SOFTWARE PROJECT - GAME DESIGN

I. Title Page

Assignment Title: 2D Game Design - Tank War and Rainbow Reef

Name: Jianfei Zhao

Link to the repository

Tank War: https://github.com/jzhao11/tank_war_game

Rainbow Reef: https://github.com/jzhao11/rainbow_reef_game

II. Introduction

a. Project Overview

As the term project, the development of these two games provides us another opportunity to familiarize the object-oriented principles and furthers our understanding in the field of software engineering. The first game is tank war, aiming to help us get basic knowledge about game design with limited time. As a matter of fact, code reusability is treated as the main purpose as well as the key point of this term project. Therefore, completion of the first game is of vital importance, given the experience and lessons we may learn from its design and implementation. If the tank war is with good OOP, both class hierarchy and a good portion of code could be reused for the programming in the second game. In other words, a more reusable tank war game will just make the life of second game development relatively easier.

b. Introduction of the Tank Game

As the first game project, the tank war is actually a perfect practice for game design. It involves a lot of development skills, such as arrangement of files, movement control with key listeners, design of user interface (game board), and so on.

From the technical perspective, much endeavor is made on the design of classes in this game. The control over tank movement together with the collision handling between gaming elements is taken as the trickiest part during the development. Besides this, the tank war game resolves problems like map loading and bullets shooting. Considering the complex functionalities it may have, the class design should distribute responsibilities into several modules to lower the coupling between classes. More importantly, a good design pattern can be rescaled in the second game to save time. Detailed discussion is available in section **VII ~ XI**.

When working on the development mentioned above, different kinds of data structures, such as HashMap, LinkedList, and ArrayList are applied into programming according to needs. As the game design progresses, attention is also paid to encapsulation to ensure that outside classes can only use some public method interfaces, without access to those private member fields. The whole project is debugged and tested back and forth, with the gaming process compared with requirements.

c. Introduction of the Second Game

My choice for the second game is rainbow reef. With the implementation procedure of tank war, the second game may become easier, since the code reusability will simply the class design and thus reduce a lot of workload.

Technically, if taking the tank war as a template, the development for rainbow reef could just be a process of rescaling.

Honestly speaking, most of the problems involved in rainbow reef have already been handled in the first game. For example, the control of seasnail (Katch) only contains horizontal movements (moving towards either left or right). This may seem to be a simplified version of the previous tank control, meaning that it will require less effort. And the starfish (Pop) could be regarded as a bullet of the seasnail, which is analogous to the logic within tank war. As a result, a large portion of code in the first game can be reused in the development of rainbow reef.

Since the rainbow reef is still featured with elements that are different from tank war, most energy will then be focused on its own logic. Moreover, on the ground of the experience learned from tank war game, some modification will be made on class design and implementation to better the efficiency. Detailed discussion for class design is available in section **VII ~ XI**.

III. Development Environment

a. Version of Java Used

The version of Java used for development is 1.8.0_101.

b. Integrated Development Environment (IDE) Used

The IDE for Java is NetBeans 8.2 on Windows System.

c. Special Libraries Used or Special Resources

Both tank war and rainbow reef game only utilize libraries that are available in JDK 1.8 without any other special ones. The sound and image resources come from the given folder (downloaded from iLearn).

IV. How to Build or Import the Project in the IDE

i). Clone the repository from the link in title page (page 1), by using the command listed below.

For tank war game:

git clone https://github.com/jzhao11/tank_war_game

For rainbow reef game:

git clone https://github.com/jzhao11/rainbow_reef_game

ii). Open the NetBeans IDE. Import the Project by clicking “File->New Project” at the top-left of the navigation bar.

iii). Choose “Java Project with Existing Sources” in the “Projects” choice box and click “Next”. Specify a name and location for the new project and click “Next”. Suggestions for project names are listed below.

For tank war game:

csc413_tank_jzhao11

For rainbow reef game:

csc413_reef_jzhao11

iv). Click “Add Folder” next to the “Source Package Folder” choice box. In the pop-up window, find and choose the local repo. Click “Open” and the file path will appear in the upper box. Then click “Next”. Default names for the repository folders cloned from GitHub are listed below.

For tank war game:

csc413-tankgame-jzhao11

For rainbow reef game:

csc413-secondgame-jzhao11

v). *.java files and related resources (image and audio files) will appear in the "Included Files" box. All these files are needed. Click "Finish" to finish importing the project.

vi). To build the project, right click the imported project in the top-left "Project" box, and choose "Clean and Build". This can be done by clicking the "Clean and Build Project" icon (Shift+F11 as default shortcut) on the top navigation bar.

V. How to Run the Project

a. Project Execution

i). To run the project for playing, choose the imported project in the top-left "Project" box. Right click this project and then choose "Run". This can be done by clicking the "Run Project" icon (Ctrl+F11 as default shortcut) on the top navigation bar.

ii). For users’ convenience, a jar file has been provided for each game, in “jar” folder of the repo. Double click the jar file and users may directly play the game without opening the Netbeans. The jar file names are listed below.

For tank war game:

csc413_tank.jar

For rainbow reef game:

csc413_reef.jar

b. Controls and Rules of the Game

For tank war game:

Tank1: W -> moving forwards; S -> moving backwards; A -> rotating left; D -> rotating right; Space -> opening fire.

