ZTD47 Design Manual

Eben Blaisdell, Jack Hall-Tipping, Matt O'Donnell, Xeniya Tsoktoyeva

The key to our success as a team was our plan that we developed before beginning the project. Before writing any code, we wanted to begin with very simple, general classes, and build off of them. We wanted to develop the system to best follow Object Oriented Design principles we have learned thus far. We believe we achieved this, and in general, our system can best be described as following Object Oriented Design methodology. We were improving the internal structure of our code by refactoring it, and this practice allowed us to add new features without facing any consistency problems.

Before creating our user stories, we wanted to find out what practices more experienced agile developers tended to follow when writing and using them. We learned that, instead of simply listing every task, which can get confusing, "user stories help keep teams focused on the end goal of 'why' a feature is needed" (https://sprint.ly/blog/agile-user-stories/). Below are the following user stories that we completed for this project:

-As a player, I want a world to play in, so that I can play a game.
-As a player, I want a general menu, so that I can better interact with the game.
-As a player, I want to be able to move, so that I can run from aliens.
-As an alien, I want to be able to move, so that I can kill the player.
-As an alien, I want to be able to kill the player, so that no humans exist.
-As a player, I want to be able to place towers, so that I can fight aliens.
-As a player, I want my towers to shoot, so they can kill aliens.
-As a player, I want to be able to shoot, so I can kill aliens.
-As a player, I want different waves, so the game gets more challenging.
-As a player, I want to be able to see how much money I have, so that I can buy new towers.
-As a player, I want more of an introduction and ending to the game, so that I know what I'm doing on this world and why.
-As a player, I want the graphics and audio to be better, so that the game feels more real.

Our user stories were created in order to best follow a Minimum Viable Product approach to software development. We felt this was the best approach to take because if we experienced some unforeseen problem, we would have some sort of full-stack playable game, instead of just for example, the backend, if that was all we implemented that far. Agile development was key in the creation of our user stories. We also noticed that certain user stories took significantly longer than others. For example, the first user story: *As a player, I want a world to play in, so that I can play a game,* took significantly less time to complete than the user story: *As a player, I want different waves, so the game gets more challenging.*

One part of the development that we really enjoyed was creating our own storyline for the *why* of the game. This meshed in nicely with our user stories in general. Our title screen tells

the story of a post-apocalyptic world, and we shifted away from solely aliens and moved onto zombies and robot soldiers as the combatants. We found that the user story process in general caused us to become invested into our story and game, and moreover, our work in general. We ended up having more fun than we anticipated, and being forced to create user stories from the beginning was probably a major contributor to this.

Most of our user stories were from the point of view of the player, as we saw this was typical of most we had seen before. We felt, however, that some user stories should be from the perspective of the zombies or aliens or robots instead. We did not feel that towers or menus should have their own user stories, because we felt that the user should own them. We came to the conclusion that there should be two characters in the user stories because it feels like there are only two types of characters in the game, the player and the combatant. We feel the system accomplishes our user stories because it follows good OOD principles and would be fairly easy for a fresh set of eyes to pick up. We ended up with a playable game with features that can be classified by our user stories, and we are proud of our project.

Our design is based on the standard Model, View, Controller design pattern. Public class files are arranged into model, view, or ctrl folders based upon closest association.

The classes that determine the game's logic are in the model folder. Our model properly stores the data and “knowledge” and concrete rules of our game. It has the classes which are used to store and manipulate states of our objects as well as classes which represent the objects itself: ViewObj, TowerEnum, Tower, Projectile, PlayerFighter, Player, Model, Mob, Fighter, Enemy. Model class instantiates our Player from the start of the program and manages game properties such as wave and money. ViewObj class is the head class which represents all the objects projected on the screen. We have a multilevel inheritance in our system such that Mob class represents all the moving objects, and Projectile, Player and Enemy classes are inherited from it. Fighter class carries out all the properties for the objects that can fight such that it is passed to the Tower and Enemy through aggregation and to the Player class through the inherited class PlayerFighter.

The Ctrl is responsible for handling all the user inputs. It sets all the key bindings and is responsible for setting the frame for the game. It also controls the “waves” of zombies in the game and sets a handler for the “start wave” button. By the end of each wave you get a certain amount of money and zombies become stronger, so the level gets harder. The Ctrl takes care of the number of Fighters and Projectiles presented by storing them in two ArrayLists.

