

CSC 413 Summer 2020
Term Project Documentation
Asu Paul

Repo links

<https://github.com/csc413-02-SU2020/csc413-tankgame-pasu19>

Table of Contents

Project Information	3
Introduction	3
Scope of Work	3
Background / Game Rule Explanations	4
a) Tank Game.....	4
Development Environment.....	9
a) Version of Java Used.....	9
b) IDE Used.....	9
c) Note on resources... ..	9
d) How to build and import the games.....	10
a) Building the JAR.....	22
b) Commands used to run the JAR.....	22
Assumptions Made.....	22
Tank Game Class Diagram.....	23
Class Description of Shared Classes.....	25
Class Description of Tank Game Specific Classes.....	27
Self Reflection.....	30
Conclusion.....	31

Project Information

This section will serve as a general overview of the project while the introduction section will delve into some more technical details of the project. The term project describes within this report consists of two different games: a tank game, and slightly modified version of Pyramid Panic. I specifically chose Pyramid Panic as the second game because I wanted to challenge myself in order to see what I am capable of. This report will reflect on my struggles and how I managed to complete both games while applying good OOP principles.

Introduction

Both the tank game and my modified version of Pyramid Panic were implemented in Java. My primary focus during the project was to try and minimize coupling while increasing the level of cohesion within my code. I feel that these two principles in conjunction with the other principles that we covered during the course such as the SRP (Single Responsibility Principle) were vital to my success in these projects.

Both the tank game and my modified version of Pyramid Panic were rather large games with various resources, features, and functionalities. This was initially very hard to tackle as I did not want to introduce (coupling) heavy dependency between my classes. My approach was a little unconventional in the sense that most things were implemented in a methodical fashion, but that my overarching goal was to get a working base product first. It is important to note that I kept the principles of OOP in mind, but I didn't allow them to hinder my progress in the early phase of these projects. This is where the class diagrams come into play.

Class diagrams were instrumental because while I did not follow them exactly; they allowed me to garner a general overview of what would be a considered a reasonable approach. Although class diagrams were required for this project: 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.

Scope of Work

The primary scope of my work was to get the project's working with decent OOP principles and a good structure that allows for ease of modification, and optimization in the future. Coming from the point of no background design: I try to make a design that follows the fundamental principles and get it operational before I go in and make optimizations to get it up to par. I strongly followed this methodology within my implementation of this project.

Background / Game rule explanations

The section is extremely important as it will describe how the games are to be played, and what rules each game has.

a) Tank game

Upon execution of the tank game jar, you will see a window with a menu like this:



Figure (1): This figure shows the start/main menu screen of the Tank Wars game.

Players can press with their mouse "BEGIN!" to start playing the game or they can either select "HELP" or "EXIT" in the same fashion. How this menu was implemented, and how mouse input was read will be discussed in detail later within this report in the Mouse-Reader and the Menu class sections, respectively.

We will first go to the “HELP” screen to better explain the purpose of the game along with its controls. Clicking the “HELP” button shows the user the following screen:

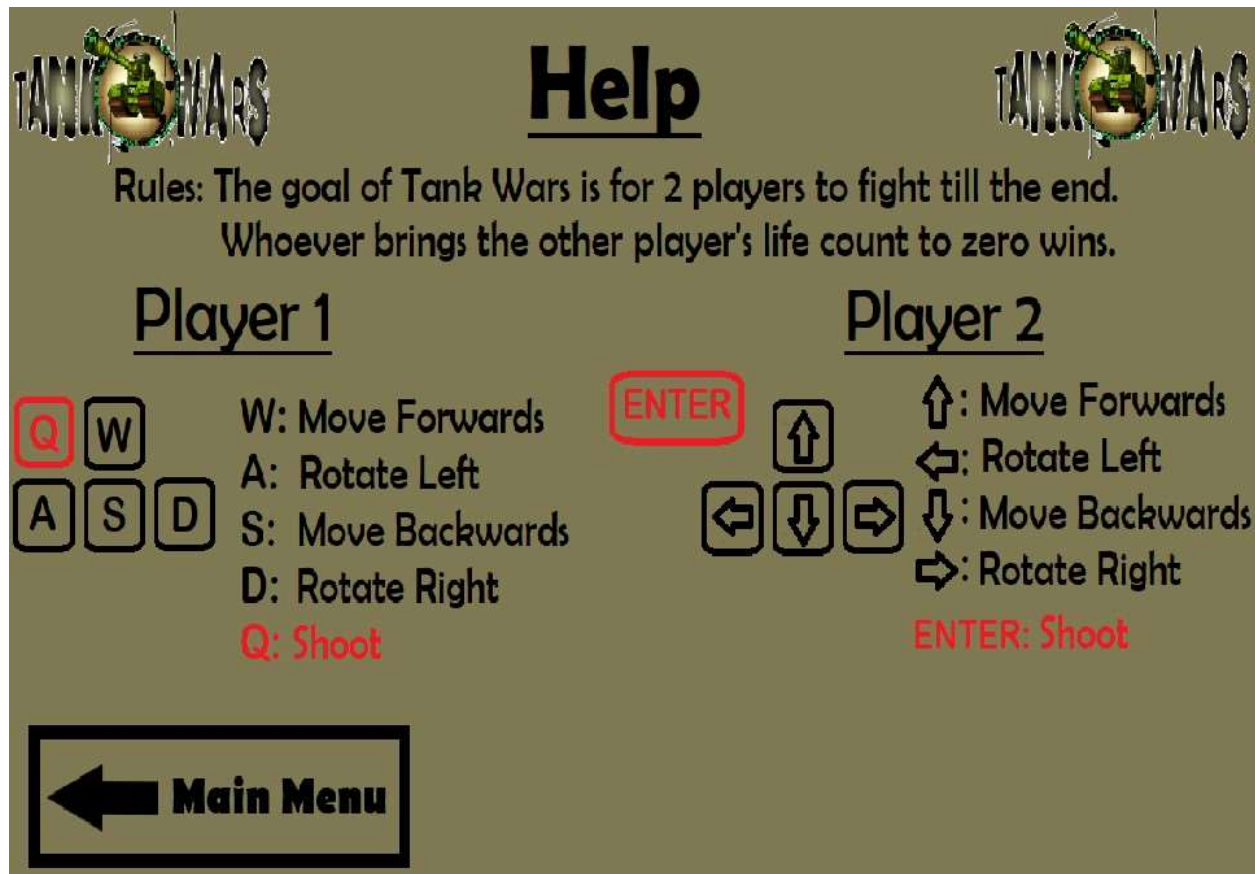
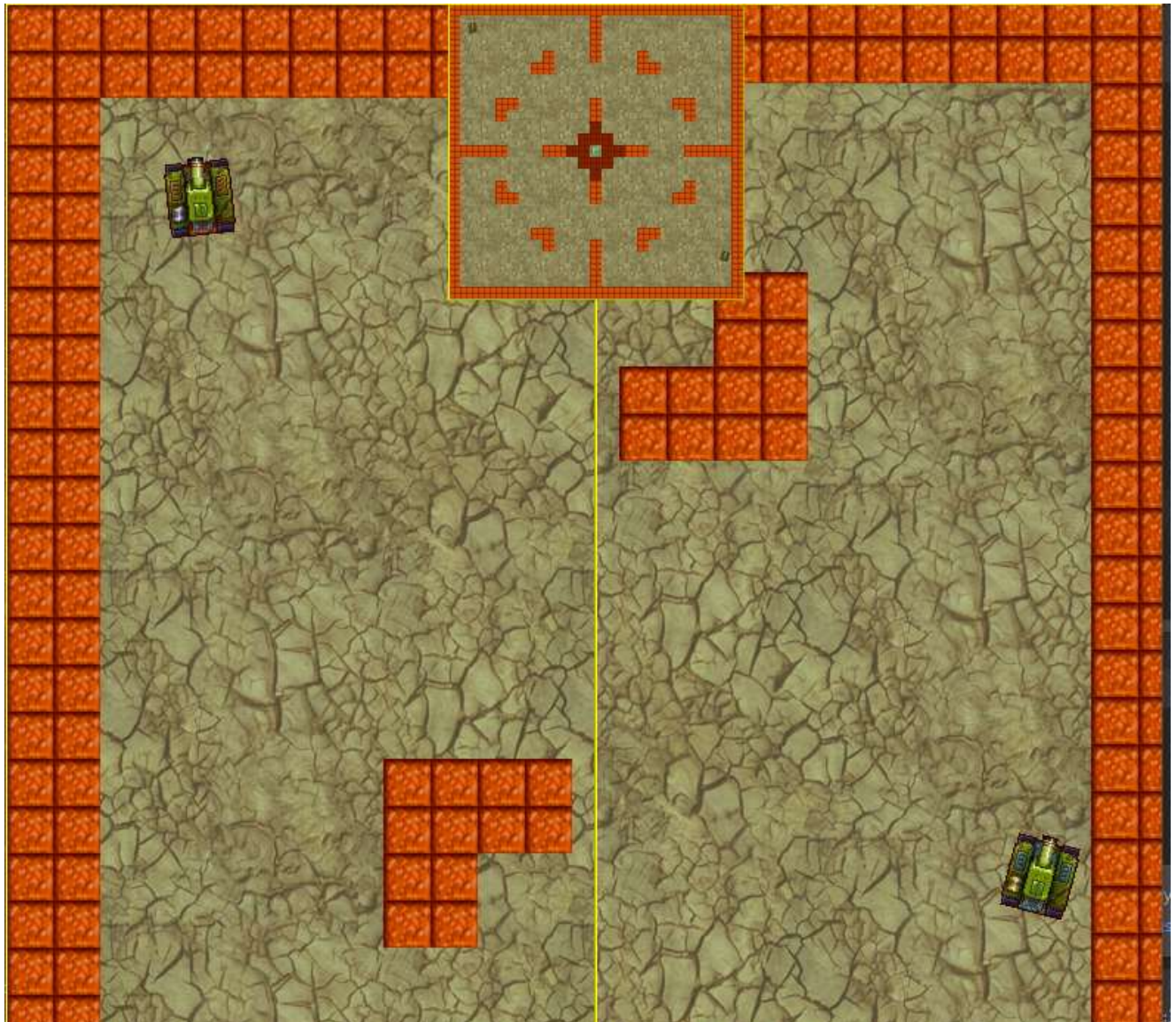


Figure (2): This figure shows the “Help” screen of the tank game.

