)	void
m	collide(Bullet)	void
m	collide(Tank)	void
m	collide(Walls)	void
m	collide(BreakableWalls)	void
m	collide(PowerUp)	void
m	destroy()	void
p	ID	int
p	isVisible	boolean

The Bullet class is an extension of the Movable Class which extends GameObject. These are the bullets that damage other tanks and breakable walls. When a bullet collides with any gameobject, an explosion is created at those coordinates and the bullet stops being drawn. An explosion sound is also created when a bullet collides with a tank or blows up a wall. This creates the illusion of bullets exploding on impact.

Lazarus Specific Classes:

Boxes:

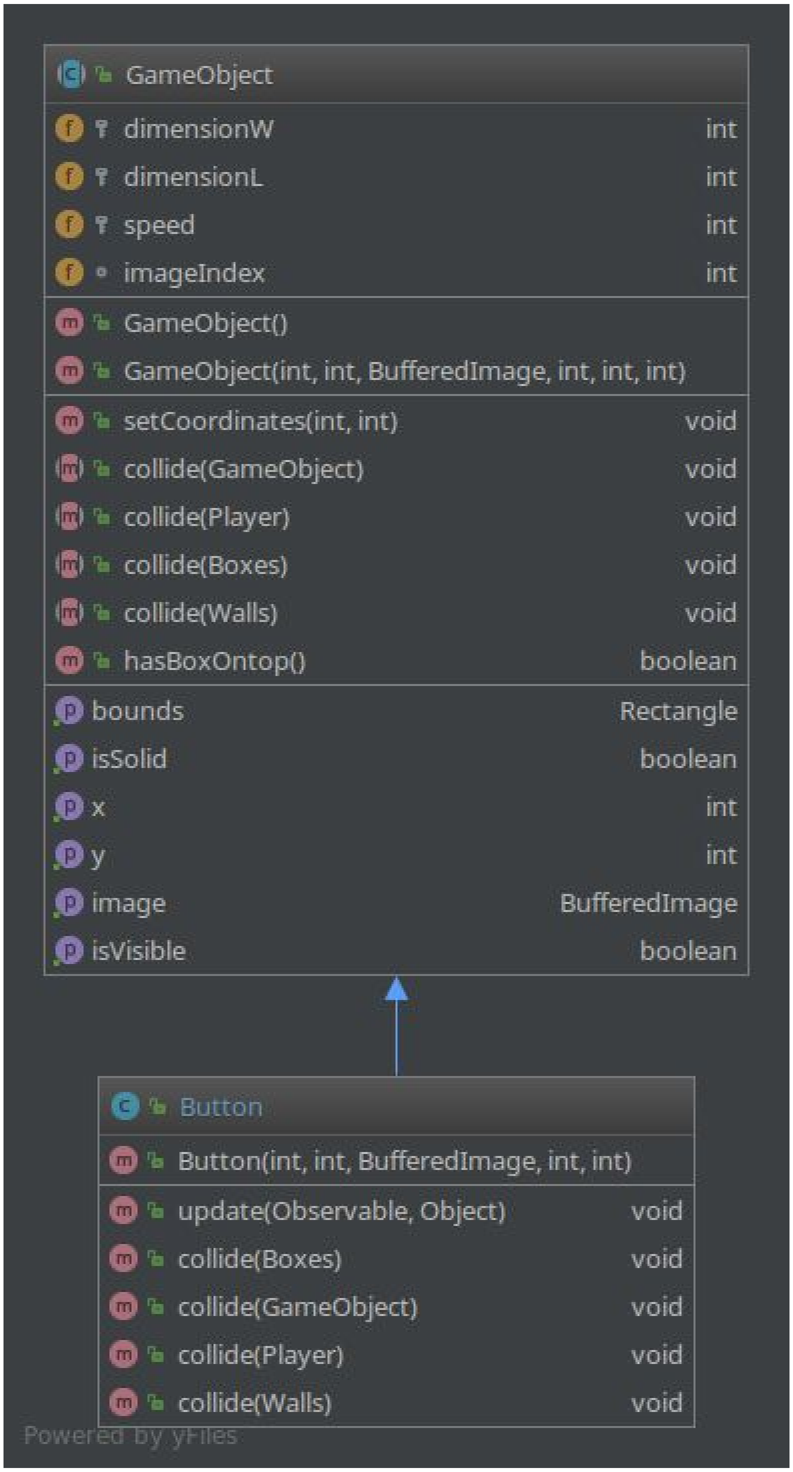


GameObject	
dimensionW	int
dimensionL	int
speed	int
imageIndex	int
GameObject()	
GameObject(int, int, BufferedImage, int, int, int)	
setCoordinates(int, int)	void
collide(GameObject)	void
collide(Player)	void
collide(Boxes)	void
collide(Walls)	void
hasBoxOntop()	boolean
bounds	Rectangle
isSolid	boolean
x	int
y	int
image	BufferedImage
isVisible	boolean