All audio in the game is handled by the GameNoise class. To create a GameNoise, pass a single string with the audio file path. To play that sound at any time in the future, simply call the play method on the instance.

The view folder contains all files required for setting the JavaFX Scenes. There are three Scenes the user can visit in ZTD47, the start/splash screen, the game screen, and the

game over screen. Each of these have a designated public class for instantiating, accessing, and manipulating them.

Three classes that represent three different screens are ViewStart, View and ViewGameOver. At first, we wanted to instantiate a StackPane such that screens are stacked on top of each other; however, it could have led to performance issues. Therefore, we kept different views and fxml files for each screen, so that designing and maintaining the screens would be easier. Three view classes allowed us to define clean and easy navigation among the screens. FXML files were loaded to separately in each view class through the *FXMLLoader* function. To each object defined in fxml file such as label or pane we were able to assign to assign an fx:id such that each button could be easily accessed using *lookup("#(some id)")* function. In order to make changes to the object defined in fxml it is necessary to cast it to the proper object after looking it up before assigning it to the variable. Our View class has many defined panes, buttons and labels, so in order to access them in the most efficient manner without repeating the code, we designed a system in Ctrl which calls all the id's and passes them as a string to a method with switch statements which assign methods to the buttons.

The class for controlling the main game view is titled View. It is divided into two parts: menu and game field. Therefore, it has two roots Pane menuRoot and Pane gameRoot, which are accessed separetly. It also sets all labels and health progress bar for the payer. The View handles the placement of the towers on the screen by working with TowerEnum enumeration.

The base class for everything that shows in the View is ViewObj. ViewObj instances display by adding an ImageView in a StackPane to the gameRoot. The ImageView handles the display of the sprite, while the StackPane handles transformation. Each StackPane's x-position, y-position, and rotation are bound to the x-position, y-position, and rotation of their associated ViewObj. To change these properties of a ViewObj, call setX, setY, and setDirection respectively. The associated getters are also implemented. ViewObj has a frame method. This method is called every frame and is passed the time in seconds since the last time frame was called. This method may be overridden to create custom functionality.

ViewObj instances are inherently stationary. To make a moving object in the View, use the Mob class, which is a child of ViewObj. Mob motion is controlled with a magnitude-angle rather than a velocity component in the x and y directions. To change the magnitude of a Mob's velocity, call setSpeed and pass in a double between 0 and 1 representing the fraction of the Mob's maximum speed at which to set the Mob's actual speed. This feature is designed to match the direction based style of the ViewObj. Mob overrides the ViewObj frame method to handle motion. This may be overridden, but there must be a super call for the Mob to move.

The Mob class is the parent of the Player class that controls all of the movement of the player. One major difference between the Player class and a base ViewObj is that Player uses two directions, the standard direction used by the ViewObj StackPane and moveDirection used by the Mob frame. The players direction is always towards the mouse, while the Player's frame

method handles the updating of standard direction and all responsibilities of the inherited frame method.

The base class for everything that participates in combat in the game is Fighter. Fighter have a few properties that are standard in modern gaming: range, frrt, power, projSpeed, projImage, tracks, health, team, viewObj, and reward, all of which are to be passed into the constructor, in that order. The range represents the maximum distance, in pixels, from their target that a Fighter can attack. The frrt is the number of attacks per second permissible for the Fighter. The power is the total damage, or reduction from the targets health, of a successful attack. The projSpeed and projImage of the Fighter represent the speed in pixels per second and image respectively of the projectile fired by the Fighter. If projImage is null, the Fighter is melee, meaning they do not fire projectiles and all attacks are immediately successful if the target is within range. The tracks argument is a boolean representing whether projectiles fired by the Fighter track their target, or if they continue in a straight line after firing. The health of a Fighter represents how much damage they can take before their die method is called, which will remove them. The team is an int representing which side of the conflict the Fighter is on; Fighters with the same team value cannot damage each other. The Player and Towers are on team 0 while Enemies are on team 1. Every Fighter is associated with their representation on the screen. The Fighter's associated ViewObj is passed as viewObj. Finally, reward is the money the Player gets when this Fighter dies; more difficult Enemys yeild a higher reward, while things like Towers have zero reward. Like frame in ViewObj, Fighters run their update method every frame. The update method is passed the duration in seconds since the last frame and a complete list of all of the Fighters currently in the game. This method handles automatic targeting of members of other teams and associated changes in the ViewObj.

