Tank Game

CSC 413 TERM PROJECT

NAYAN PANDEY

https://github.com/csc413-02-summer2019/csc413-tankgame-nayanpandey2012

Project Introduction

The term project of tank game was implemented in Java. The game was based on various resources, features and functionalities. During the project, based on SRP(Single Responsibility Principle) I focused on try and minimize coupling while increasing the level of cohesion within my code.

It was difficult at first to tackle the introduction of coupling in heavy dependency in between classes. However,, focusing on OOP In mind, I was able to get the perfect class diagrams, which assist me to have a general overview of a reasonable approach. I often make draft class diagrams before I start any project of a medium to large size. Within this documentation, my compartmentalization and step-by-step approach will become evident as I will structure the document in the hierarchical fashion which I adhered to when writing the actual code.

The goal of the project was able to work in a OOP principle with a structure in such a way that I could easily modify and optimize the codes in future. For the project I have used following methodology.

Introduction of the Tank Game

This section is going to explain how the game works. Upon execution of tank game jar, the following screen with menu will pop up.

Players can press with their mouse the following options:

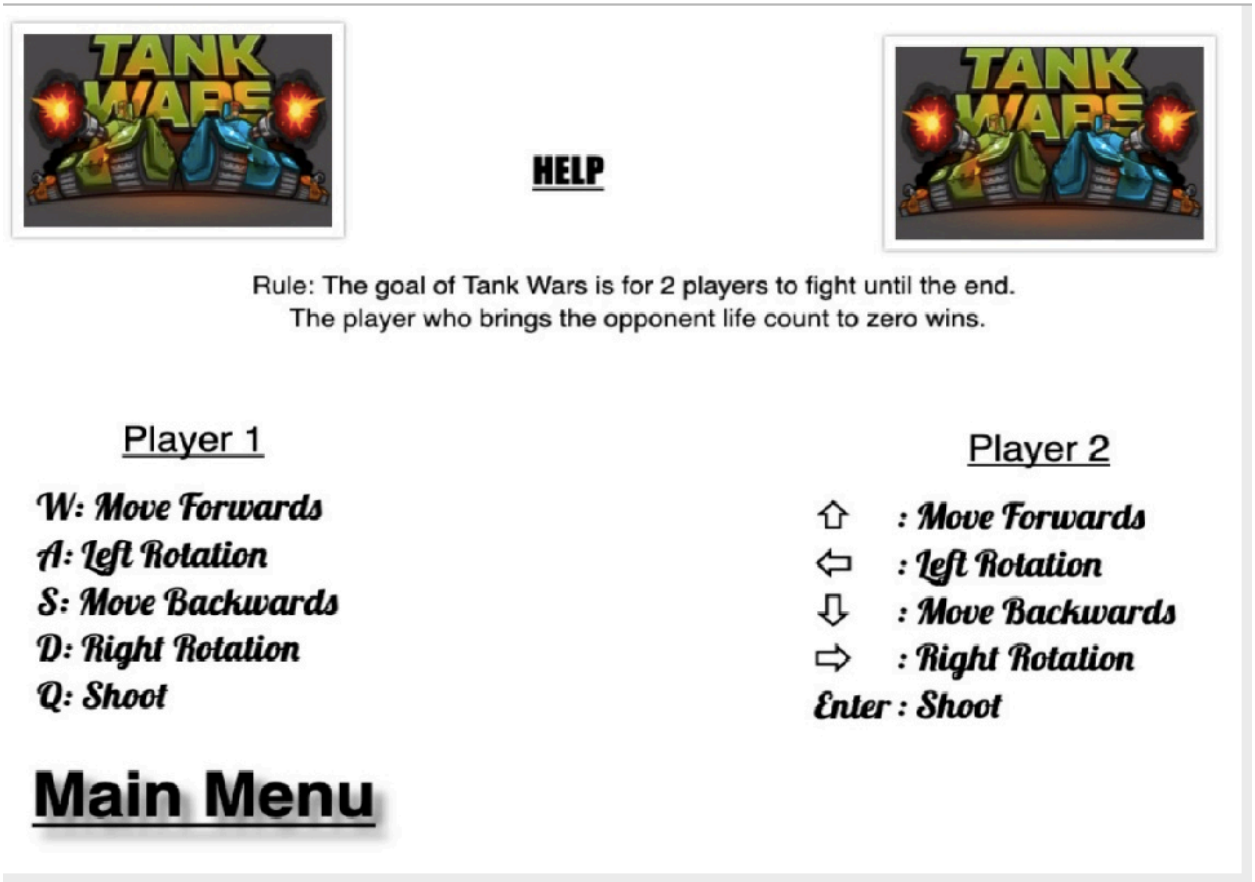
BEGIN! - To start the Game.

HELP – To know the controls and purpose of the game

EXIT – To get out of the game

All these options are implemented with mouse input , which we will discuss in the MouseReader and Menu class later on..

Pressing the HELP button, it will be directed in the following screen.



As mentioned the HELP button is provide the better understanding of controls and the purpose of the game.

Pressing the Main Menu, at the bottom left of the screen will bring us back to the earlier, Menu screen.

Now pressing the BEGIN! Button, we will have the following interface.

We will have two split screens. Each player have a screen, and each screen have the live counters and health bars, along with a mini map on the middle of the interface, which assists the players to strategized their positions. The surrounding perimeters are bordered by unbreakable walls. There are some significant amount of breakable walls also that the player are supposed to struggle before they could get some extra power up (heart) and speed up. Once the bullet are collided with these unbreakable walls they disappeared. On the other hand, Breakable walls also disappeared after getting two collision with this bullets.

As mentioned in the interface, each player will have 2 lives. Once a player lives count is zero, the opponent player win the game.

Development Environment

a) Version of Java used: Java 10.0.2

b) IDE used: IntelliJ Idea

c) Note on resources: Most of the resources used in this project were provided on ilearn. Some of them are downloaded from various websites. All rights and licensing belong to their respective owners.

How to build and import the game

I have included all the screenshots that required in order to import and run the game within the IntelliJ Idea IDE.