Tank2: Up -> moving forwards; Down -> moving backwards; Left -> rotating left; Right -> rotating right; Enter -> opening fire.

The key controls for two tanks are listed above. Tanks can move forwards and backwards, and also rotate left and right, so that they are able to move in all directions. If only pressing the keys for rotating left or right, the tank will not move forwards or backwards.

Each tank has 100 health points, 5 basic damage points for each bullet, and at most 3 lives. When a tank runs out of its lives, the other tank wins, which is the only winning condition in this game.

There are 3 types of powerups, including extra damage, extra life, and health healing. Every time powerups will be generated as pairs, with random type and random positions. After each time interval, the method for creating new powerups will be called.

For rainbow reef game:

Seasnail (Katch): Left -> moving left; Right -> moving right.

There is no keyboard control for the starfish (Pop), and he can bounce freely around the area except the base. However, it can be controlled by the seasnail. Basically, the horizontal movement of seasnail will bounce starfish and prevent it from falling the base. The collision point along seasnail's shell will determine the direction of Pop's movement. To be more precise, bounces on the left half of seasnail will send it to left and bounces on the right half will send it to right. If bouncing at the center, the starfish will move up vertically.

Every time starfish collides with an octopus (Bigleg), its movement will be slightly accelerated. As he becomes faster, the game will turn out to be increasingly difficult. The game has 2 levels, with an initial of 3 lives. When running out of all the lives, the game is over. It is noteworthy that only those octopuses are required to be cleared to pass a level (an octopus can be killed with 2 hits). In other words, if the starfish defeats all octopuses, the current level is considered as finished, no matter how many remaining reefs there are. If passing level 2, there will be a congratulation page showing on the board.

There are 2 types of powerups, extra life and extra score, which are generated at top of the game board to increase the difficulty of acquiring them. The ExtraLife refers to the block with a heart in it. Each of them will add the lives count by 1 (at most 5 lives). The ExtraScore refers to the block with purple center and white border. Each of them will add 500 bonus points (much larger than other reefs) to the total score.

VI. Assumptions in Design and Implementation

For tank war game:

Collision handling is one of the characteristics within this term project. The tank war game only checks the collisions between tanks and walls, and between bullets and tanks, and between bullets and walls. And it is assumed that there is no collision between the two tanks. Each gaming element, such as tank, wall, powerup, and bullet, is regarded as a rectangle. Hence, collision is defined as the case where two objects (rectangles) intersect each other. In this game, a tank will stop immediately when it collides with a wall, and every bullet will cause damage when it hits a tank or a breakable wall, whereas any unbreakable wall is immune to damage.

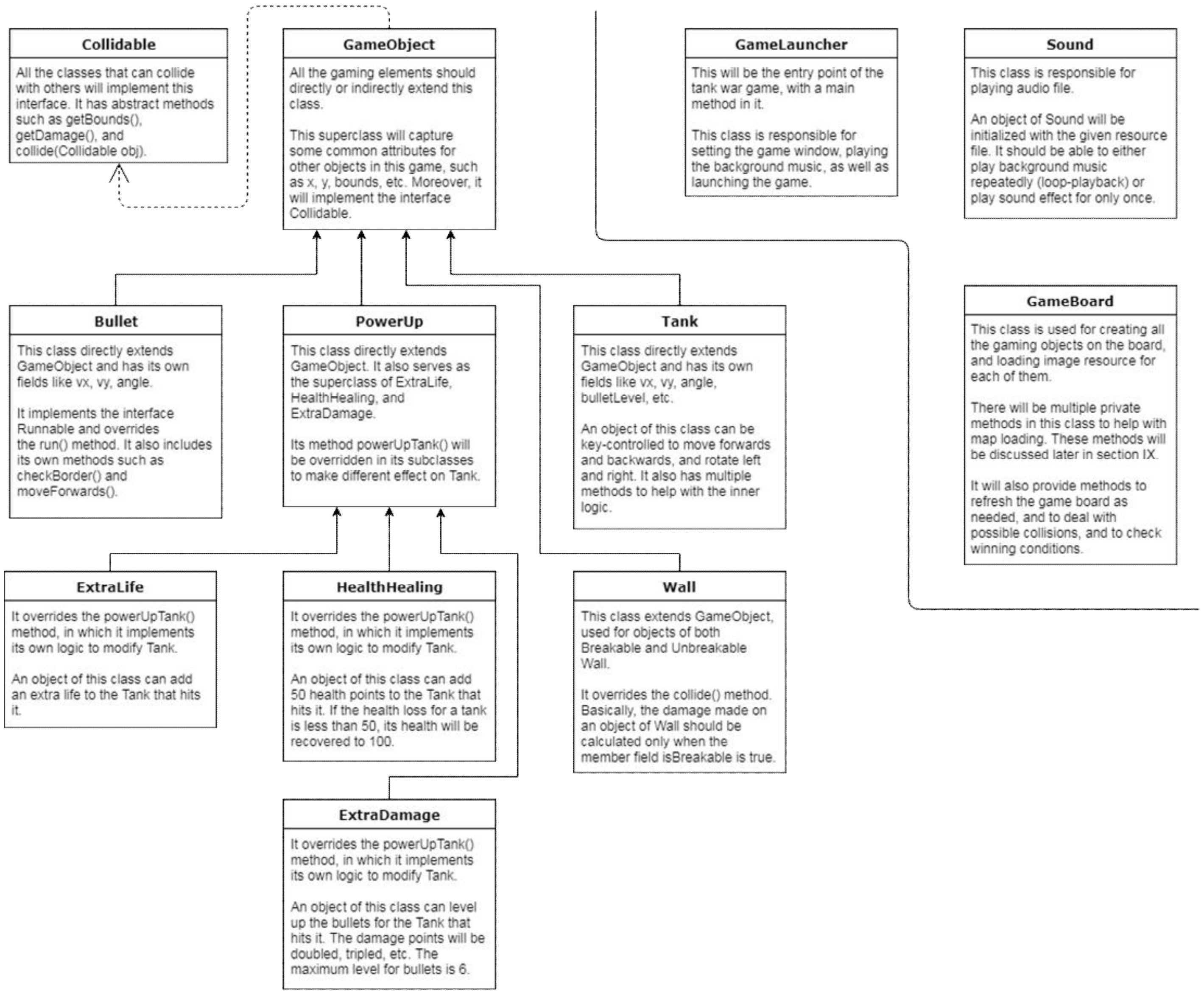
When a tank's health is down to 0, it loses a life and will then revive at the position where it was killed, as long as it still has remaining life. If a tank runs out of its lives, the other tank wins the war and all the bullets still running on the board will be cleared, with their damage calculation canceled.