On the “Screen”, we can see the Rules of the game which are to fight until one of the players live count is drained to zero. It is important to mention that PowerUps (health boost, speed boost) can also be picked up in-order to make the game more fun. The controls for each player are displayed in an easy to understand fashion within the help screen. There is also a button on the bottom left of the screen which the user can press to return to the main menu.

We will return to the main menu and press “BEGIN!” to illustrate what the interface of our game looks like:



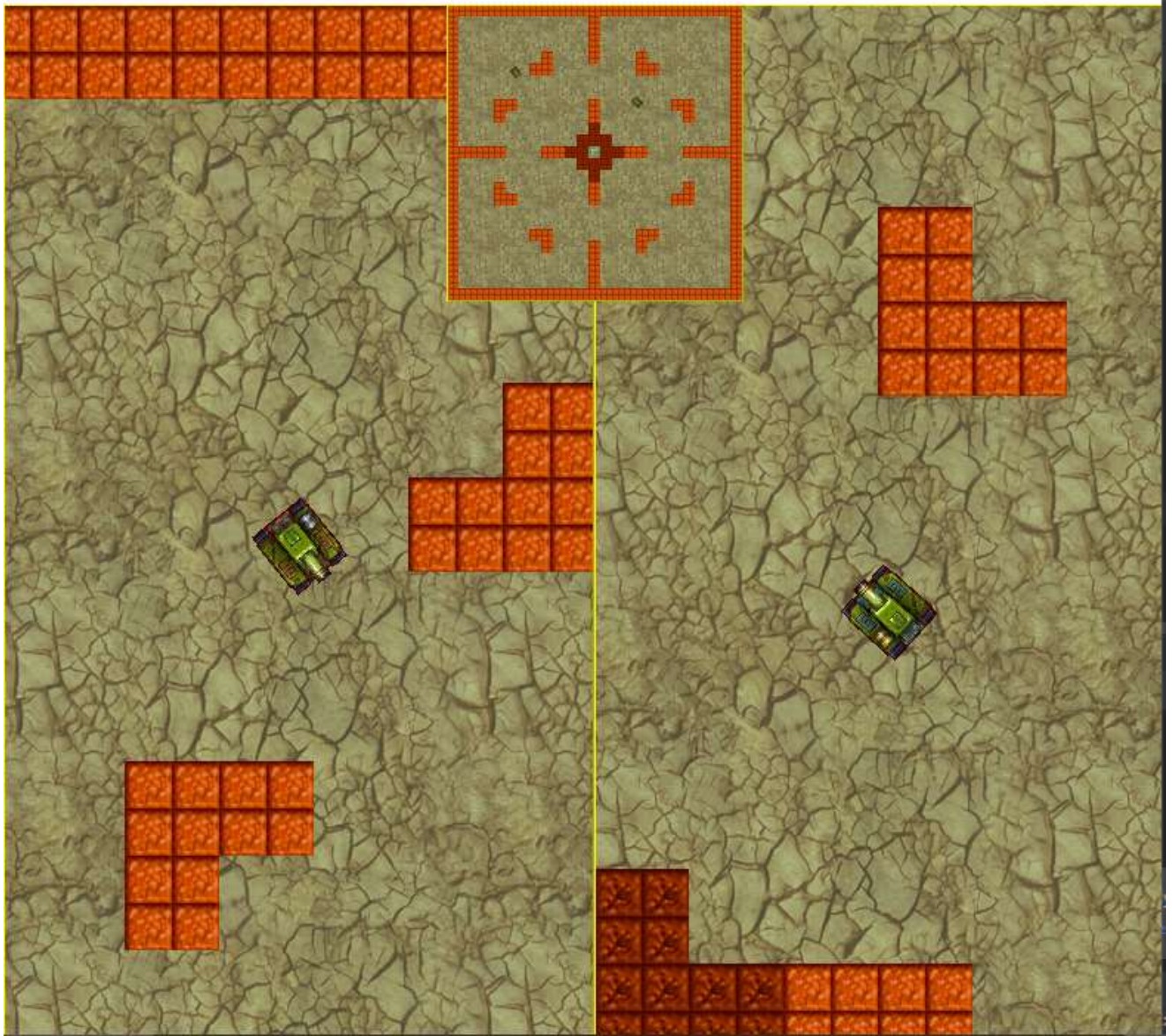


Figure (3): This figure shows what players would see as soon as they start playing by pressing the “BEGIN!” button. There are two split screens, one for each player as well as corresponding live counters and health bars. There is a mini- map which has strategically been placed directly in the center, so players don’t have unfair difficulty levels with respect to viewing the overall map. The dark red walls around the perimeter of the map are border wall and cannot be destroyed while the

lighter colored walls are breakable meaning that they disappear once shot enough times (2 shots).

Once a player loses, a win screen such as this one is shown to reflect that:

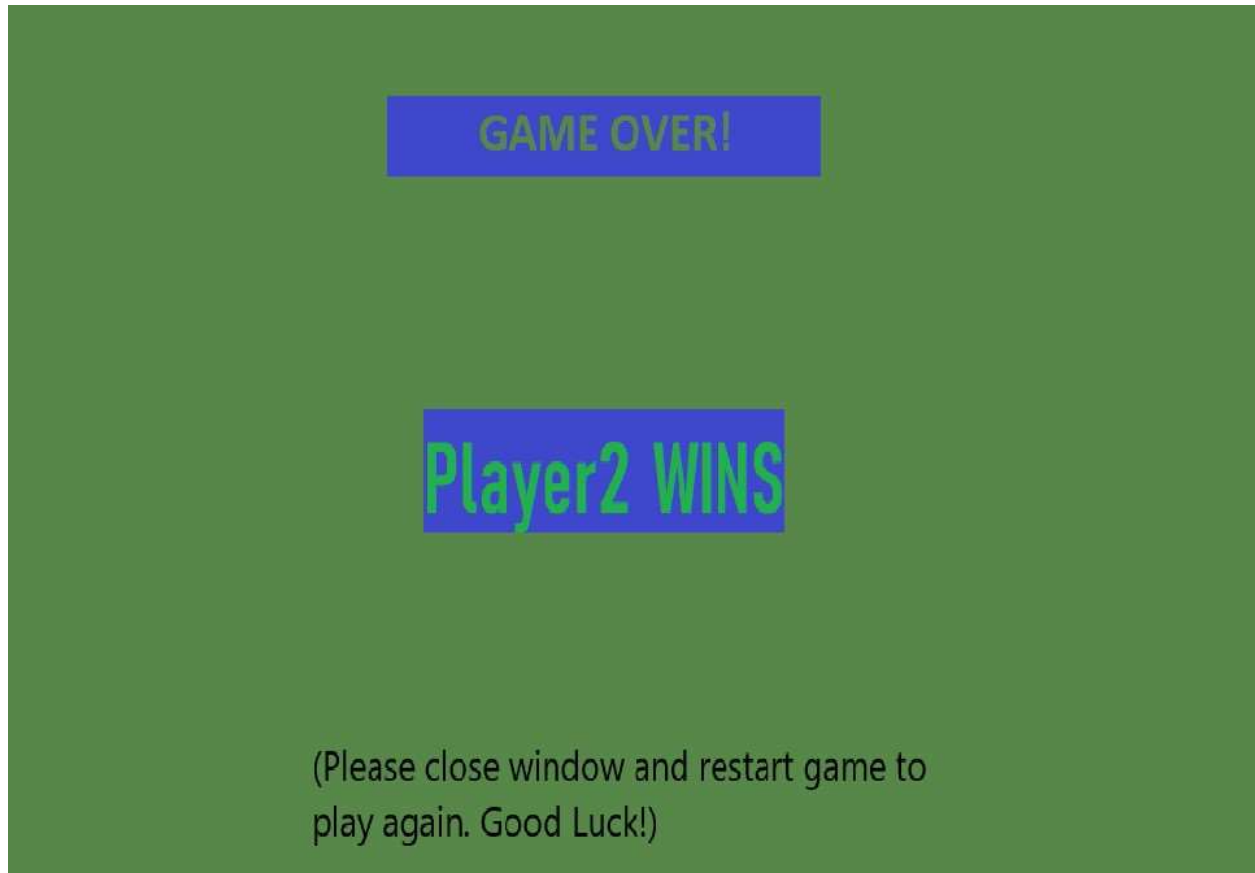


Figure (4): This figure shows the Player2 victory screen which is shown once they beat Player1 by draining their live count to zero.

Development Environment

- a) Version of Java used: Java 12.0.2
- b) IDE used: IntelliJ Idea
- c) Note on resources: most of the images used in this project were provided. The only two external images were the health boost and the speed boost in the

Tank Game which I based off icons from “Game-icons.net”. Regarding code resources, I used and modified the given Tank Rotation Example given via ilearn. All rights and licensing belong to their respective owners. All the code used in these two projects was purely my own except for the base code given in the Tank Rotation Example.

How to build and Import the games

The easiest way to run my game is simply to double click the JAR file on a machine with the appropriate (or later) Java version. However, I show all the steps required to import and run the game within the IntelliJ Idea IDE. NOTE: I will only demonstrate how to import and build the second game as the process is identical for the Tank Game.