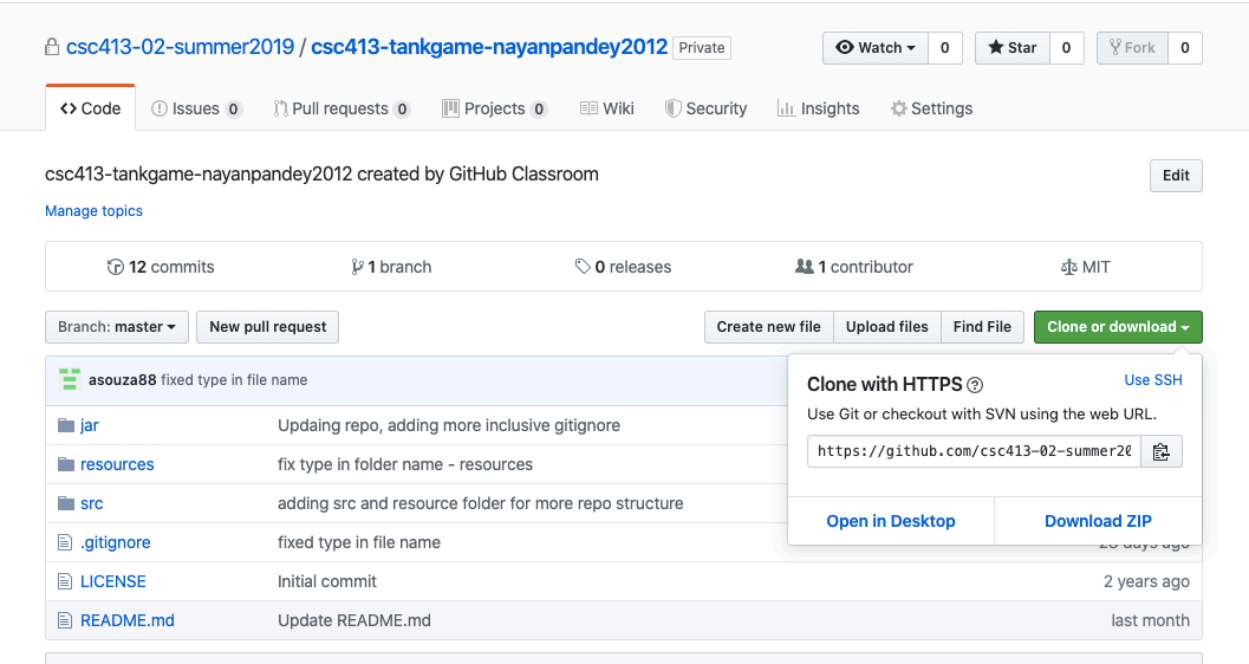


Figure Go to the repo link and click the green “Clone or download” button.

I went to the repo link and copy the clone link, which later on I cloned from the Terminal.

After cloning, I open the IntelliJ Idea IDE, and clicked “Import Project”, where on the next screen, select where you extracted the zip file and press “OK”.

.

Figure Launch the IntelliJ Idea IDE and click “Import Project”. On the next screen, select where you extracted the zip file and press “OK”.

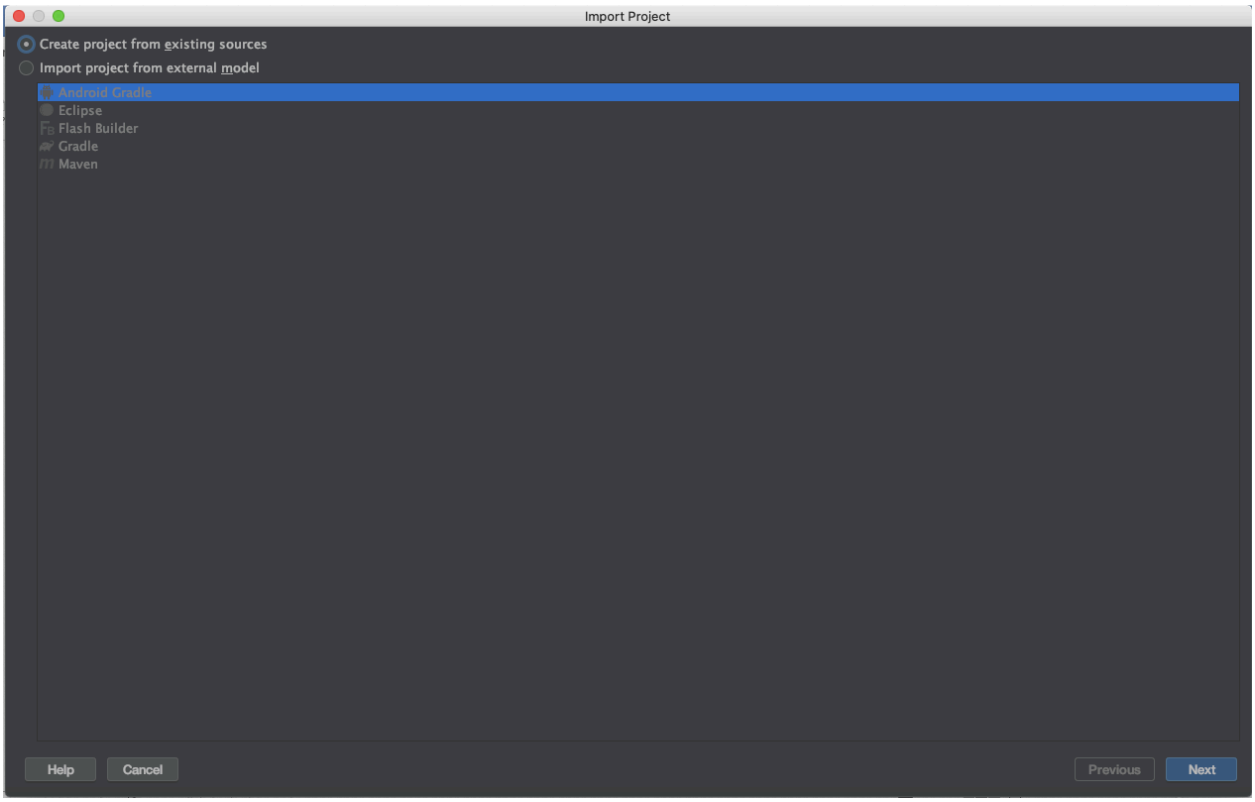


Figure: Click “Create project from existing sources and press “Next”.