For rainbow reef game:

The frequency of collision checking in rainbow reef is much higher than tank war. As a matter of fact, the starfish may collide with any other objects on the map. The border walls together with the black inner blocks are unbreakable and hitting them will not contribute to the total score. Every time it bounces the seasnail, its moving direction will range from 225° to 315°, depending on the collision position on that shell.

The initial speed for starfish is 6 in level 1, and will rise to 8 in level 2. Every time the starfish hits an octopus, its moving velocity will be incremented by 1, leading to increasing level of difficulty. Based on this assumption, the challenge of winning the game is not that easy as it seems to be, even though the number of keys used for game control is much less than tank war.

VII. Tank Game Class Diagram



Collidable

All the classes that can collide with others will implement this interface. It has abstract methods such as getBounds(), getDamage(), and collide(Collidable obj).

GameObject

All the gaming elements should directly or indirectly extend this class.

This superclass will capture some common attributes for other objects in this game, such as x, y, bounds, etc. Moreover, it will implement the interface Collidable.

GameLauncher

This will be the entry point of the tank war game, with a main method in it.

This class is responsible for setting the game window, playing the background music, as well as launching the game.

Sound

This class is responsible for playing audio file.

An object of Sound will be initialized with the given resource file. It should be able to either play background music repeatedly (loop-playback) or play sound effect for only once.

Bullet

This class directly extends GameObject and has its own fields like vx, vy, angle.

It implements the interface Runnable and overrides the run() method. It also includes its own methods such as checkBorder() and moveForwards().

PowerUp

This class directly extends GameObject. It also serves as the superclass of ExtraLife, HealthHealing, and ExtraDamage.

Its method powerUpTank() will be overridden in its subclasses to make different effect on Tank.

Tank

This class directly extends GameObject and has its own fields like vx, vy, angle, bulletLevel, etc.

An object of this class can be key-controlled to move forwards and backwards, and rotate left and right. It also has multiple methods to help with the inner logic.

GameBoard

This class is used for creating all the gaming objects on the board, and loading image resource for each of them.

There will be multiple private methods in this class to help with map loading. These methods will be discussed later in section IX.

It will also provide methods to refresh the game board as needed, and to deal with possible collisions, and to check winning conditions.

ExtraLife

It overrides the powerUpTank() method, in which it implements its own logic to modify Tank.

An object of this class can add an extra life to the Tank that hits it.

HealthHealing

It overrides the powerUpTank() method, in which it implements its own logic to modify Tank.

An object of this class can add 50 health points to the Tank that hits it. If the health loss for a tank is less than 50, its health will be recovered to 100.

Wall

This class extends GameObject, used for objects of both Breakable and Unbreakable Wall.

It overrides the collide() method. Basically, the damage made on an object of Wall should be calculated only when the member field isBreakable is true.