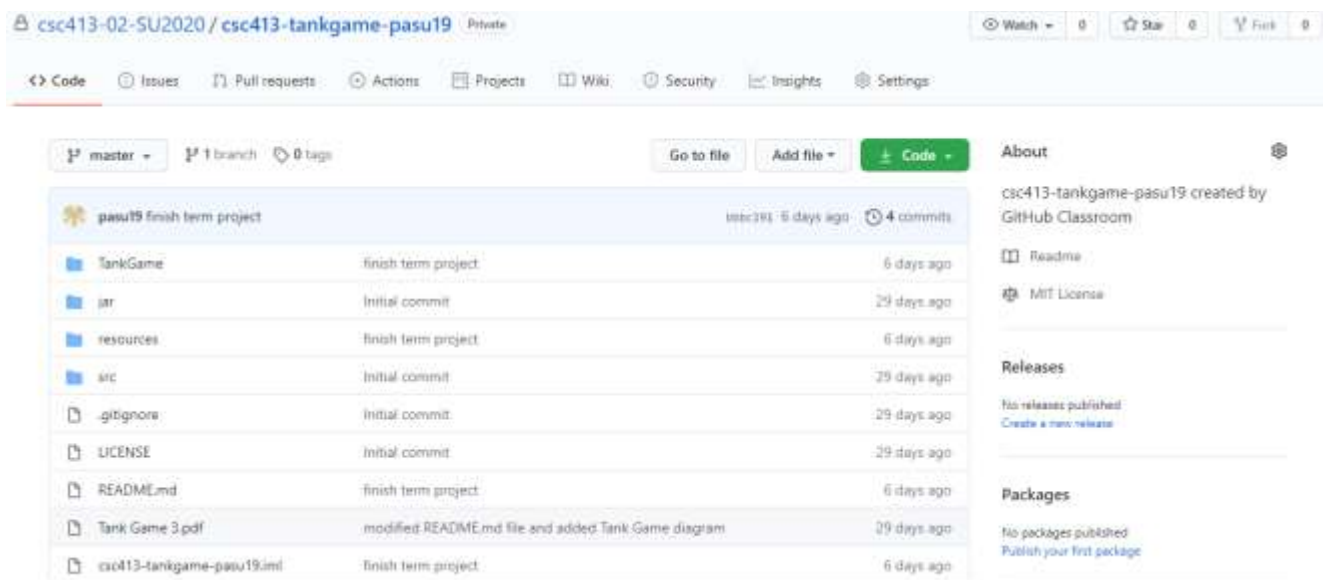
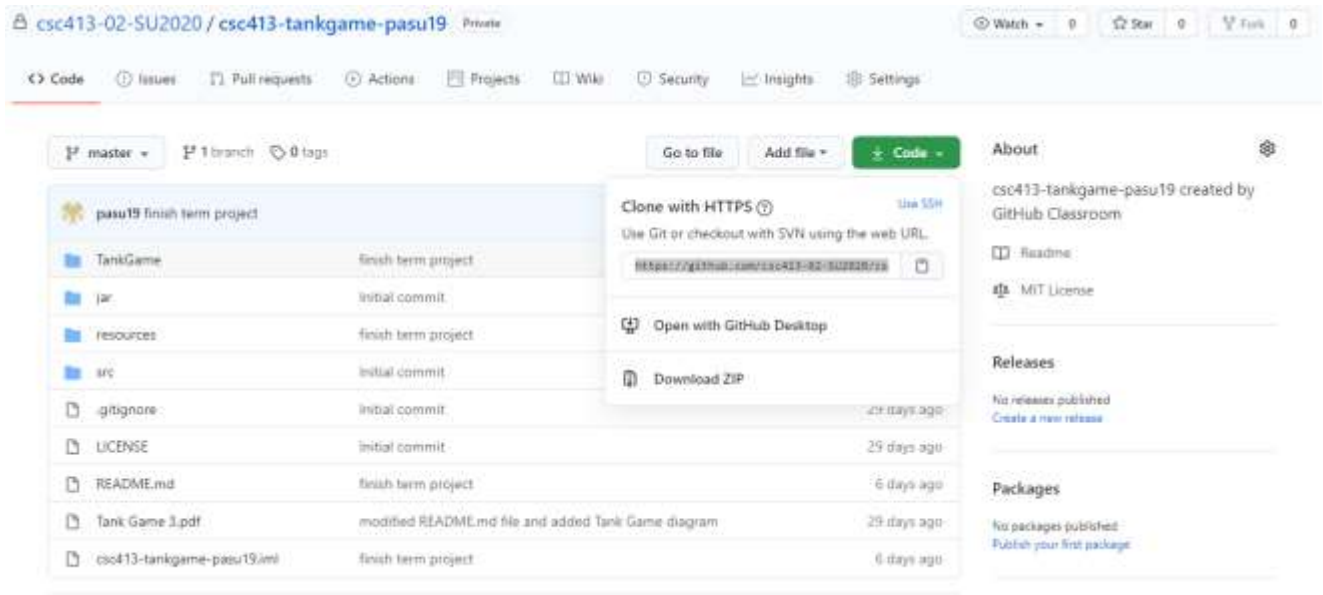


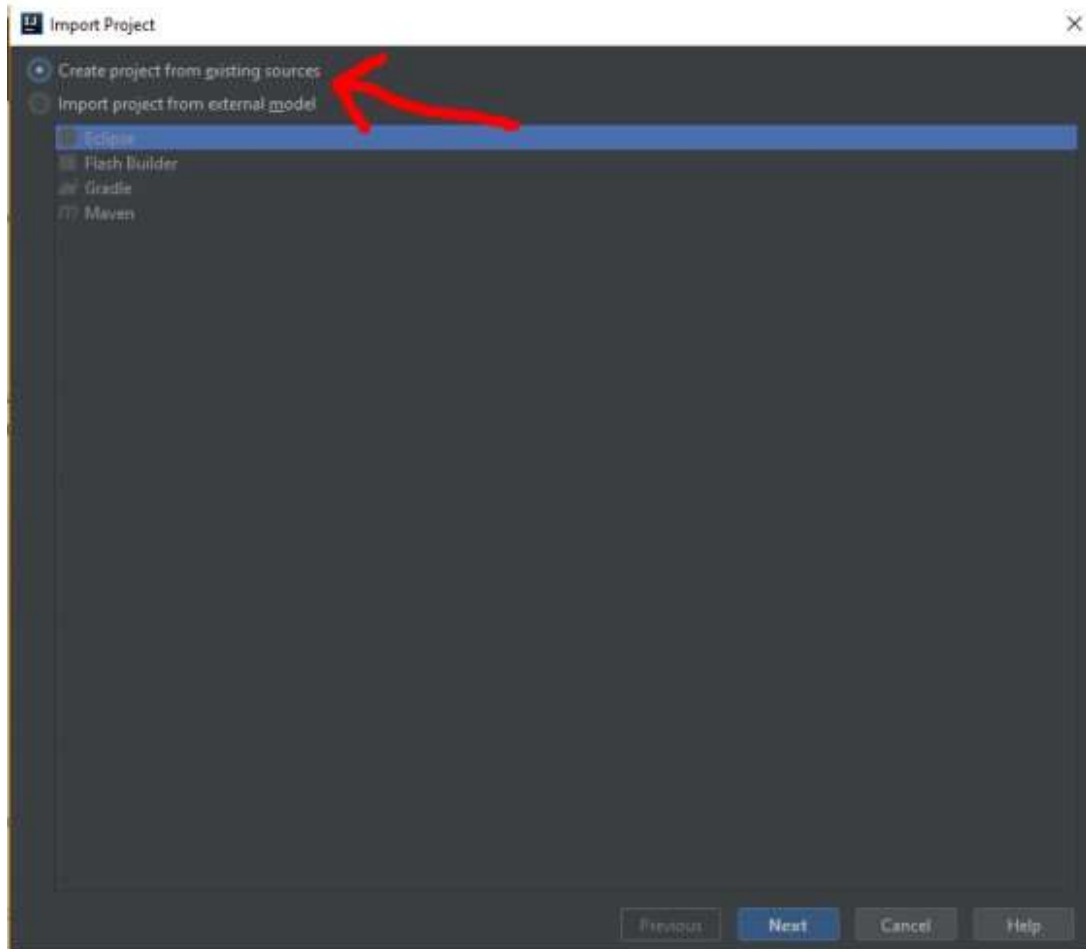
Figure (7a): Go to the repo link and click the green “Clone or download” button.