While the Player is a combatant, many things about their combat is much different that standard Fighters. For example, when the Player dies, the entire game ends. To handle cases like this, the PlayerFighter class inherits from the standard Fighter. This method overrides the update method to avoid automatic targeting, as player firing is handled in the Ctrl.

Towers are ViewObj's that have an aggregated Fighter that controls it's targeting and firing. Towers are instantiated based on their x-position, y-position, and towerType. The towerType is an element of TowerEnum. TowerEnum contains all information about the different tower types. To create a new type of tower, simply create a sprite for the tower and projectile (if applicable), in any image format accepted by JavaFX, and add a line to TowerEnum with the proper information. The program will manage all other changes automatically, including adding a new button for that tower type.

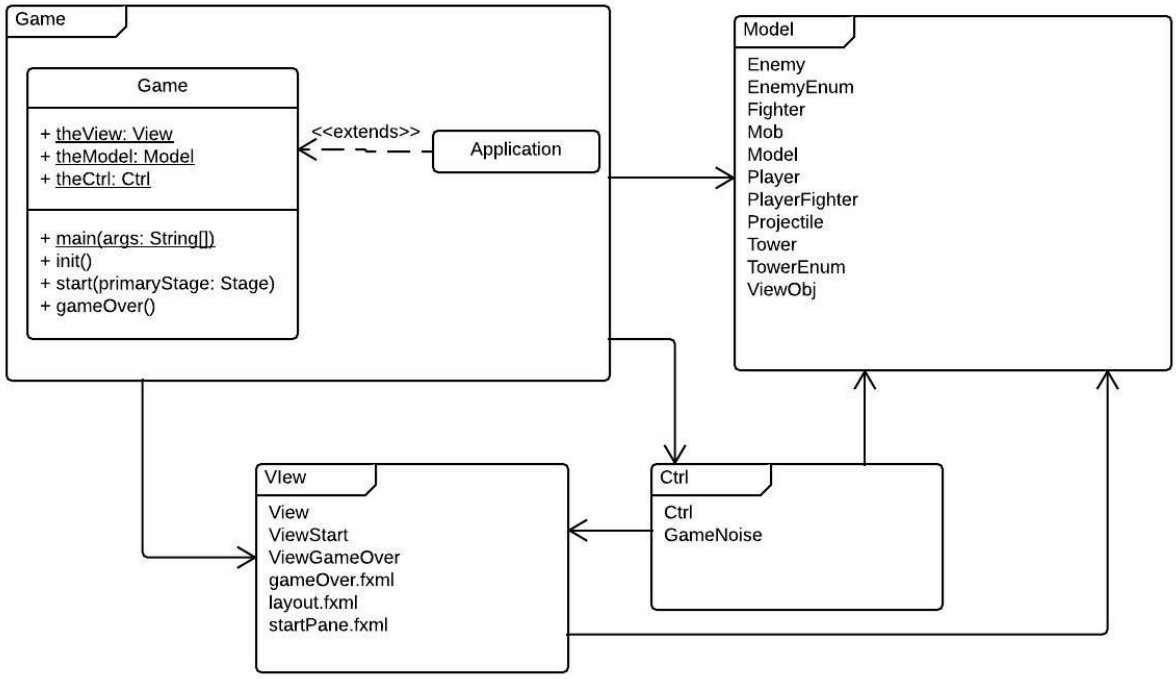
Enemies are what eventually ends the game for the Player. Enemies inherit from Mob, since they move after the Player. They have an aggregated Fighter that controls its direction and targeting. Enemies are instantiated based on their x-position, y-position, and enemyType. The enemyType is an element of EnemyEnum. EnemyEnum contains all information about the different enemy types, including reward. To create a new type of enemy for the game, simply

create a sprite for the enemy and their projectile (if applicable), in any image format accepted by JavaFX, and add a line to EnemyEnum with the proper information. The program will manage all other changes automatically, including balanced spawn rate of the new enemy in the waves.