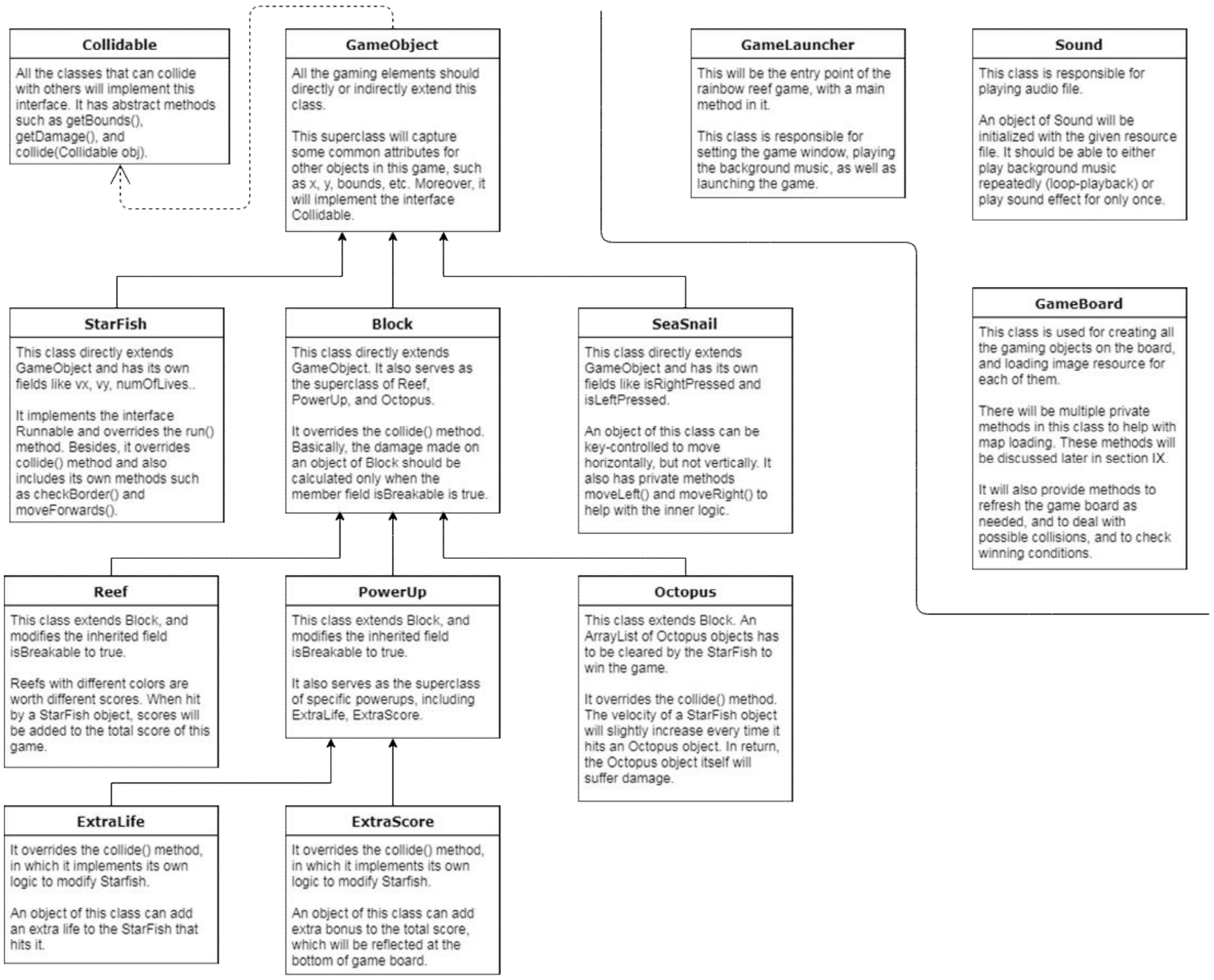
ExtraDamage

It overrides the powerUpTank() method, in which it implements its own logic to modify Tank.

An object of this class can level up the bullets for the Tank that hits it. The damage points will be doubled, tripled, etc. The maximum level for bullets is 6.

Figure 1. Class hierarchy for tank war game. Due to the limited space here, detailed information will be discussed in later sections.

VIII. Second Game Class Diagram


Collidable

All the classes that can collide with others will implement this interface. It has abstract methods such as getBounds(), getDamage(), and collide(Collidable obj).

GameObject

All the gaming elements should directly or indirectly extend this class.

This superclass will capture some common attributes for other objects in this game, such as x, y, bounds, etc. Moreover, it will implement the interface Collidable.

GameLauncher

This will be the entry point of the rainbow reef game, with a main method in it.

This class is responsible for setting the game window, playing the background music, as well as launching the game.

Sound

This class is responsible for playing audio file.

An object of Sound will be initialized with the given resource file. It should be able to either play background music repeatedly (loop-playback) or play sound effect for only once.

StarFish

This class directly extends GameObject and has its own fields like vx, vy, numOfLives..

It implements the interface Runnable and overrides the run() method. Besides, it overrides collide() method and also includes its own methods such as checkBorder() and moveForwards().

Block

This class directly extends GameObject. It also serves as the superclass of Reef, PowerUp, and Octopus.

It overrides the collide() method. Basically, the damage made on an object of Block should be calculated only when the member field isBreakable is true.

SeaSnail

This class directly extends GameObject and has its own fields like isRightPressed and isLeftPressed.

An object of this class can be key-controlled to move horizontally, but not vertically. It also has private methods moveLeft() and moveRight() to help with the inner logic.

GameBoard

This class is used for creating all the gaming objects on the board, and loading image resource for each of them.

There will be multiple private methods in this class to help with map loading. These methods will be discussed later in section IX.

It will also provide methods to refresh the game board as needed, and to deal with possible collisions, and to check winning conditions.

Reef

This class extends Block, and modifies the inherited field isBreakable to true.

Reefs with different colors are worth different scores. When hit by a StarFish object, scores will be added to the total score of this game.

PowerUp

This class extends Block, and modifies the inherited field isBreakable to true.

It also serves as the superclass of specific powerups, including ExtraLife, ExtraScore.

Octopus

This class extends Block. An ArrayList of Octopus objects has to be cleared by the StarFish to win the game.