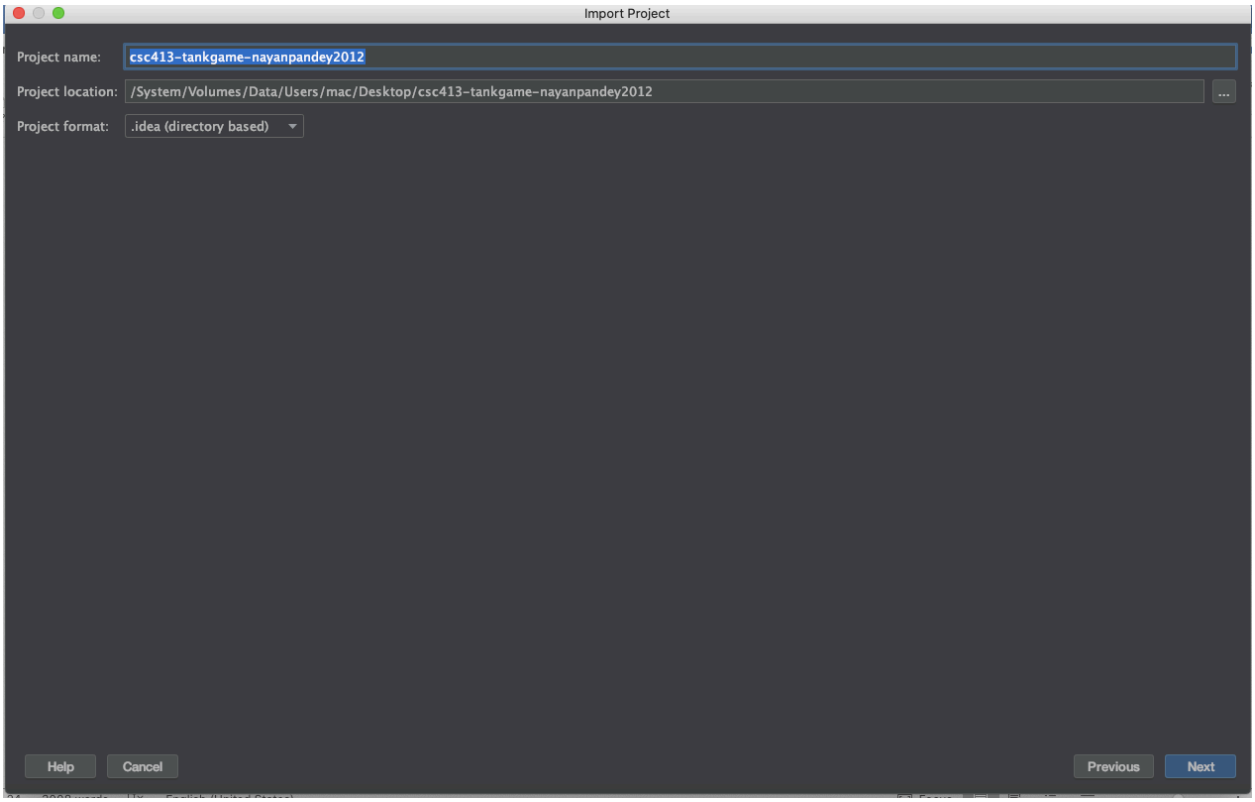


Figure: Verify that the Project format matches the figure and press “Next”.

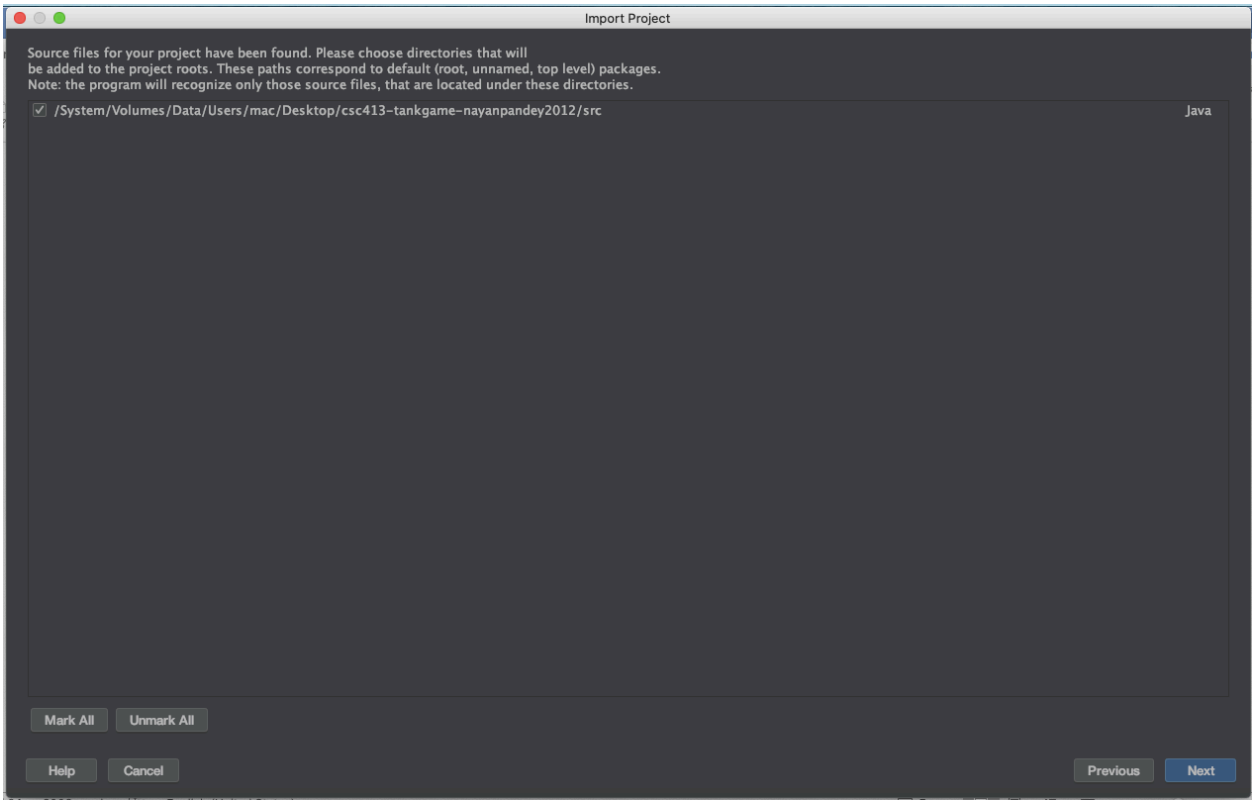


Figure: Click “Next”.