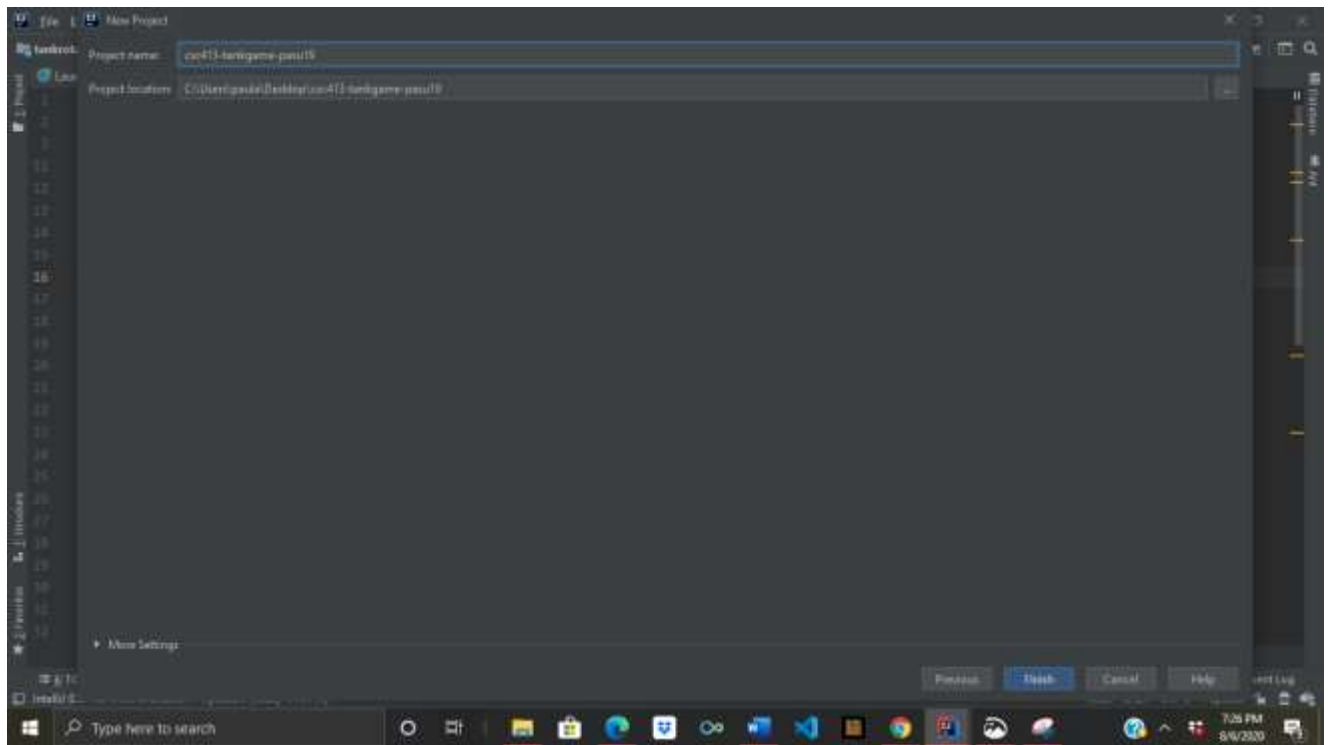
Figure(7b): Click “Download ZIP” and you will receive a zip folder in your downloads location.
Then extract the zip to a known location

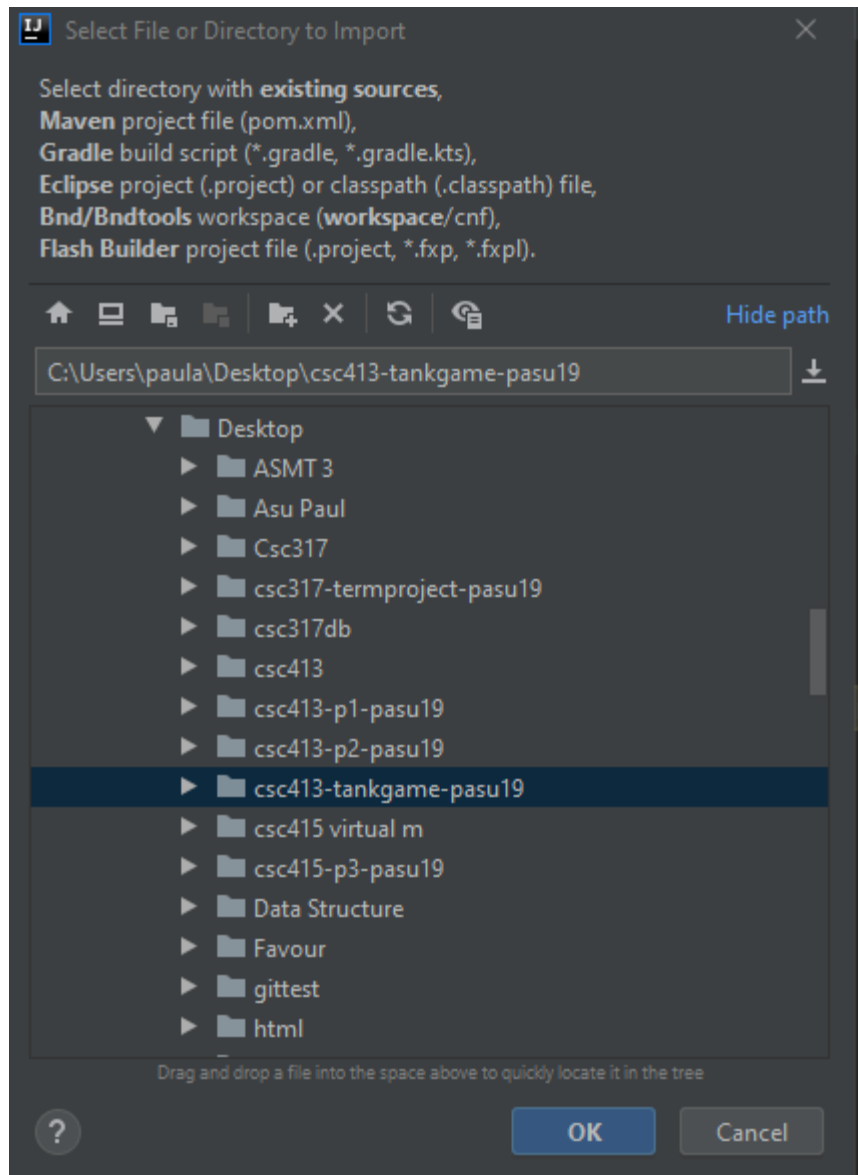


Figure(7c): Launch the IntelliJ Idea IDE and click “Import Project”. On the next screen, select where you extracted the zip file and press “OK”.



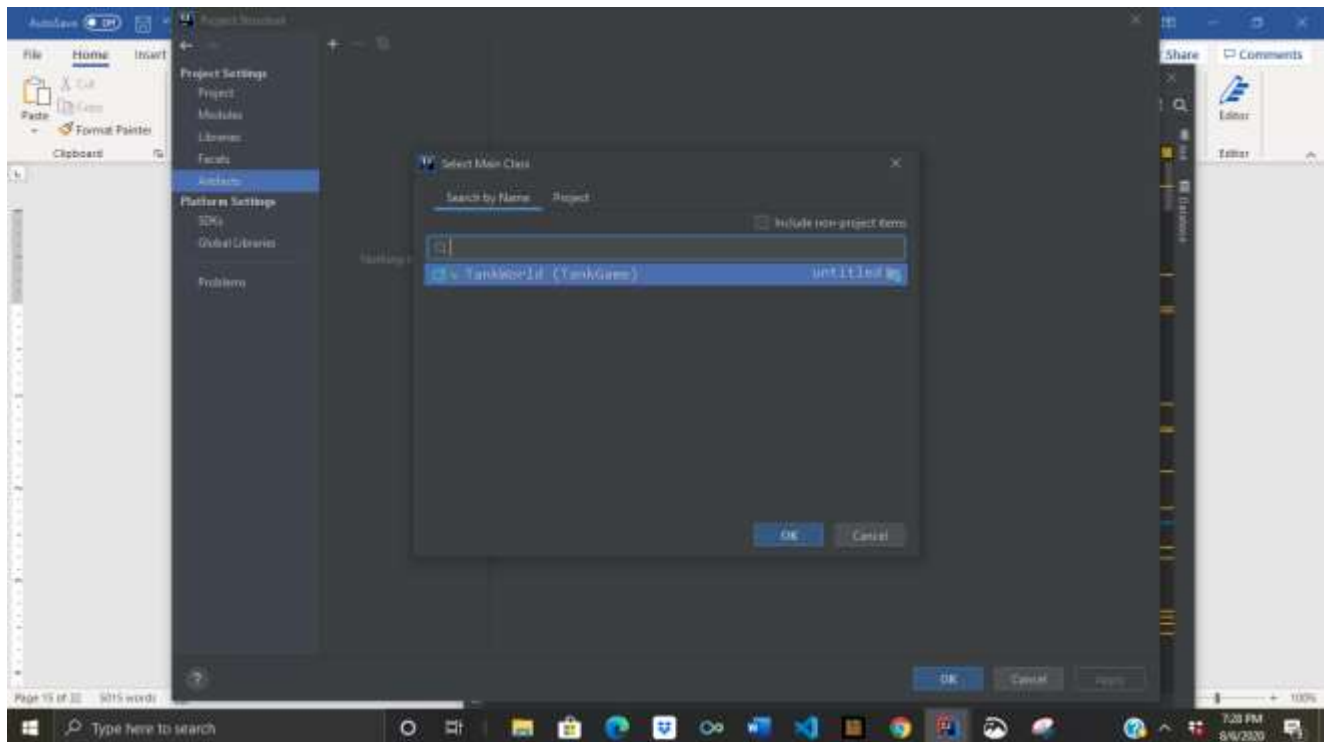
Figure(7d): Click “Create project from existing sources and press “Next”.