It overrides the collide() method. The velocity of a StarFish object will slightly increase every time it hits an Octopus object. In return, the Octopus object itself will suffer damage.

ExtraLife

It overrides the collide() method, in which it implements its own logic to modify Starfish.

An object of this class can add an extra life to the StarFish that hits it.

ExtraScore

It overrides the collide() method, in which it implements its own logic to modify Starfish.

An object of this class can add extra bonus to the total score, which will be reflected at the bottom of game board.

Figure 2. Class hierarchy for rainbow reef game. Due to the limited space here, detailed information will be discussed in later sections.

IX. Class Descriptions of All Classes Shared by Both Games

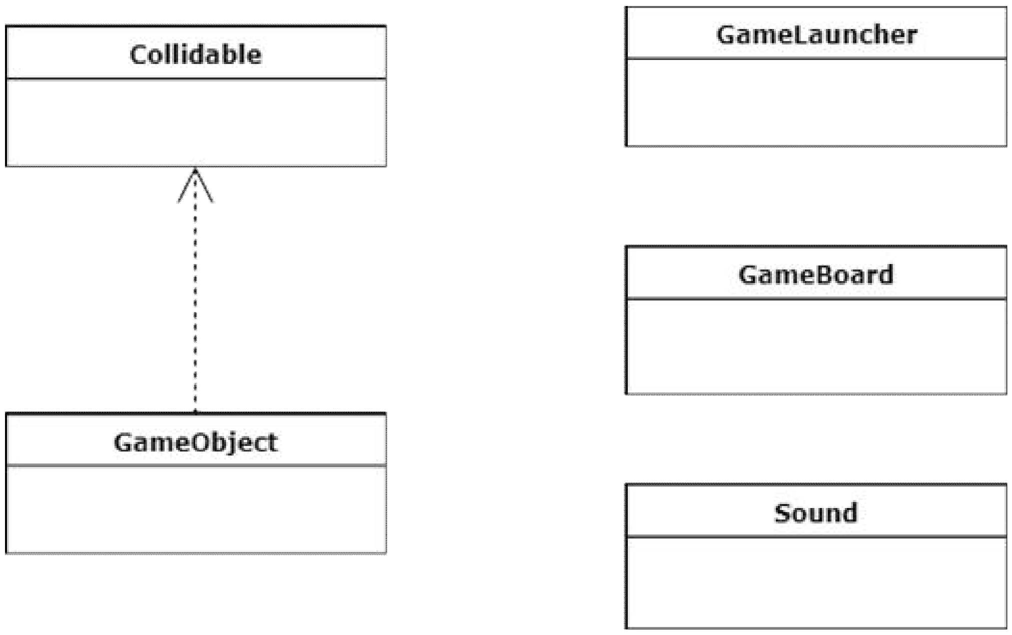


Figure 3. Classes shared by two games. Detailed information about their functions are discussed below.

As mentioned in section **II**, the reusability of code is the main focus during the development of these two games. Grounded on this consideration, the design of class hierarchy is intended to replace those large and complicated responsibilities into small and simple ones. To define classes that are highly cohesive, my design pattern firstly divides the required modules into two parts. The first part contains some specific functional classes, including Sound, GameBoard, and GameLauncher, which are shown above. The second part contains all the classes related to game objects, with GameObject as their superclass, which implements the interface Collidable. Within this project, Sound, GameBoard, GameLauncher, GameObject, and Collidable are shared by both games.

Class Sound is working as an audio player, which can be used to handle sound effect or music. An object of class Sound could be initialized with an input audio file, usually in lossless wav format. It has a member function play(), with a single boolean parameter to clarity whether it is loop playback. In this way, this class can support either playing background music continuously or playing sound effect for only once.

GameBoard is responsible for loading all the elements on the map, together with a split screen and a mini map. Its member function init() will be invoked to initialize all the gaming elements on the board, such as the tanks, walls, powerups in tank war, or the seasnail, starfish, reefs in rainbow reef. Then the resource images for each of them will be loaded on the map with the call of loadMap(). One of the important functions provided by this class is collision handler. Basically, the method checkCollision() will take a movable object as reference and scan the possible collisions with other objects by using collide(). Within the collide() function, it will retrieve the bounds (of Rectangle type) for two objects and invoke intersects() method in class Rectangle. If there is overlap between the bounds, this will be confirmed as a collision. In this case, both objects involved in the collision will call the collide() method, which they have already overridden after implementing the interface Collidable. Furthermore, GameBoard also implements the interface KeyListener and Runnable, and therefore overrides keyPressed(), keyReleased() to add keyboard controls, and overrides run() to refresh the game board with a specific time interval. During its running, the winning condition will be continuously checked to determine whether the game should continue or not. To reduce the complexity for debugging, the process for loading game objects is separated as several steps, with each extracted as a private function. They will work together to build up the board, and it is relatively easy to figure out where the problem is when something goes wrong on the map.

GameLauncher serves as the entry point of each game, with a static main function in it. It will create objects for both GameBoard and Sound, and set the size and title for game window, as well as add key listener to game board for movement control, and then launch the game for playing.