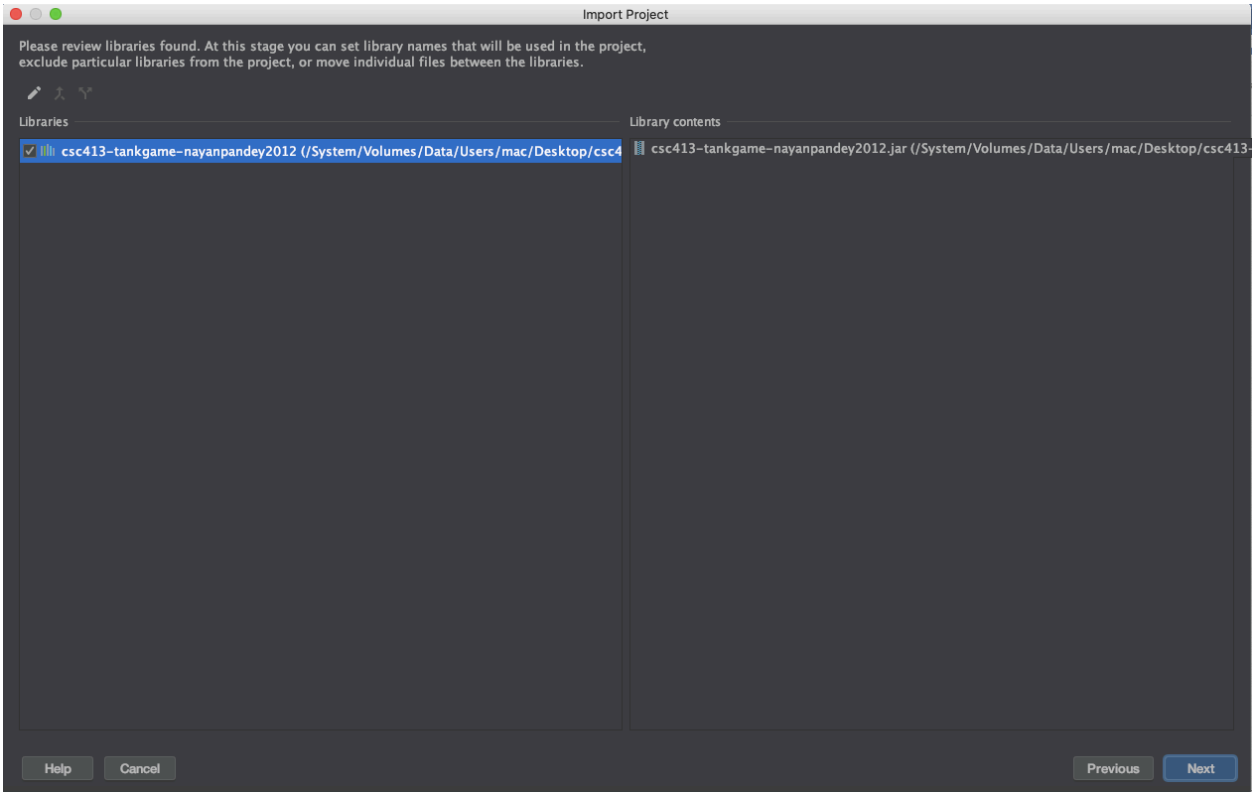


Figure: Click “Next”.

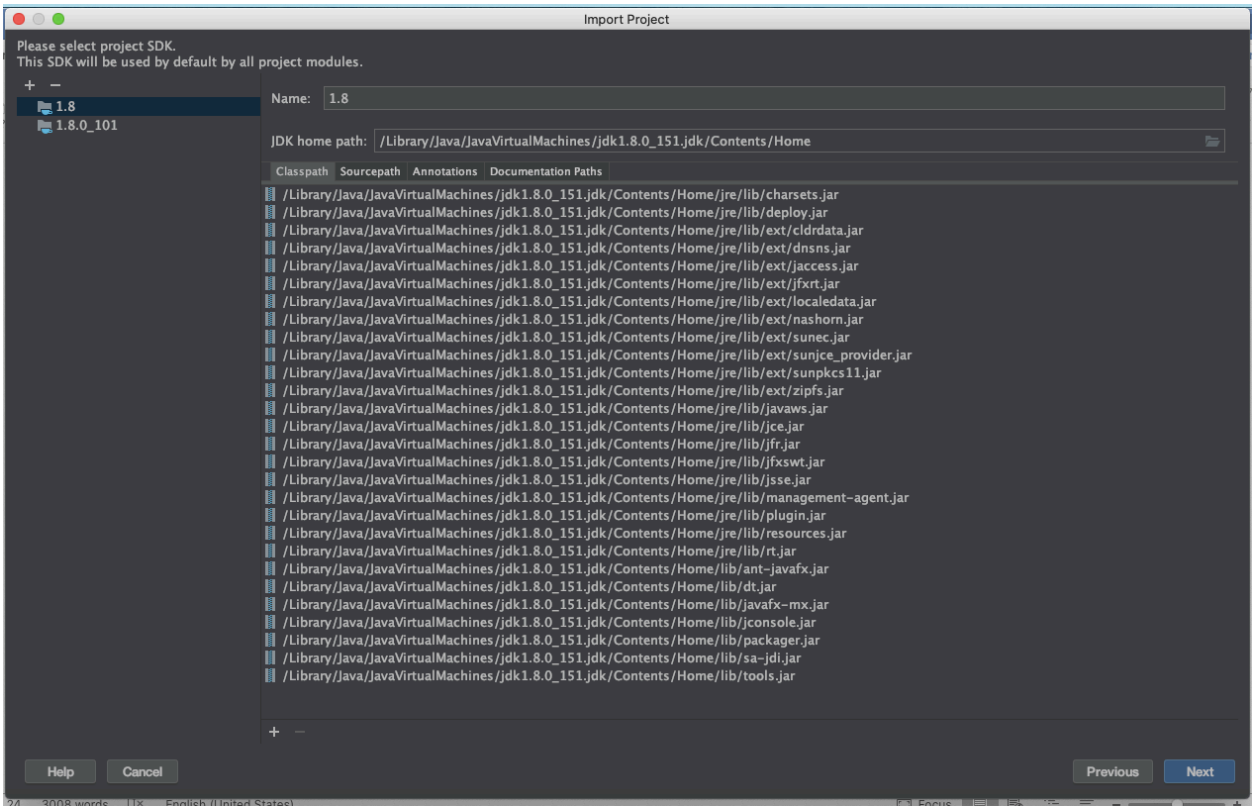


Figure: Click “Next”.