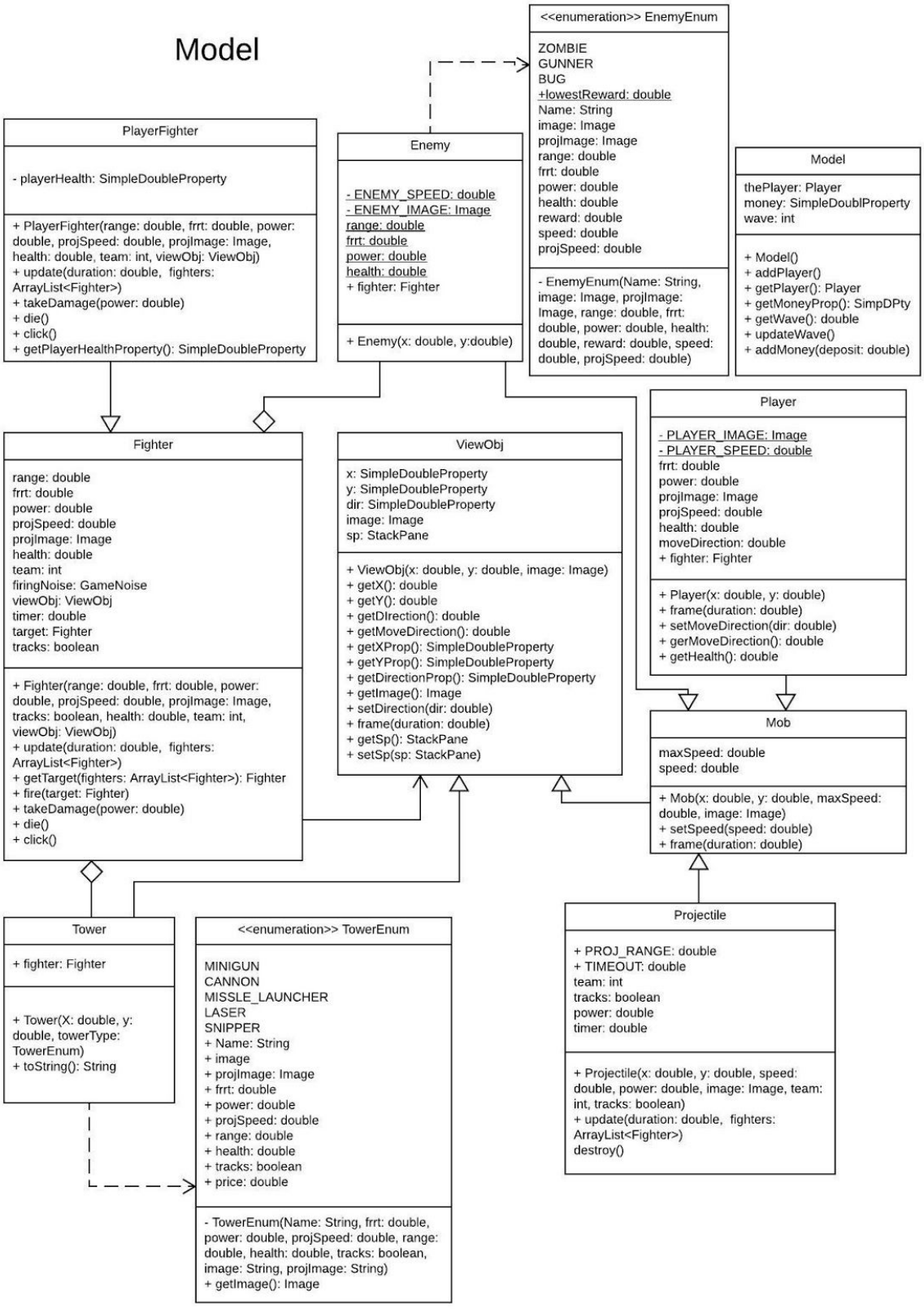
The Projectile class is a widely used child of the Mob class in ZTD47. Projectiles have speed, power, track, and team, passed directly from the Fighter that fired them. If the Projectile tracks, it changes direction in its frame method. Projectile deal damage equal to their power to any Fighter of another team that they collide with. Upon such a collision, the Projectile is immediately destroyed.

Throughout the entire process, we tried to keep our code as readable as possible. When we were running into complex problems we tried to formulate an algorithm such that our problem is broken up into sub-parts and then tied together with one solution. We avoided repetition by implementing general methods and tried to write everything in a short and concise manner. Practice of leaving comments in the code allowed everyone to be able to catch up on their own. We learned a lot from each other during ZTD47 implementation and we all consider it as a valuable experience. The whole project did not focus only coding, but also involved a lot of managing and refactoring in order to keep the system clean.