The interface Collidable is specifically designed to handle collisions between game objects. Since there are too many possibilities for two objects to collide with each other, there is no effective way to distinguish what the objects are and modify their states (e.g., health, isAlive, etc.) according to the collision. This can also be explained by Dependency Inversion Principle in OOP, which means that implementation should rely on abstraction, not the opposite. Collidable only declares abstract method getDamage(), getBounds(), and collide(), and does not care about which two are involved in each collision. It just requests the objects to execute their own method to complete the modification caused by collision.

Known as abstraction, this interface could significantly reduce the coding workload. Otherwise, if without this interface, the collision monitor will turn out to be a problem of permutation and combination, which may lead to non-portable hard code and make code maintenance impossible.

As the root for all gaming elements in each game, class GameObject defines some common properties such as x, y, health, isAlive, which are shared by its descendant classes. Apart from this, it declares bounds, a member variable in Rectangle type, to refer to a rectangle area that each game object currently occupies. It also directly implements the interface Collidable, and overrides abstract methods getDamage(), getBounds(), and collide(), which will be modified by its subclasses to deal with collisions. There is a static initializer in this class for storing all the image resources in a static HashMap, by using descriptive String (e.g., "Tank", "ExtraLife", etc.) as the key to retrieve corresponding BufferedImage.

X. Class Descriptions of Classes Specific to Tank Game

Due to the uniqueness of tank war and rainbow reef, it is inevitable that they have their own inner logic about gaming. Most difference about class hierarchy is reflected in gaming elements. As shown in the Figure 4 below, the classes related to game objects are specific to tank war game, except for the GameObject itself.

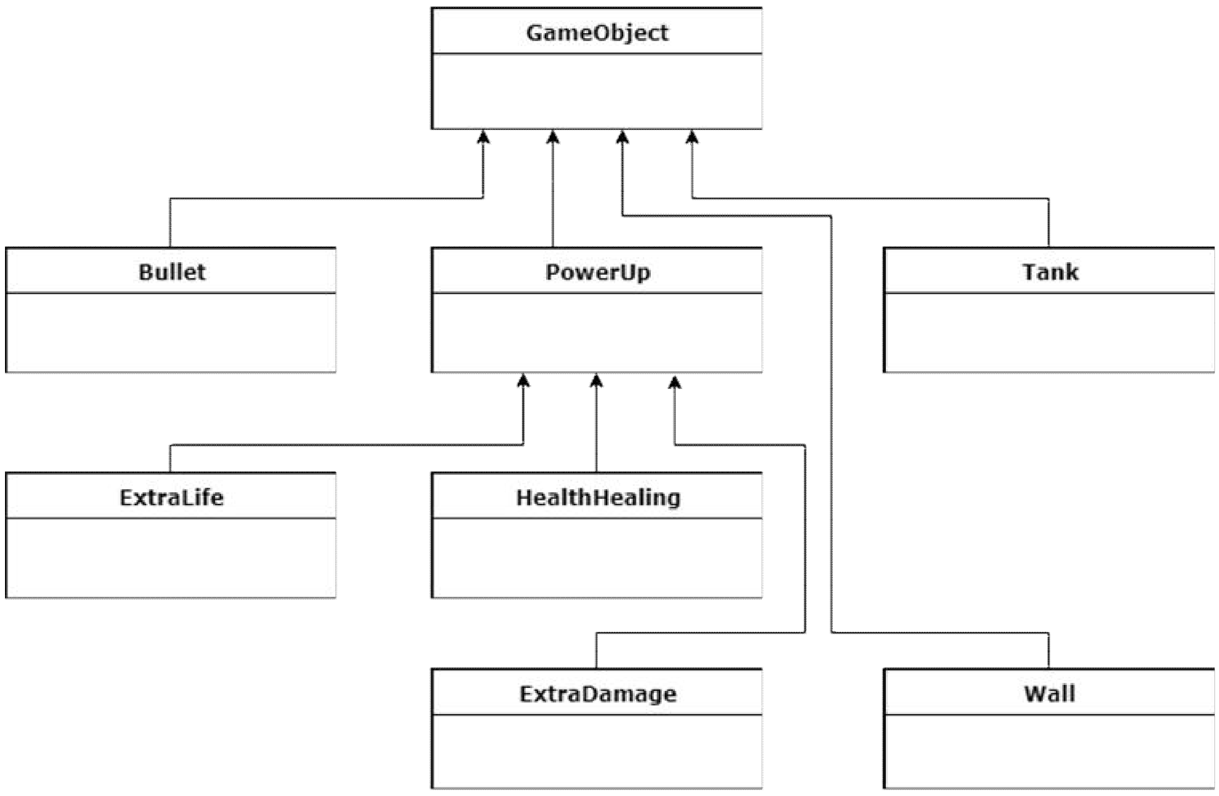


Figure 4. Classes specific for tank war game. Detailed information about their functions is discussed below. GameObject is also included here to maintain the completeness of inheritance hierarchy.

Tank could be seen as the most complex one among all the subclasses of GameObject. And this is not because of the amount of code, but rather the implementation logic for the control of tank movement. There are five operations that can be applied to a tank, and each one is extracted as an individual function. They are moveForwards(), moveBackwards(), rotateLeft(), rotateRight(), and openFire(), respectively, with their names clarifying what they do with the tank. In this class, the member field vx, vy, and angle are working together with several boolean variables to manipulate the tank. To be more precise, the boolean variables are monitoring the keys to decide whether to execute specific control, while the x-y positions are calculated through the combination of vx (velocity along x-coordinate), vy (velocity along y-coordinate), and angle (rotation angle). In addition, member variable fireRate and fireDelay utilize modulus in math to provide a time-delay manner for bullet shooting in order to avoid laser-beam firing. The method checkBorder() will be invoked when the tank hits an object of Wall, and will prevent the tank from moving until there is no collision with the wall.