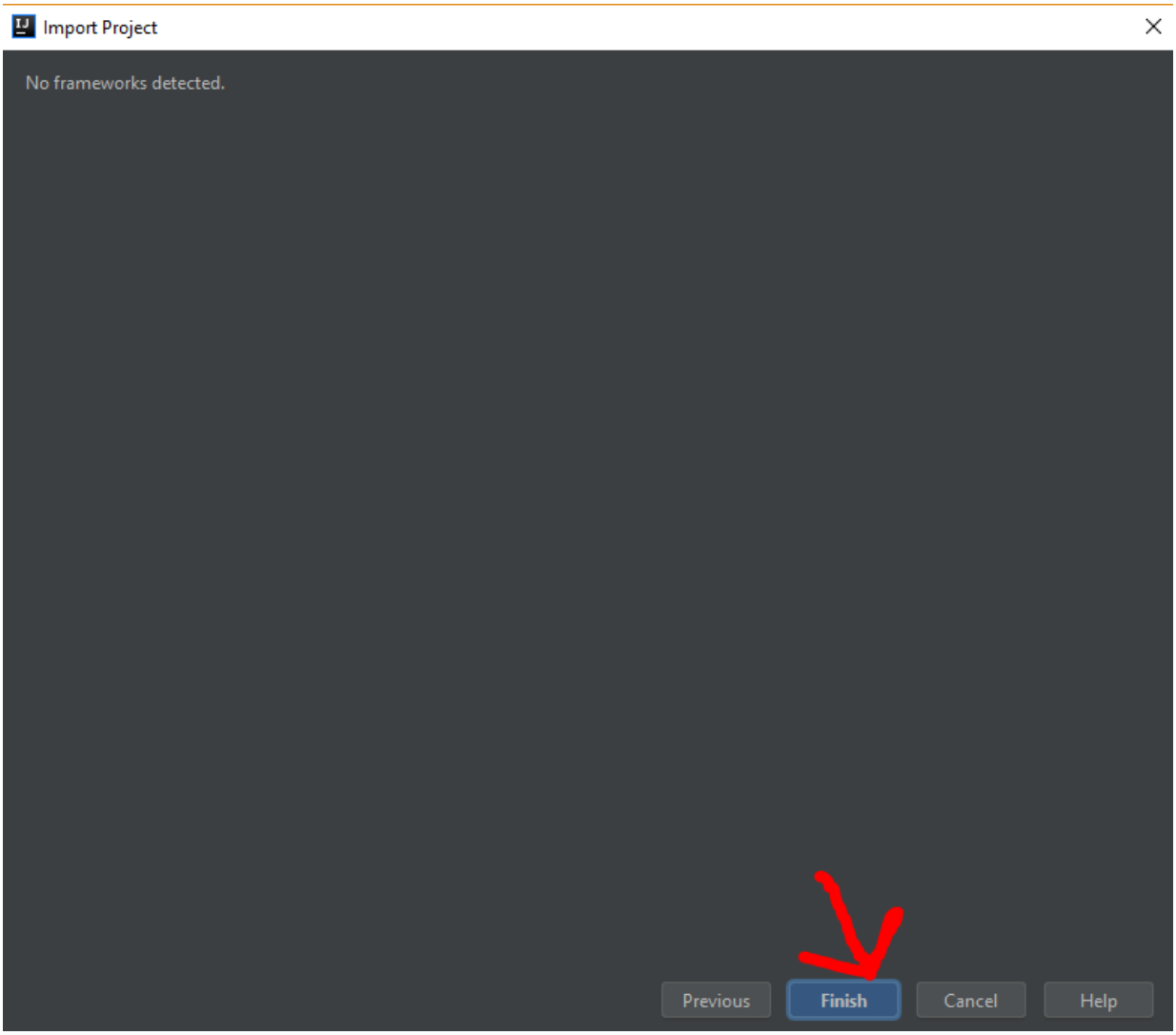


Figure: Click “Next”.

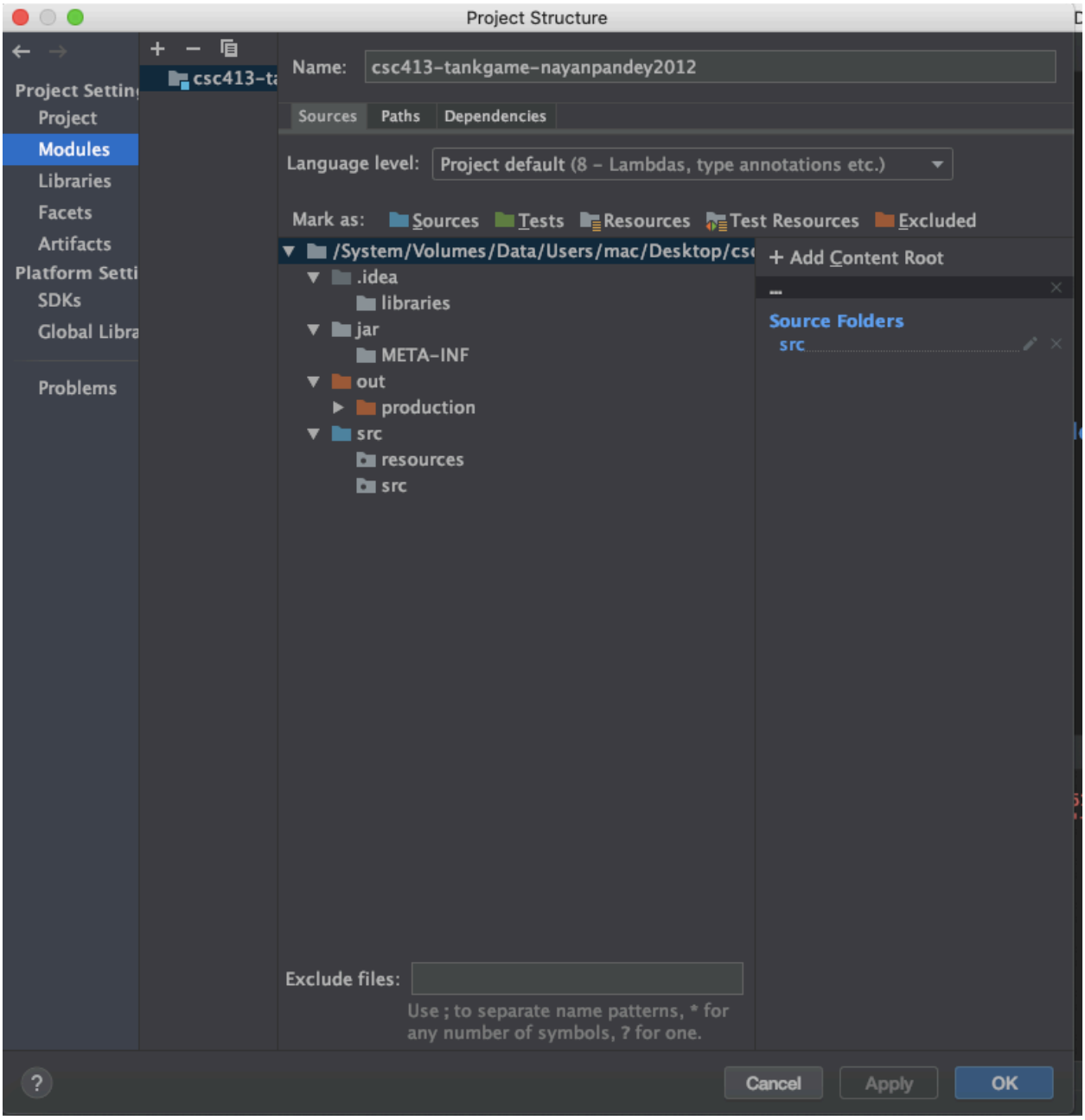


Figure: Go to the project structure window(ctrl-alt-shift-s) and make sure that the src folder is selected as a source and that the src/resources folder is selected as a resource folder as shown in the figure.