Boxes	
weight	int
Boxes(int, int, BufferedImage, int, int, int)	
update(Observable, Object)	void
setCoordinates(int, int)	void
collide(GameObject)	void
collide(Boxes)	void
collide(Player)	void
collide(Walls)	void
destroy()	void
hasBoxOntop()	boolean
bounds	Rectangle
boxOntop	boolean
isSolid	boolean
x	int
y	int
image	BufferedImage
isMoving	boolean
soundFX	boolean
isVisible	boolean

The Box class is an extension of GameObject class to allow for interaction with other objects while also having the ability to move around in the map. Each new instance of this class is suppose to represent a random box varying by weight. This class can either be a cardboard, wood, metal, or stone box which crushes or gets crushed depending on the difference of weight of the dropping box and if they stack on top.

Button:

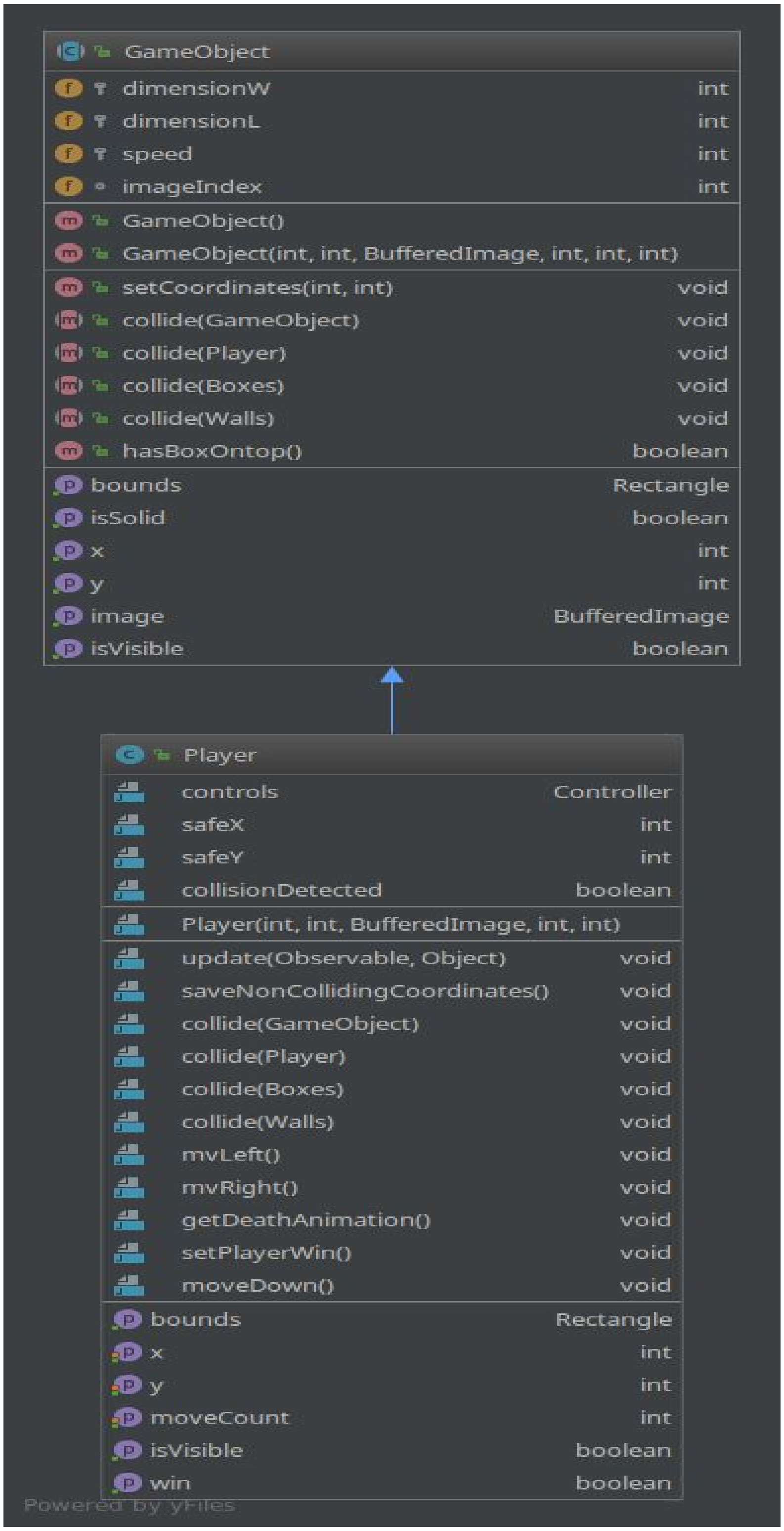


GameObject	
dimensionW	int
dimensionL	int
speed	int
imageIndex	int
GameObject()	
GameObject(int, int, BufferedImage, int, int, int)	
setCoordinates(int, int)	void
collide(GameObject)	void
collide(Player)	void
collide(Boxes)	void
collide(Walls)	void
hasBoxOntop()	boolean
bounds	Rectangle
isSolid	boolean
x	int
y	int
image	BufferedImage
isVisible	boolean