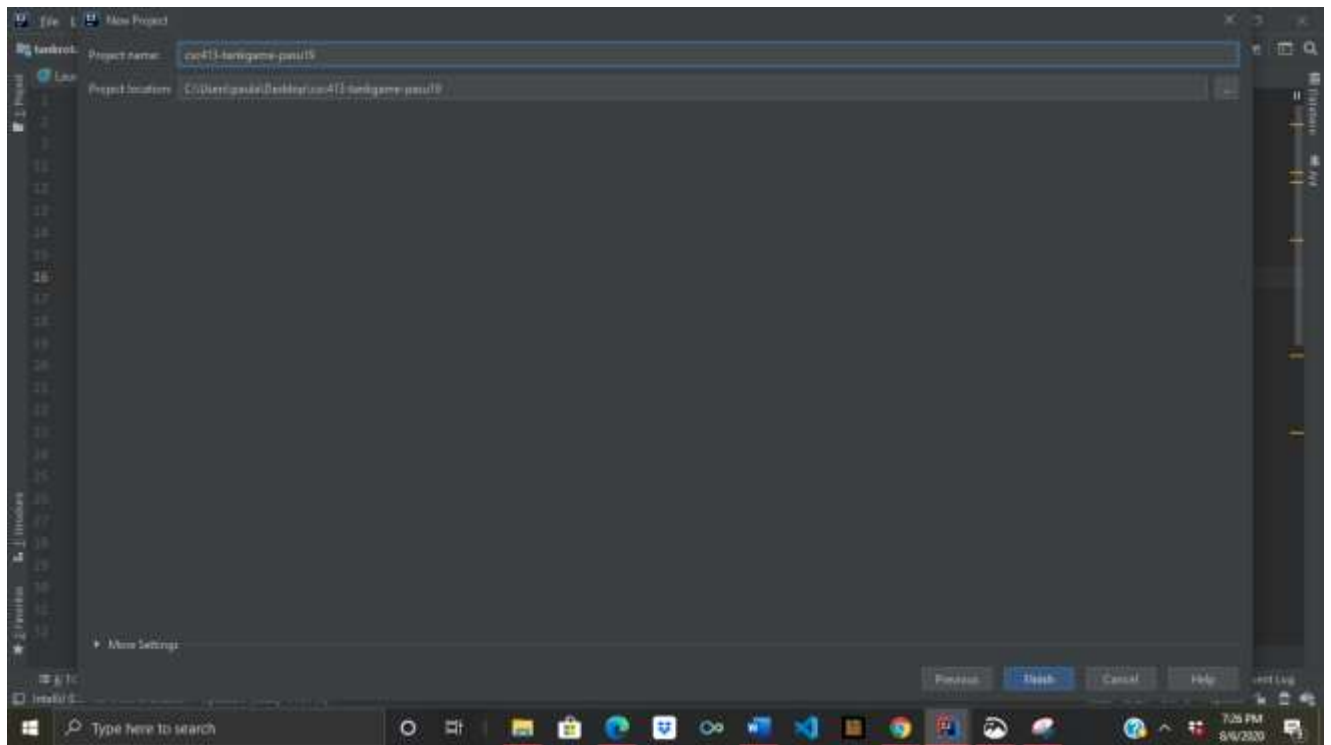


Figure(7e): Verify that the Project format matches the figure and press “Next”.

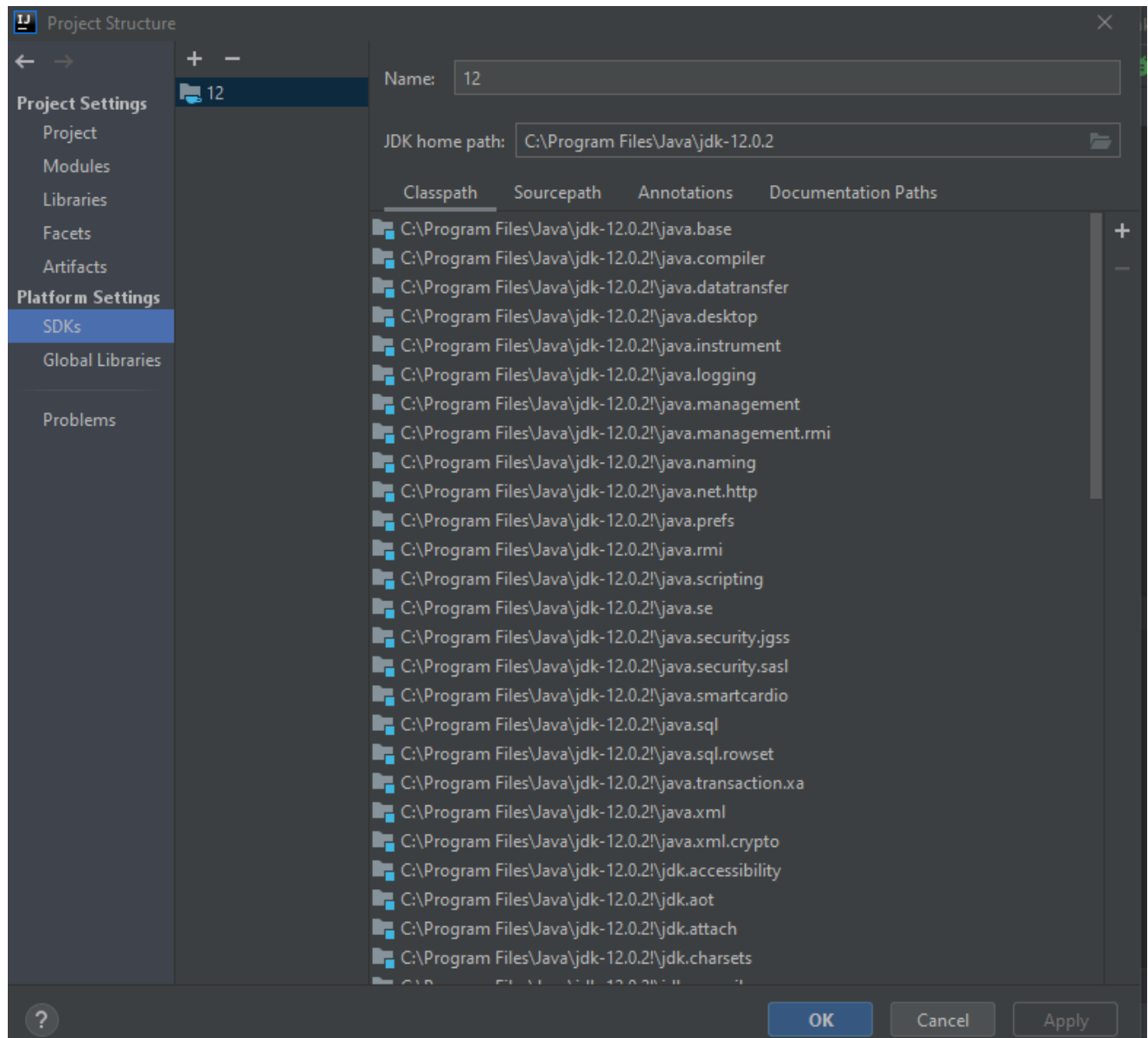
Figure(7f): Click “Next”.

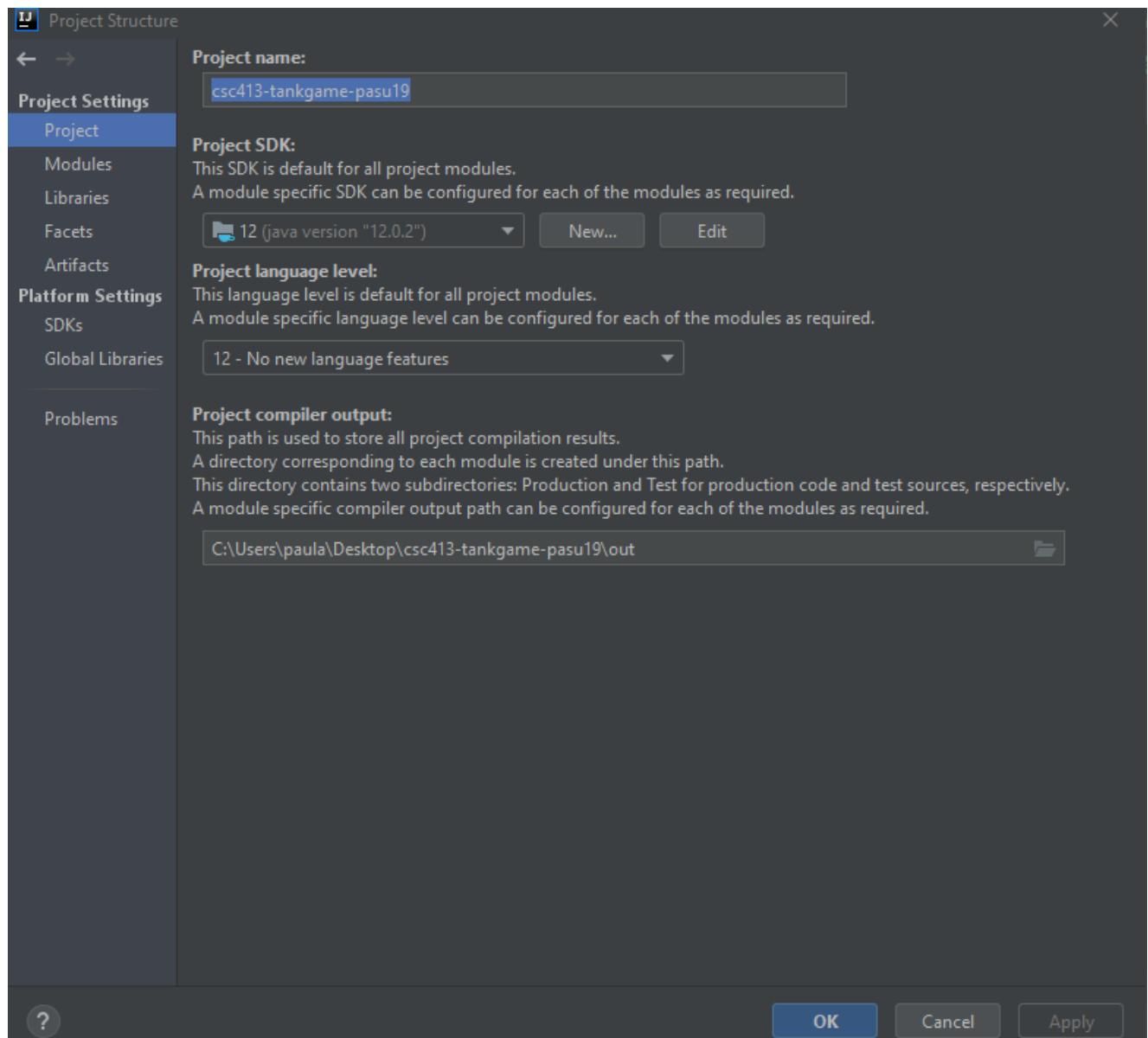


Figure(7g): Click “Next”.