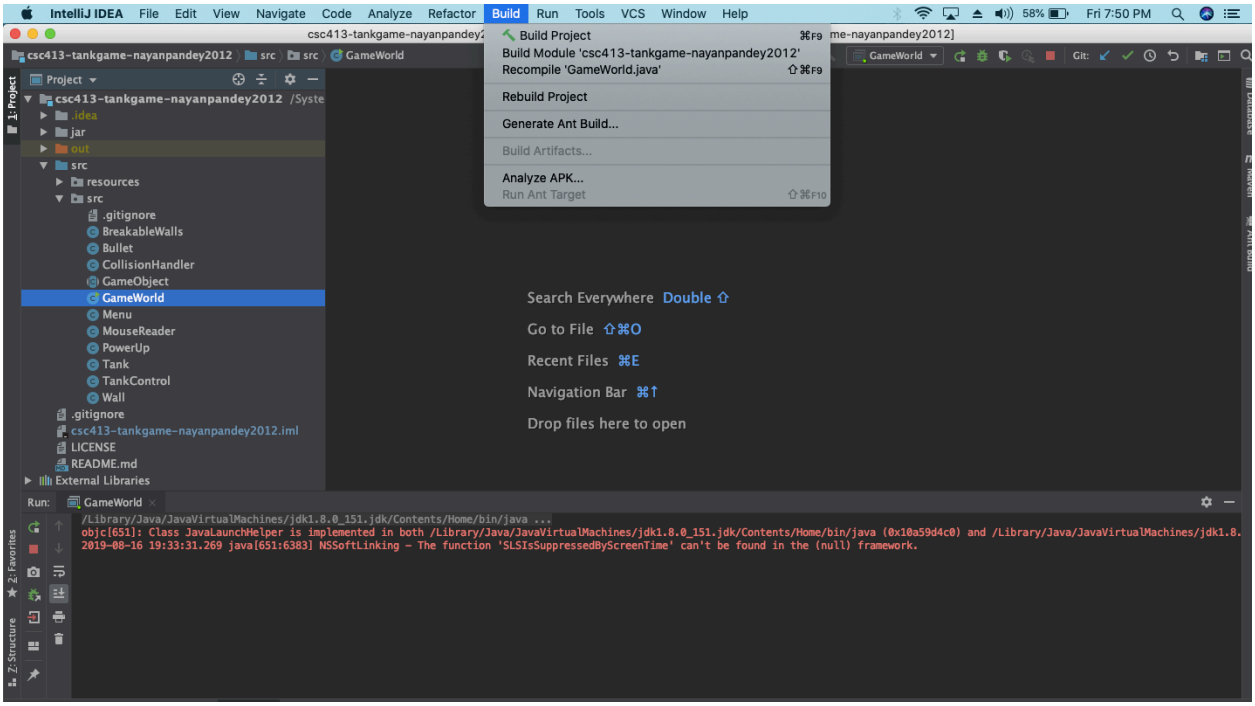


Figure: Click “Build” and then “Build Project” to build the project.

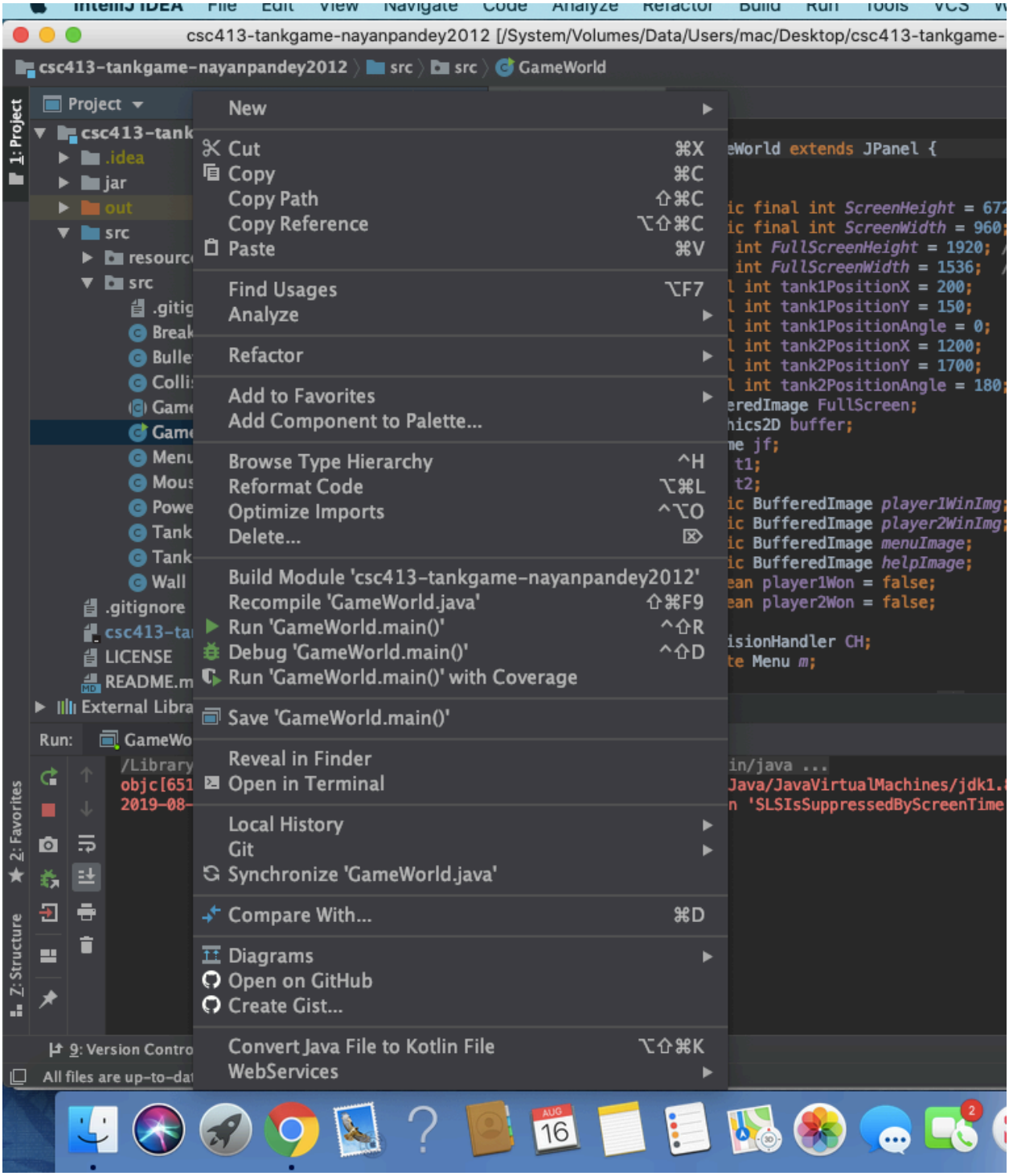


Figure: Right click on "**.**GameWorld" and click "Run 'GameWorld.main()' ".