Since bullets are running freely after created by a tank, Bullet is defined as an independent class, which implements the interface Runnable. Once the tank starts to fire, new bullets will be created as threads, so that they can move forward independently until it collides with other objects to trigger damage. To distinguish the tank to which a specific bullet belongs, there is a LinkedList container in class Tank to store all the bullets it has created. As long as the bullet does not hit any walls or the enemy tank, it is still considered as alive and will then keep on moving forward by repeatedly calling the method run().

As the superclass for all kinds of powerups, PowerUp defines the function powerUpTank(), and the details about how to modify the state of a tank can be varied in its subclasses ExtraDamage, ExtraLife, and HealthHealing. An object of class ExtraLife overrides the powerUpTank() so that it will add the number of lives by 1 to the Tank object that hits it. This method is also redefined in class HealthHealing, in which it will increase the health of a Tank object by 50 points. If the health loss for a tank is less than 50, its health will be recovered to 100. In class ExtraDamage, such method will level up the bullets for the Tank, with a maximum level of 6. As a result, the damage points will be doubled, and tripled, etc. All these modifications made to a tank have positive effects and are valid until the game ends.

Wall directly extends class GameObject, and is used for creating objects of both breakable and unbreakable walls. It has a boolean member variable to notify whether the wall is breakable or not, and it also plays a key role in the overridden version of collide() method. Basically, the damage made on an object of Wall should be calculated only when the field isBreakable is true.

Although both share the class GameBoard, the tank war game has its unique features. For instance, it requires a split screen, with left and right half focusing on two tanks. Additionally, lives count and health bar are generated and moving with the tank as a whole. Thus, private methods such as checkScreen() and loadHealthBar() are only included in tank war.

XI. Class Descriptions of Classes Specific to Second Game

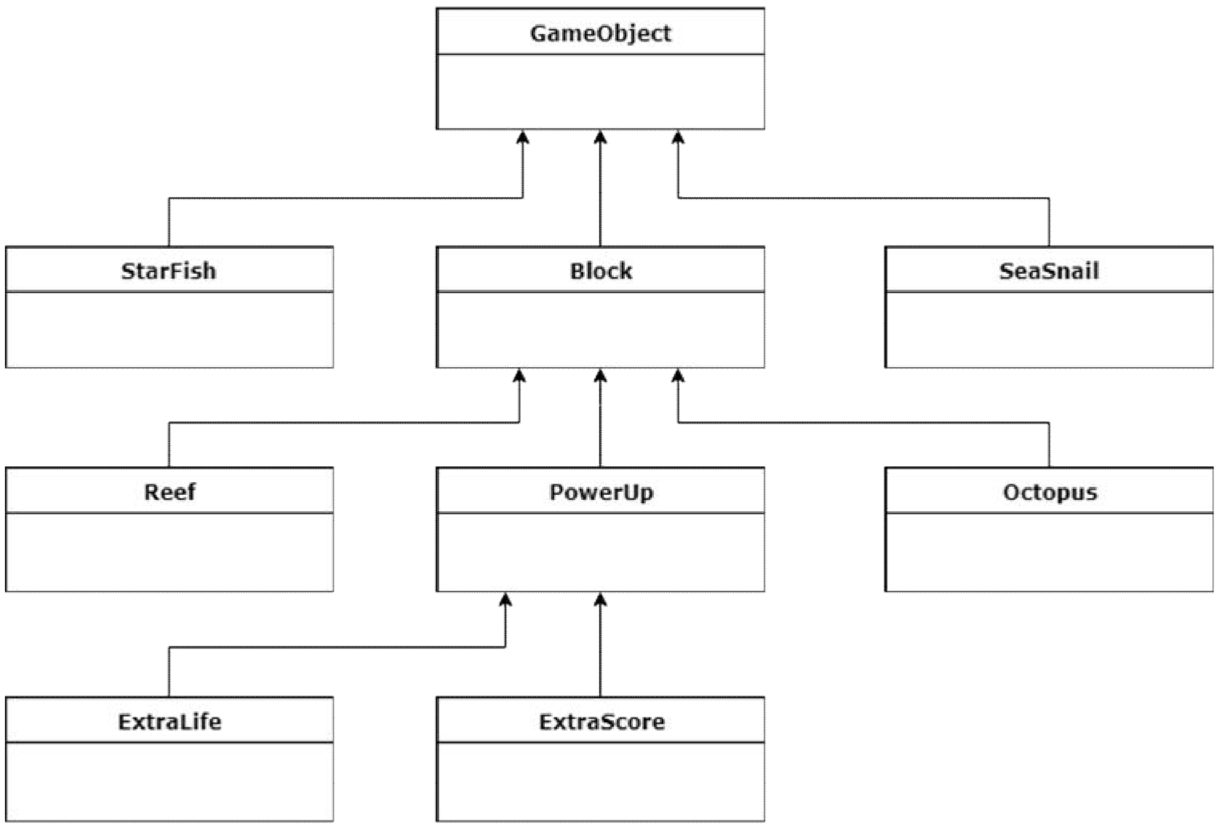


Figure 5. Classes specific for rainbow reef game. Detailed information about their functions is discussed below. GameObject is also included here to maintain the completeness of inheritance hierarchy.