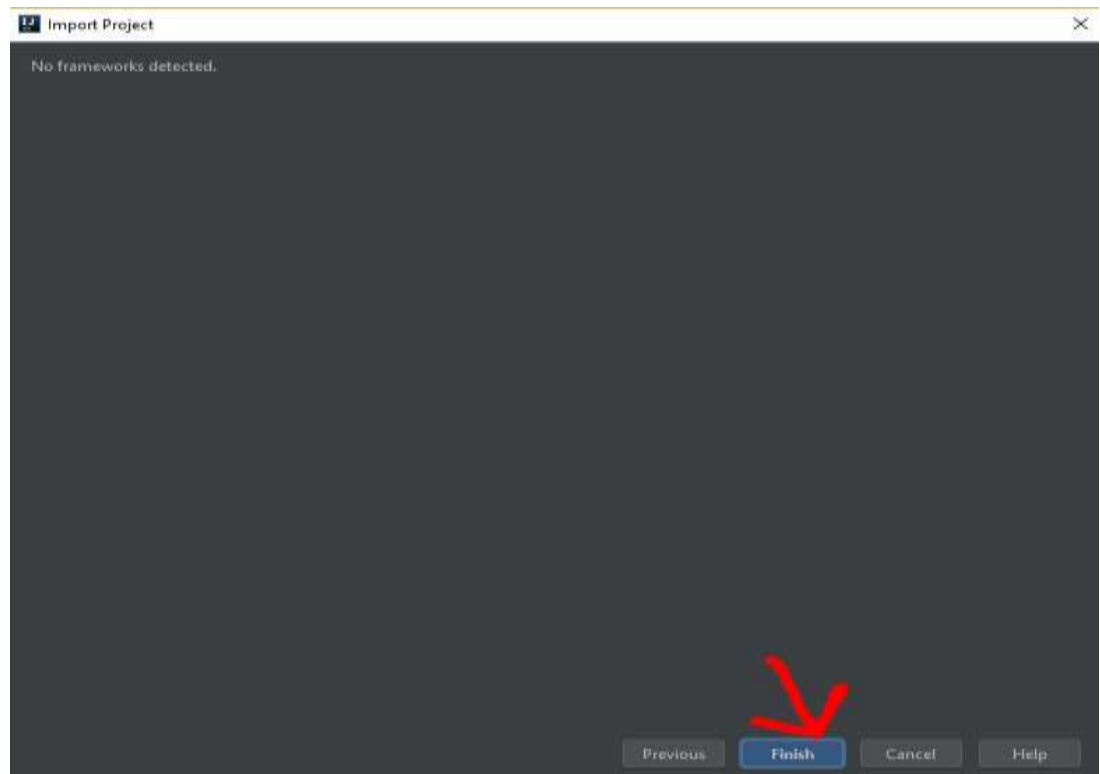


Figure(7h): Click “Next”.

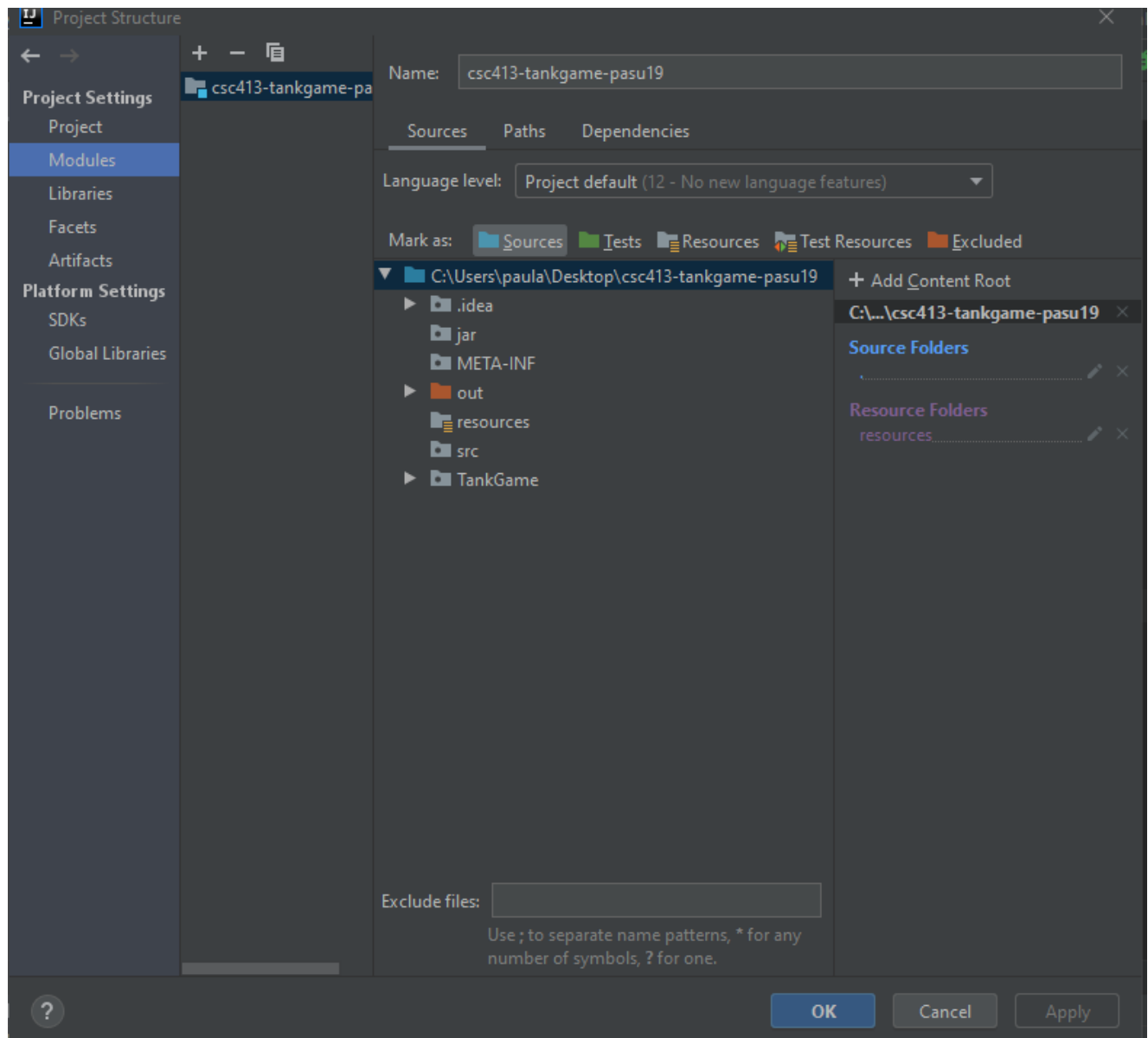




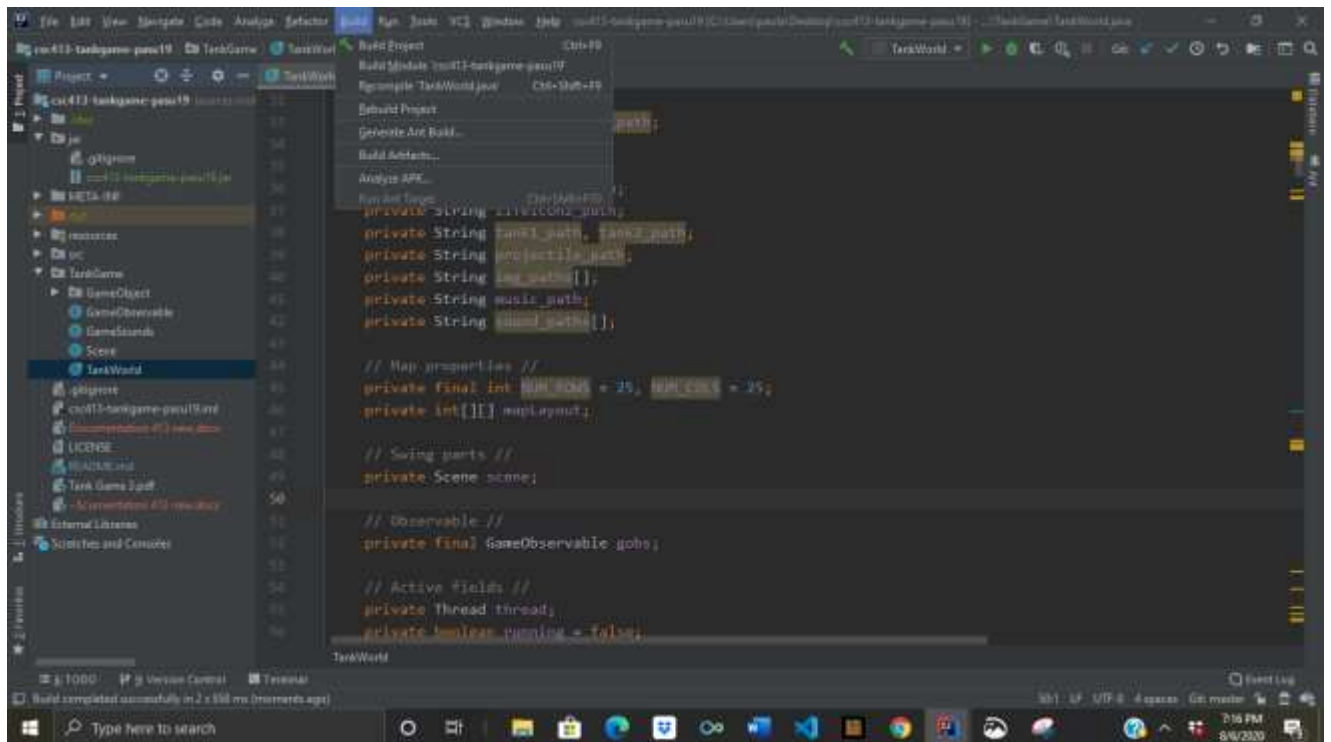
Figure(7h): Click “Next”.