Figure: We can see that the game runs successfully.

Building jar: I used IntelliJ's Jar building feature to build my jar file. I did this by navigating to the Project Structure, selecting artifacts and creating a new jar file with the rebuild each time option selected. This ensures that a new jar is generated in the appropriate location(jar folder) each time.

Commands used to run the jar: The easiest way to run the jar is by double clicking it on a machine that has the proper version of Java. The java – jar [put filename here].jar can be used where the square brackets are removed to run the jar on the command line. I used Windows PowerShell for testing.

Assumptions Made

I actually didn't make very many assumptions during the design and implementation phase of this project. I preferred adding walls before the player, so I only check for collisions of Wall to player since my inner loop only starts at that initial point. I could've added identical methods for the reverse cases but I personally didn't feel that it would be necessary at the time. My other assumption is that the player will not be allowed to move at an angle in the second game. This is done to preserve the retro/block feel of the game and to make it more challenging. Please note that I intended for each appropriate button to be pressed for movements on its own. For example, a player should hold down the W key to keep moving up and release and press another key to turn

directions. This assumption allowed me to make my code simpler and to focus on the more important(OOP principles) aspects of the project.

Tank Game Class Diagram

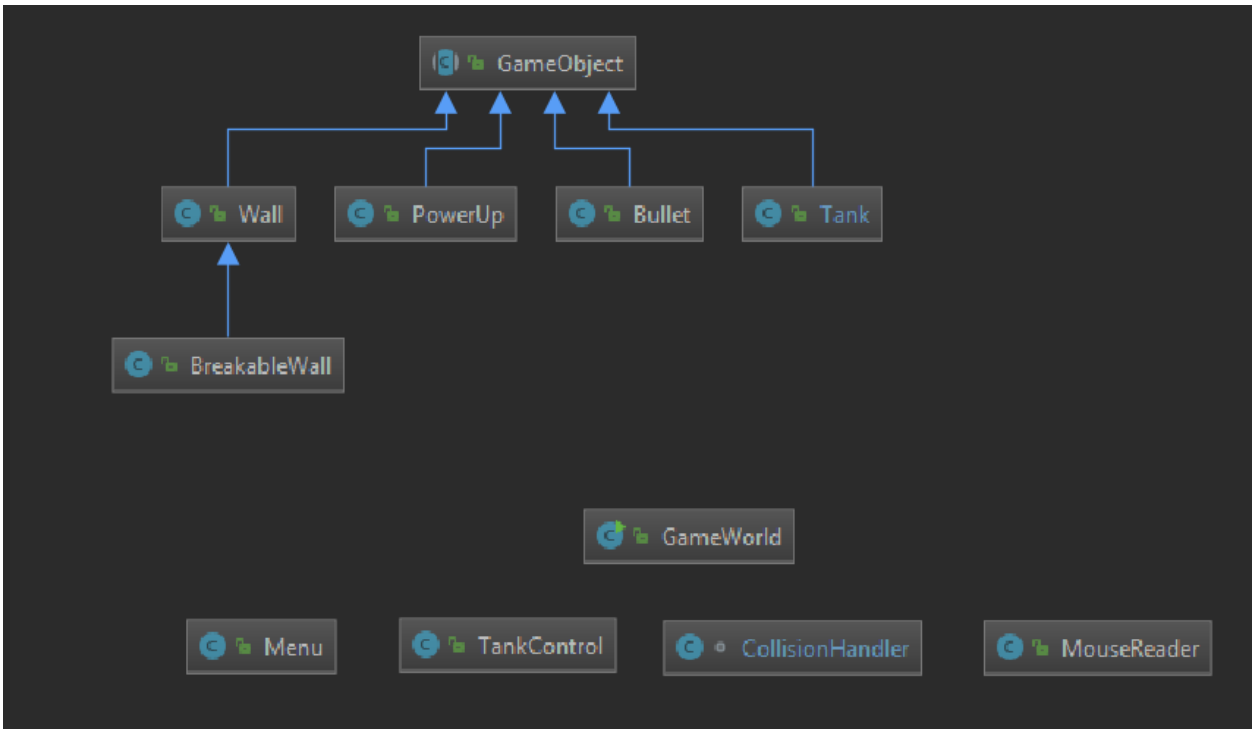


Figure (8): This figure shows the Tank Game class diagram.

The core abstraction of the class is in the abstract GameObject class which is extended by various other classes(Wall, PowerUp, Bullet, Tank, and BreakableWall(indirectly)). The GameObject class provides base functionality such as x and y positions as well as angle and x and y velocities. The Wall class is used to make border/background walls while the BreakableWall class is used to create BreakableWalls. Therefore, the BreakableWall class has a health associated with its instances. The PowerUp class consists of the Health Boost and the Speed Boost. The Tank class doesn't store bullets but is used to trigger the launch/creation of a bullet via the Bullet class' constructor. TankControl is used to allow for control of the Tanks via appropriate keyboard keys. The Menu class controls the displaying of the shapes and text used for buttons in the main menu. The CollisionHandler handles collisions between various objects within the game when applicable. The MouseReader class is used to read mouse input and select the appropriate game state based on which button is clicked. I was very careful to ensure that button areas don't remain "clickable" after a user's choice has been completed. The GameWorld class controls the status of the game as a whole. These classes will be described in great detail within the future sections of this documentation.

Class Descriptions.