As Figure 5 indicates, class SeaSnail is one of the gaming elements only available in rainbow reef. However, it can be taken as a partial copy of the class

Tank in tank war, due to the comparability between these two. Technically, SeaSnail is actually a simplified version of class Tank, since it only needs the control over horizontal movement along x-coordinate, but excludes the rotation and the vertical movement.

Given that map loading and keyboard control are relatively easier than tank war, class StarFish is the most difficult module during the development of the second game. This is understandable since StarFish plays the most important role in rainbow reef. It not only serves as the "weapon" to defeat enemy octopuses, but also triggers large amount of collisions. When taking that "weapon" property into account, it is analogous to the class Bullet in tank war, especially in terms of implementation logic. Based on such analogy, we may regard it as a "bullet" that always moves forward, while leaving the collision handler to determine which direction is "forward". To be more specific, starfish will be created as a thread in the game running process. Its member function checkBorder() and the method collide() that it overrides will cooperate to dictate where it should go.

Block in rainbow reef is comparable to the class Wall in previous game, but still with some difference. For instance, an object of Block is unbreakable, similar to the unbreakable wall in tank war. Besides, Block undertakes multiple tasks in class hierarchy. Instead of simply extending the GameObject by adding health as its member field, it also functions as the superclass for Reef, Octopus, and PowerUp. It modifies the collide() method in interface Collidable, and the effect of this overriding could be explained from two aspects. For one thing, it defines whether the damage calculation should be taken and how it would be taken on a Block object. For another, this version will not be fully replaced in the subclasses, but will be invoked as part of their implementation to reduce duplicate code.

As one of the subclasses of Block, Reef changes the member variable isBreakable to true, making each reef breakable. When overriding the collide() method, it calls super.collide() at first for damage calculation and then requests StarFish to add certain points to the total score it has achieved.

Although the object of Octopus may look different, there is in fact nothing special for this class. Its object can be treated as a Block, except that it has bigger size and will request StarFish to increment its moving speed every time after collision.

PowerUp herein becomes rather simple. Since its parent class Block has already clarified how to handle collision, the only thing it has to do is setting isBreakable as true to share this property among concrete powerups. With most preparation done in ancestor classes, ExtraLife just needs to add an extra life to the starfish object, while class ExtraScore modifies StarFish by adding extra bonus to the total score.

Similar to tank war, class GameBoard in rainbow reef has its own unique methods. For instance, each Octopus object occupies space twice as large as other blocks or reefs. Moreover, the unbreakable blocks and breakable reefs on the game board are featured with regular pattern. Thus, multiple private methods are designed in GameBoard to generate these game objects with interleaving order. Additionally, the rainbow reef game contains two levels, leading to a different method for monitoring the winning condition.

XII. Self-reflection on Development Process during Term Project

Having been tested for a lot of times, this project covers the requirements listed for two games and both can be played as expected under the assumptions mentioned in section **VI**. However, there is still defect within the design. For example, there could be a MapEditor class for user to edit the map as they want. Another optimization is to set a cut-off radius for tank and starfish, and then we use this radius to scan the possible collisions within a limited range. In this way, the time complexity for collision checking method can be lowered.

Despite of the drawbacks it may have, this project satisfies basic OOP principles. The tank game does prove its good reusability when its modules and implementing strategies are applied to the second game. Furthermore, it is extensible for future modification, such as involving other elements or powerups in the game. I regard this term project as a comprehensive practice for OOP design, where much effort has been devoted to class design and removal of code redundancy.

After completing the first game, code reusability has made significant contribution to the implementation process of the second one. Resorting to the class design pattern in tank war, responsibilities and functionalities in the second game can be clearly distributed in multiple modules (classes). As a result, compared to the time spent on tank war design, the amount of time for rainbow reef development has been greatly reduced.

XIII. Project Conclusion

Each game can be played by building and running the project in NetBeans or simply by executing the jar file in repo. Since the jar file is generated by using the latest version, both launching methods will have the same result.

For tank game, the operation for tank movement and bullet shooting works normally without any lagging, while the damage imposed on tanks or breakable walls is calculated in a timely manner. The tank-wall, bullet-wall, and bullet-tank collisions are correctly monitored as expected. With those randomly generated powerups, the tanks will be strengthened in terms of health and attack, contributing to the playability of this game. More importantly, this is a “portable” game, meaning that the experience acquired from class design and a good portion of programming code could be referenced by other 2D games during their implementation.

For rainbow reef, the movement control for both seasnail and starfish can be easily understood by users. Moreover, the gaming status (continuing, winning each level, or losing) is correctly checked. There may exist strange trajectories for starfish movement, when it bounces in the upper half of game board. This could be explained by the high frequency of unpredictable collisions among those adjacent blocks and reefs. Nevertheless, such case happens rarely. Hence, the second game is, by and large, a successful project by rescaling the code and classes in tank war game. The acceleration of starfish moving speed, combined with the increasing difficulty from level 1 to level 2, may challenge users’ manipulation and bring more fun.