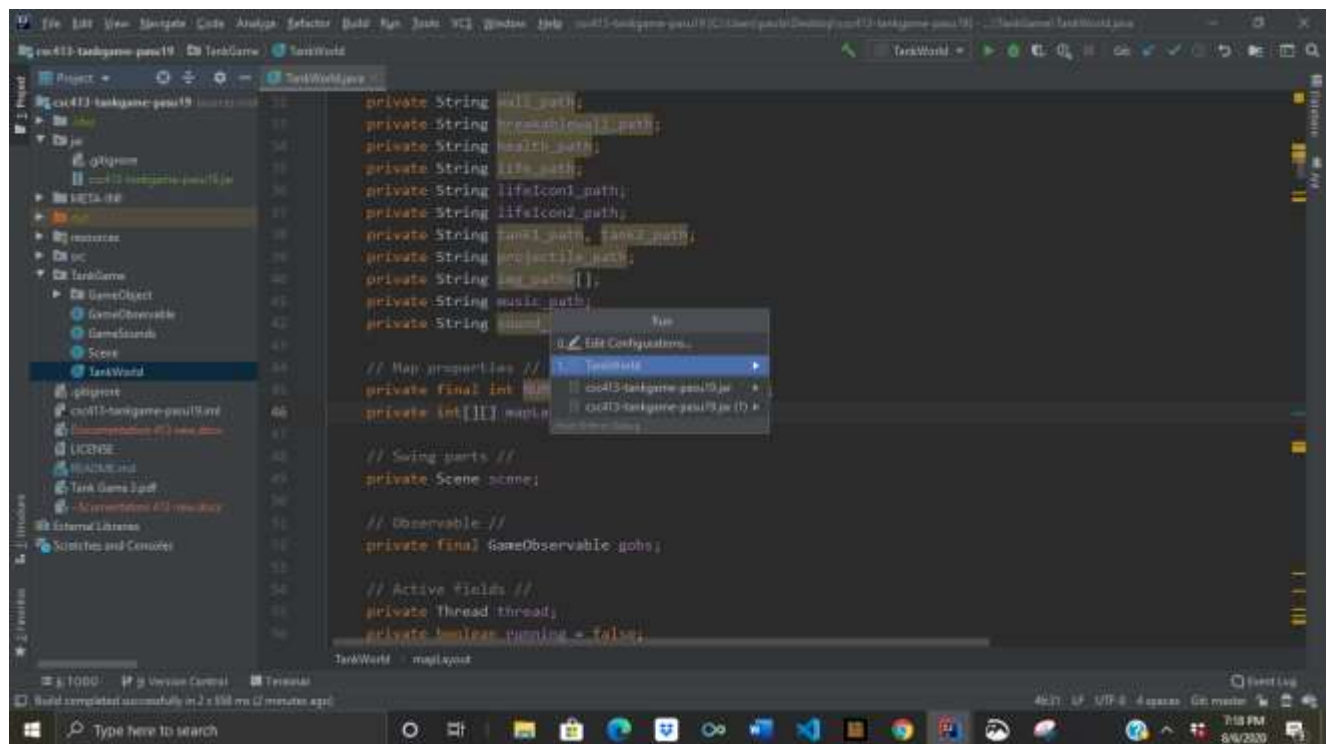
Figure(7i): Click “Next”.



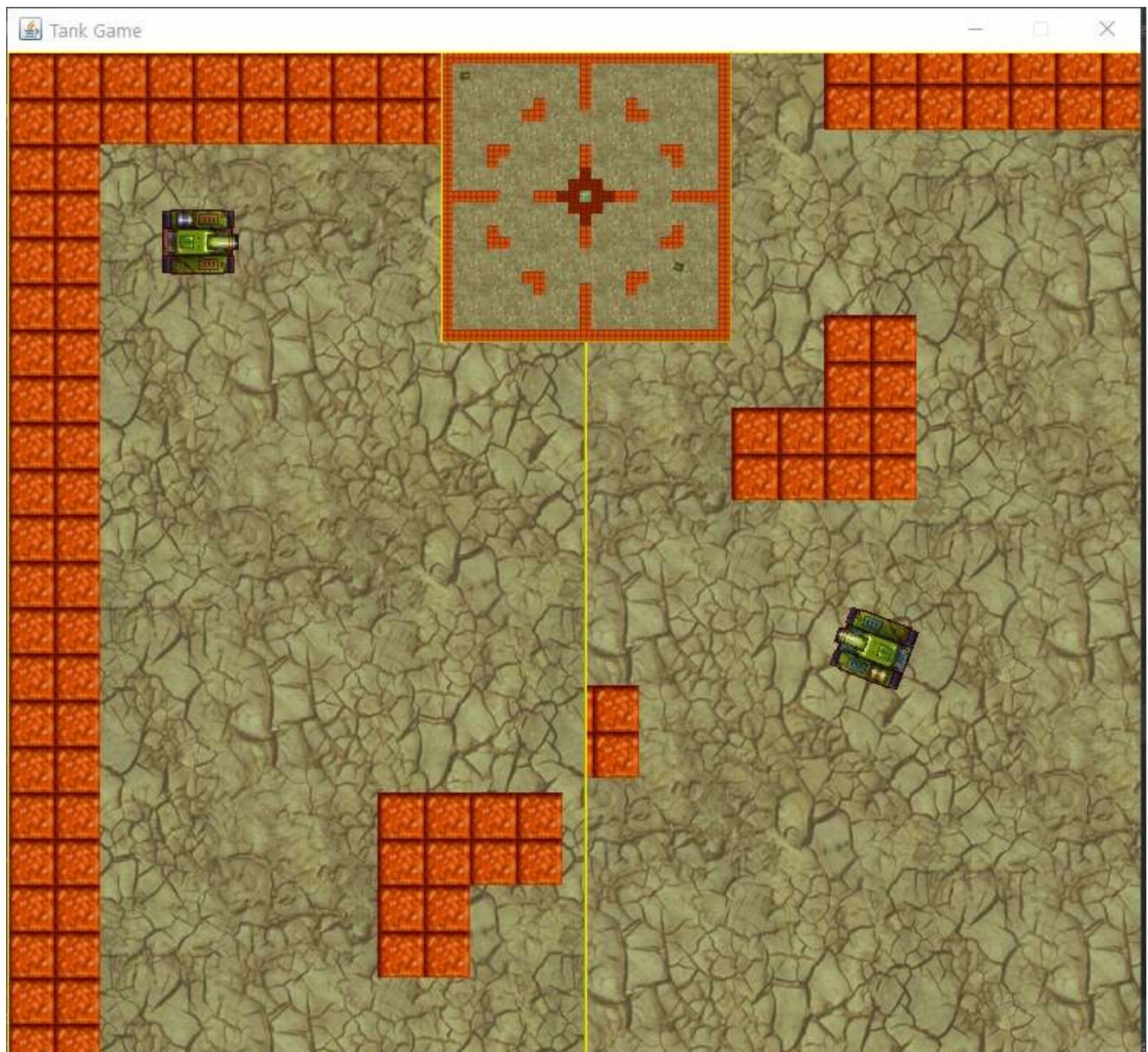
Figure(7j): Go to the project structure window(ctrl-alt-shift-s) and make sure that the src folder is selected as a source and that the src /resources folder is selected as a resource folder as shown in the figure.



Figure(7k): Click “Build” and then “Build Project” to build the project.



Figure(71): Right click on “GameWorld” and click “Run ‘GameWorld.main()’”.



Figure(7m): We can see that the game runs successfully.

a) 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.

b) 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 did have make very many assumptions during the design and implementation phase of this project. I did make a slight assumption in my design of the Second Game: my collisions are depending on the order that Game Objects are added. For example, I always add 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