- **GameWorld**: This class is the entry point of our program and handles creation along with maintenance of the game. A menu was implemented in the Tank Game, so an enumeration

variable was used to store and monitor the current state of the game. The GameWorld class has a plethora of variables, so I won't describe each and every one in great detail. Rather, I will focus on the overall responsibility of the class and how it carries out that responsibility while adhering to good OOP principles and the SRP(Single Responsibility Principle).
The init() method of GameWorld is used to read in and set the many images used to provide animations and various displays within our project. I personally decided not to use sprite sheets, so I am storing BufferedImages into well-organized arrays which will be described in great detail within the next sections of this documentation. The init() function is also used to create the map of the game which is stored as a single array. This map is extremely large coming in at a size of 2528x1920 pixels, but such a size is required because I spent a lot of time in ensuring that the map is symmetrical and strategically designed. The init() method is also used to initialize all other map/game related items such as the player, player control, creatures, pickups, etc.
The drawImage() method is used to display the game as a whole. It handles centering the display on the player, displaying the number of lives/scarabs, displaying the score, and showing the proper images if a player wins or loses. It is very important to note that all of the previously discussed functionality is done in conjunction with other classes via the use of getters to receive the states of the player.
The main method is used to loop over an ArrayList of game objects and update them. It is also used to set the appropriate boolean if the player is dead or needs respawning. It also calls repaint to constantly update the display.

- **GameObject**: This class is where my concepts of abstraction really shine through. This is an abstract class that holds x and y positions along with x and y velocities. It also holds a Rectangle which is crucial when it comes to my implementation of collisions. There are getters and setters for all of the previously mentioned variables except the Rectangle which only needs a getter because we set it within its respective class.
  The GameObject class includes 3 abstract methods: update(), drawImage(Graphics2D), and collision() which are used by all of the GameObjects in our game. This generalization may not seem like a big deal but it has huge implications when it comes to reducing the amount of duplicate code. It also offers flexibility because we can simply invoke the update() function on a GameObject instead of having to call its own separate functions that could accomplish the necessary tasks.

- **CollisionHandler**: This class has a single method (HandleCollisions(ArrayList<GameObject>)) which takes in an ArrayList of GameObjects, handles the collisions between them, and returns a new ArrayList of GameObjects. The CollisionHandler class varies slightly between the Tank Game and the second game in the sense that it obviously checks for collisions between different objects, but the core functionality of the class remains consistent. The CollisionHandler class uses a nested for loop to get an object and compare it with all objects after it in the list. It uses the instanceof operator to determine the type of objects. Getters for the GameObject Rectangle are used to receive rectangles and compare them for intersections with the built in intersects() method of the Rectangle class. Appropriate handling is done for each different occurrence of a collision.

- **Wall class**: The wall class is a rather simple class that allows for the creation of Wall objects. It is very important to note that this class extends the abstract GameObject class and thus implements all of its abstract methods. This class has minor differences in both projects, but I will explain them here as the core functionality is the same. The Wall class has a boolean variables used to denote which image should be used when the drawImage function is called. In the Tank Game, there is a single boolean variable to denote if a particular instance of the Wall class represents a background image or a regular wall. Since the wall class extends GameObject, each Wall has an x and y position which is initialized via the Wall constructor. The Buffered Images within the Wall class as private static fields objects because it is inefficient for each object to hold its own Buffered Image. For example, I have hundreds of walls with the same type in my second game and it would be completely unnecessary for each one to hold its own identical image. Package private setters are used to set the images of the Wall class. These are called from the GameWorld which is the entry point of our code

- **Tank**: This class is a really important class. It extends the GameObject class and adds a lot of functionality to it. We use the proper boolean variables which were set in TankControl to move the tank in the update method. A SpawnBullet() method exists to spawn bullets with a given x, y and angle. It is important to note that there is a restriction on how often a player can call this method due to the LastFired variable which holds the last time a bullet was fired. Players can only fire once per second.
  The class has methods for rotation and for moving in various directions. A boolean variable is used to check if a tank is currently speed boosted. If speed boosted, it travels at roughly 4 times the original speed. We also have a time variable to control the speed boost and to ensure that it is shut off after 1 second. I feel confident that this class is very much so in line with the Single Responsibility Principle.

- **Bullet**: This class extends GameObject because it adds and implements functionality to the abstract GameObject class. A string is used to denote the owner of the bullet while a isInActive boolean variable is used to mark whether the bullet is inactive. Inactive bullets are removed from the game_objects ArrayList in the main method of GameWorld. We store 3 images as static fields in this class: bulletImage, largeeExplosionImage, smallExplosionImage. The bulletImage, holds an image of the bullet as the name suggests while the small_explode is used to display the image of explosion (appears when a BreakableWall is shot). The large_explode image is used to display the image of a large explosion which occurs whenever a tank shoots another tank. Bullets are marked inactive if they cross the boundaries of the game map. This prevents us from holding unnecessary Bullet objects in our game_objects ArrayList. We wait a couple iterations after a Bullet collides with a tank before marking it as inactive to ensure that there is ample time for the explosion image to display.

- **PowerUp**: This class extends GameObject since it has a x and y location as well as an image and a Rectangle. I only had two possible pickups, so I decided to use two boolean variables(isHealthBoost, isSpeedBoost) to allow for control over which image should be shown and how collisions can be handled. In the collision method, we simply set a boolean variable isActive to false. This allows us to later remove the used up PowerUp object.

- **BreakableWall**: This class adds some slight functionality to the Wall but it is different enough to suggest that having its own class would be justified. BreakableWall extends the wall class since it adds extra functionality. The class differs from the Wall class because it has a health field as well as some methods to manage its health. A boolean variable is used to mark it as "dead" or "alive", so it can be cleaned up in main and removed from the game_objects array when appropriate.

- **Menu**: This class wasn't required, but I added it when I decided to implement the functionality of a menu within my game. This class is simply used to display the curved rectangles used as buttons on the main menu as well as the text used for options in the main menu.

- **MouseReader**: This class implements the MouseListener interface, so it has to also implement all of the methods specified within the MouseListener interface. I only cared if a mouse was pressed, so mousePressed(MouseEvent) was the only required method that I didn't leave blank. In this method, I use a static game_state variable from GameWorld and thus decide where to register clicks. The game_state variable is advanced/returned to the proper game_state when buttons are pressed with the mouse.

Self-Reflection

This term project was a very interesting experience for me. Initially, I seemed to hit a wall with collisions (no pun intended) when I was working on the collisions for the Tank Game. Eventually, I was able to understand many concepts which I hadn't really delved into before. I really enjoyed both of these projects even though they were pretty difficult for me. I approached the games in a piece by piece aspect where I tried to get something minimal working before adding to it, addressing glitches, and optimizing my code to better follow OOP principles. I am very satisfied with the fact that I have been able to fully complete both games while actually implementing the good OOP principles that we discussed within the course.

Conclusion

In this project, I designed and implemented a Tank Game implemented in Java where great attention was paid to ensure that good OOP principles and especially the Single Responsibility Principle were followed. I am very grateful for having the opportunity to complete these projects in this fashion as I feel that it has greatly enhanced my understanding of software development.