Button	
Button(int, int, BufferedImage, int, int)	
update(Observable, Object)	void
collide(Boxes)	void
collide(GameObject)	void
collide(Player)	void
collide(Walls)	void

The Button class is an extension of GameObjects class and like Boxes this allow the interaction with the player while letting the player move about in the game. When it collides with the player this triggers the victory window to come up to congratulate the player.

<u>Player:</u>

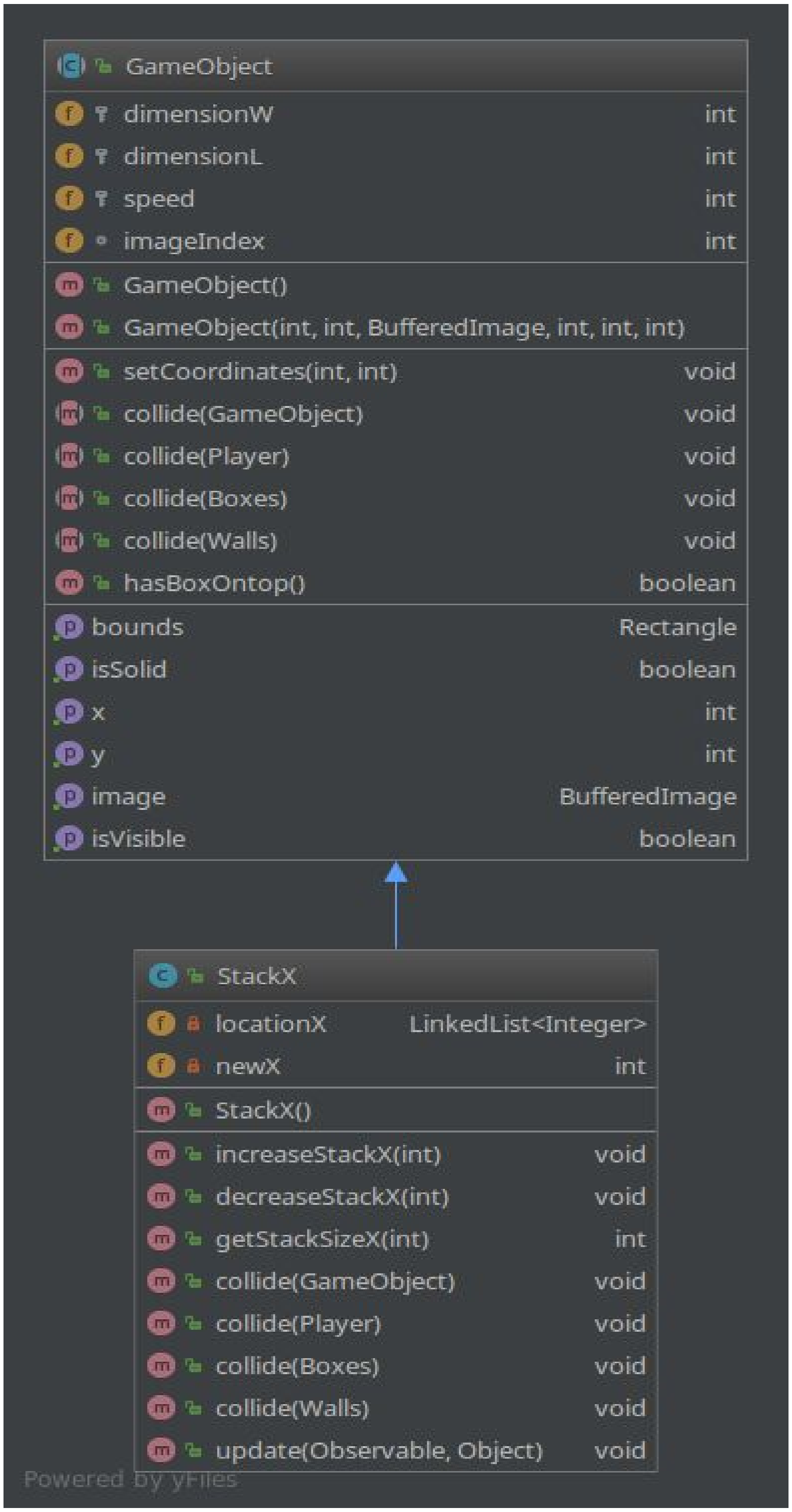


GameObject	
dimensionW	int
dimensionL	int
speed	int
imageIndex	int
GameObject()	
GameObject(int, int, BufferedImage, int, int, int)	
setCoordinates(int, int)	void
collide(GameObject)	void
collide(Player)	void
collide(Boxes)	void
collide(Walls)	void
hasBoxOntop()	boolean
bounds	Rectangle
isSolid	boolean
x	int
y	int
image	BufferedImage
isVisible	boolean

Player	
controls	Controller
safeX	int
safeY	int
collisionDetected	boolean
Player(int, int, BufferedImage, int, int)	
update(Observable, Object)	void
saveNonCollidingCoordinates()	void
collide(GameObject)	void
collide(Player)	void
collide(Boxes)	void
collide(Walls)	void
mvLeft()	void
mvRight()	void
getDeathAnimation()	void
setPlayerWin()	void
moveDown()	void
bounds	Rectangle
x	int
y	int
moveCount	int
isVisible	boolean
win	boolean

The Player Object is suppose to represent Lazarus in the game. Player extends GameObject and allows Lazarus to move in the left and right directions depending on the height of the boxes it is relative to in the game. Lazarus has an interaction with dropping boxes as it will get squashed if a box lands on him which also ends the game prompting the game over screen. Lazarus also interacts with the button which prompts the victory screen giving the player of satisfaction of beating a really easy game.

StackX:



GameObject	
dimensionW	int
dimensionL	int
speed	int
imageIndex	int
GameObject()	
GameObject(int, int, BufferedImage, int, int, int)	
setCoordinates(int, int)	void
collide(GameObject)	void
collide(Player)	void
collide(Boxes)	void
collide(Walls)	void
hasBoxOntop()	boolean
bounds	Rectangle
isSolid	boolean
x	int
y	int
image	BufferedImage
isVisible	boolean

StackX	
locationX	LinkedList<Integer>
newX	int
StackX()	
increaseStackX(int)	void
decreaseStackX(int)	void
getStackSizeX(int)	int
collide(GameObject)	void
collide(Player)	void
collide(Boxes)	void
collide(Walls)	void
update(Observable, Object)	void

The StackX class extends GameObjects and stores the amount of boxes stacked in a certain x-location. This is used with Lazarus movement which allows Lazarus to know if there are boxes or no boxes in order to determine if Lazarus is able to go up a box or go down a box.

Results, Reflections, and Conclusions:

The results of the two games are two functional games with their key features implemented. For the tank game , performance may vary depending on your hardware due to optimization issues. We tested the game of both partner's machines and performance was much better on one compared to the other. It is important to turn off any anti ghosting or any sticky key features that may cause issue to multiple keyboard presses, especially for the space and enter keys.

The power ups for the tank game could not be completed in time, but the rest of the features are there with tank lives, sounds, explosions, and a gameover screen. One bug that I found but was unable to fix was that the tanks would not go back to spawn if one died while colliding with the other tank. This is probably due to the tank collide method overriding the respawn method. We learned alot about using GUIs in java and how animations, key controls, and sound can be produced a fun game that you can share with others.

This was definitely the most difficult project we had so far in our cs career. Many mistakes were made, and plenty of hours were spent fixing/resolving them. We had issues merging and resolving conflicts, as well as working together to build a complex program at the same time. Several hours were spent getting tank movement and rotations as smooth as they are, as there were often times when the tank would not rotate the image correctly or when the tank would move in a direction it was not facing. Collisions were tough only because checking collisions for several objects would severely slow down the game if not implemented correctly. In Lazarus, majority of the time was spent on collisions between boxes and as well as allowing Lazarus to be able to jump up and down boxes. We were able to take the collision methods created in tank game and insert them into Lazarus but some changes had to be made. Since the collision between player was made already, we needed collision between the boxes. In order to do so we had to make changes to the dimensions of the boxes and player from 40x40 to 39x39 to avoid diagonal collisions. As well as we had to make an adjustment of adding the boxes to gameObjects array list until after it has stop moving to avoid the box from colliding on itself when spawned. There was a bug that was unresolved with Lazarus where he would hover in the air instead of jumping down a level on some occasions.

This term project proved to be quite difficult mostly due to it being the first time we went from ideas to codem, rather than being given a skeleton code that needed to fulfill very specific requirements. It made us deal with the consequences of decisions that we made, whether it may be design decisions or time/space complexity decisions.