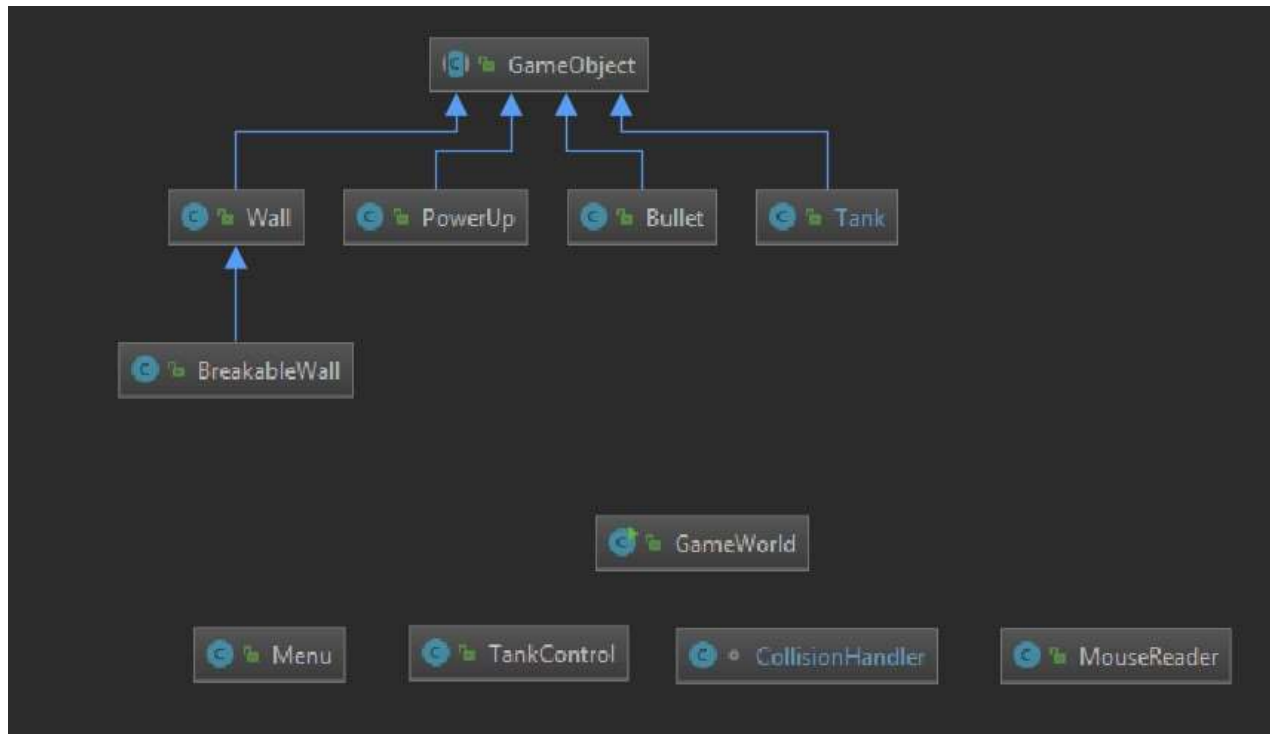


Figure (8): This figure shows the Tank Game class diagram. The core abstraction of the class is in the abstract `GameObject` class which is extended by various other classes (`Wall`, `PowerUp`, `Bullet`, `Tank`, and `BreakableWall`(indirectly)). The `GameObject` class provides base functionality such 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 does not 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 do not remain "clickable" after a user's choice has been completed. The `GameWorld` class controls the status of the game. These classes will be described in detail within the future sections of this documentation.

Second Game Class Diagram

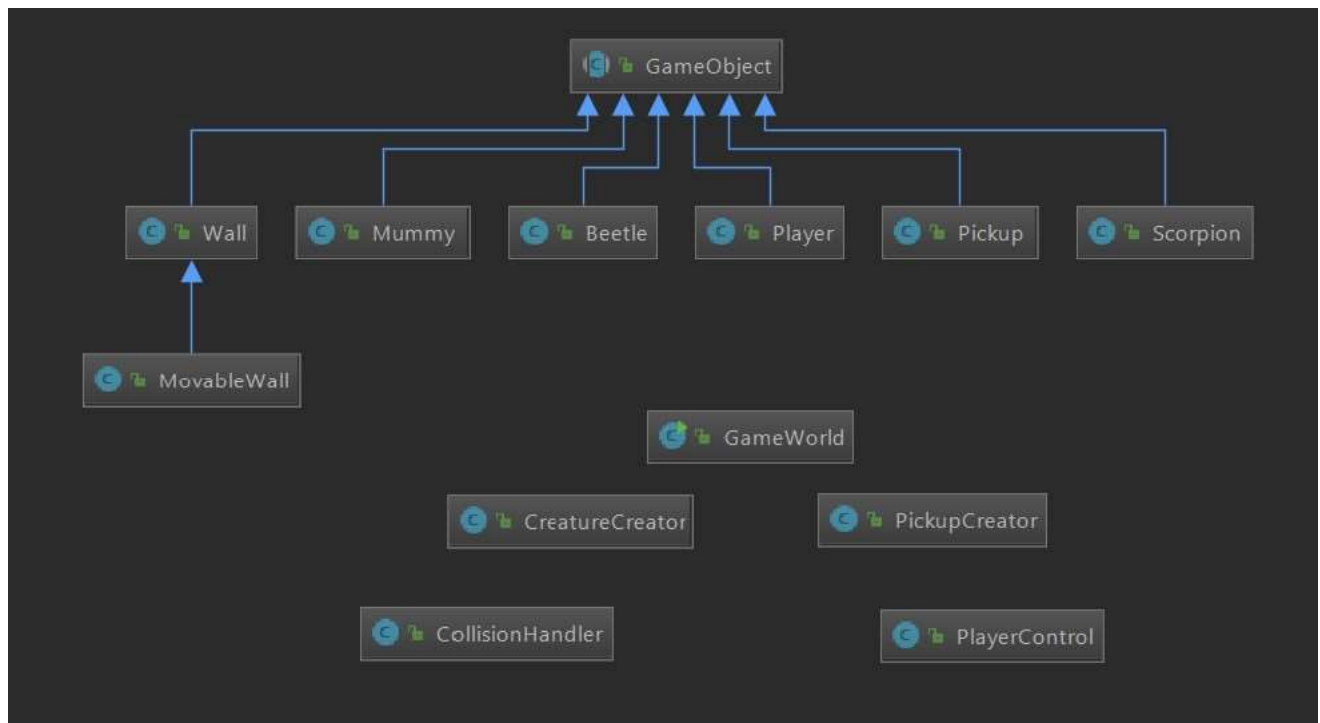


Figure (9): This figure shows the second game class diagram. My goal with this project was to utilize abstraction whenever possible. I started off with an abstract class **GameObject** which allowed me to group the functionality which is common between various object classes. Since **GameObject** is an abstract class, all its concrete child classes must provide their own implementation for all of its abstract methods. I have six first degree child classes of **GameObject**: **Wall**, **Mummy**, **Beetle**, **Player**, **Pickup**, and **Scorpion**. These classes are all similar enough to be considered a **GameObject** but each of them offers unique functionality to carry out their respective needs. The **GameObject** class has one second degree child class: **MovableWall** which has the same functionality as a regular wall, but with added functionality and a different set of additional methods. I feel that I have done a fairly good job at following the SRP (Single Responsibility Principle) by grouping the classes in this fashion. The last few classes are **GameWorld**, **CreatureCreator**, **PickupCreator**, **CollisionHandler**, and **PlayerControl**. I initially had the functionality of creating creatures, creating **PickUps**, and handling collisions all within the **GameWorld** class. However, I

eventually realized that I was giving the GameWorld class far too many responsibilities, and that I should break those up into their own classes to promote better OOP principles.

Class Descriptions of Shared Classes

- **GameWorld:** This class is the entry point of our program and handles creation along with maintenance of the game. It has some slight variations between the Tank Game and the second game, but its core functionality is identical. For example, a menu was implemented in the Tank Game, so an enumeration variable was used to store and monitor the current state of the game. This is not done for the second game because the second game doesn't include a menu, but such functionality would still fall under the general umbrella of monitoring and updating the game as a whole.

The GameWorld class has a plethora of variables, so I will not 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 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 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 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 problem, 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.

- **PlayerControl(TankControl in Tank Game):** This class has private final ints to hold each necessary button from the keyboard. The class implements the `KeyListener` interface, so it has to implement the methods within that interface (`keyTyped`, `keyPressed`, and `keyReleased`) which all take in a `KeyEvent` object. The class holds a data field of type `Tank` (for the tank game) and a data field of type `Player` (for the second game). Appropriate methods for these data fields are called to set Booleans that they have been pressed or released.
- **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 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.

In the second game, the Wall class has 3 Boolean variables (`is background`, `is border`, `is sword`) to denote which image should be shown if `drawImage` is invoked on that object.

NOTE: I stored 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.

Class Descriptions of Tank Game specific classes

- **Tank:** This class is an 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, vx, vy, 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 `isActive` 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: `bullet_img`, `big_explosion_img`, `explosion_img`. The `bullet_img` holds an image of the bullet as the name suggests while the `explosion_img` is used to display the image of an explosion (appears when a BreakableWall is shot). The `big_explosion` 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.

Class Descriptions of Second Game specific classes

- **Player:** The player class in the second game is analogous to the Tank class in the Tank Game. This class controls and maintains various aspects relating to the player. It holds 4 arrays of BufferedImages(down, up, left, right) as well as 4 booleans(reset up_img, reset_down_img, reset_left_img, and reset_right_img). It also holds another 4 integer variables to denote which image is currently being displayed. This allows us to loop through these images for a given time (250 ms). We also have a current_img variable to denote which image is currently being showed. This allows us to determine if the player is currently going up, left, down, or right. Alternatively, I could have used a set of boolean variables to determine this.

The class has methods to control movements in all 4 different directions as well as methods to get and set the score, whether the player is alive, the number of scarabs, and if the sword is active.

There is also a getOffsetBounds() method which I used to shrink the width and height of the player's rectangle by four. The rationale behind this is that I am using this method in my CollisionHandler for player to wall collisions, so my player can easily walk through tight areas. In a similar fashion, I store prev_x and prev_y variables to track the player's previous coordinates. Upon hitting a collision, I simply restore the players x and y to their coordinates before the collision. There is a very specific reason for me to do this. In the Tank Game, I had a do not move variable to handle collisions which worked fine for that game but caused issues with this game. Consider the case where a player goes straight up against a wall and continues pressing up for some time. The player can go back without any problems, but they cannot turn left or right as they have practically become wedged against the wall. After struggling with how to address this, I finally settled on the simpler approach of restoring the coordinates which eliminates this issue completely.

- **MovableWall:** This class extends the Wall class and adds moving functionality to it. The way it does this is by using two methods(incrementVx(int), incrementVy(int)). IncrementVx(int) is used for horizontal walls where the player's x velocity is passed in to give the appearance of them pushing the wall. IncrementVy(int) is used for vertical walls where the player's y velocity is passed in to give the appearance of them pushing the wall. The player's appropriate velocities are only passed in when they collide with the wall in the appropriate place (i.e. top and bottom for vertical movable walls).

Self-Reflection

This term project was a very interesting experience for me. Initially, I seemed to hit a wall with collisions (not funny at all) when I was working on the collisions for the Tank Game. Eventually, I was able to understand many concepts which I had not really delved into before. Coming from

engineering background, I had not worked on projects of this scale (at least in Java). I really enjoyed both projects even though they were 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 implementing the good OOP principles that we discussed within the course.

Conclusion

In this project, I designed and implemented a Tank Game. This game was 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.