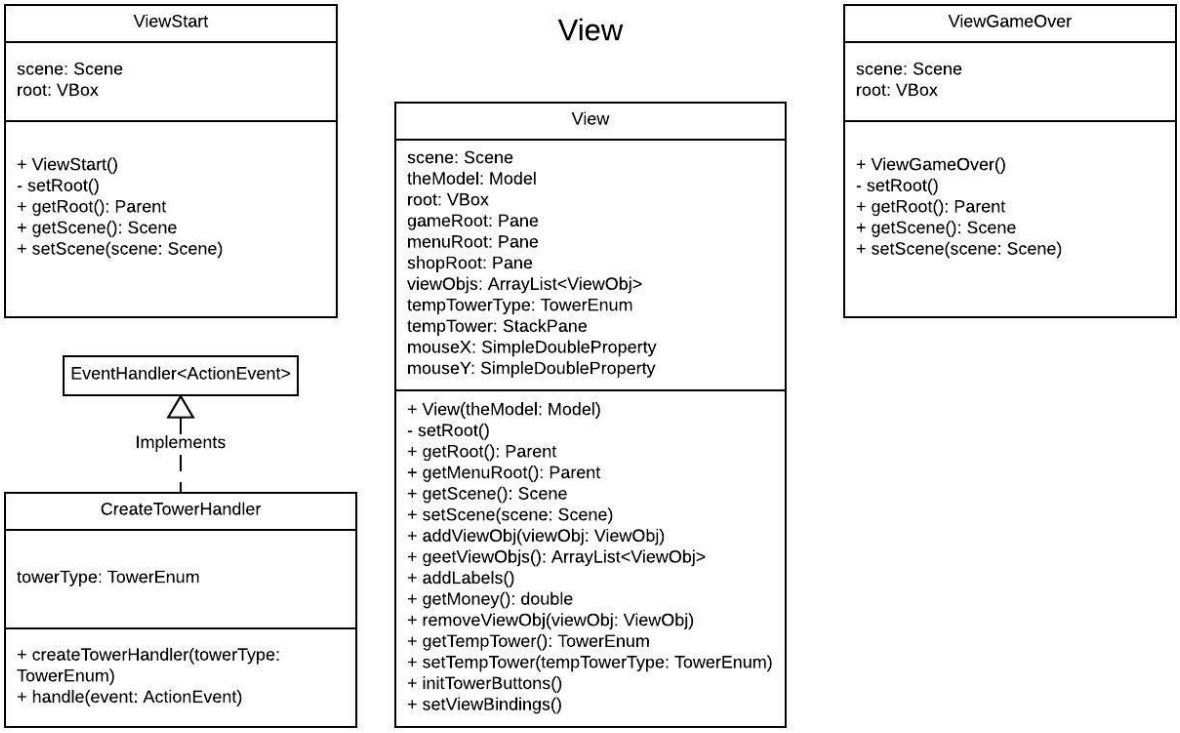
Packages:



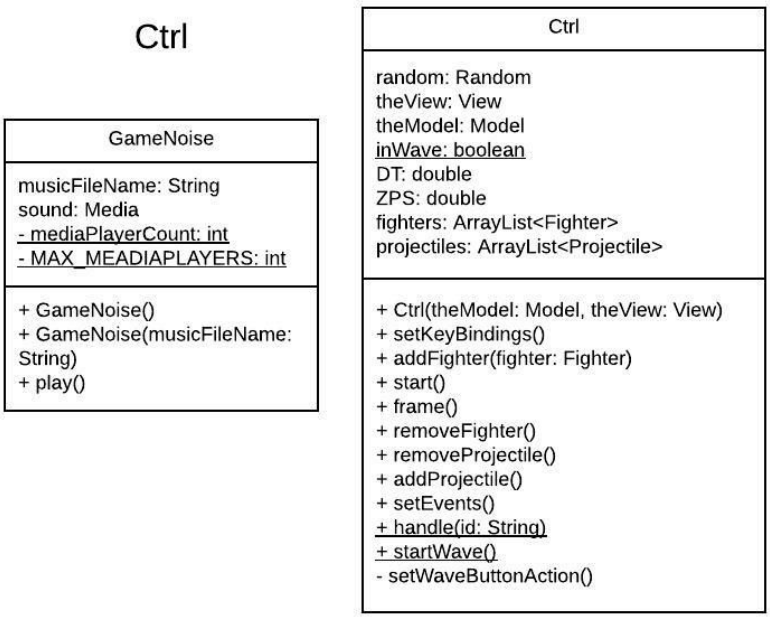
Model Package:



View Package:

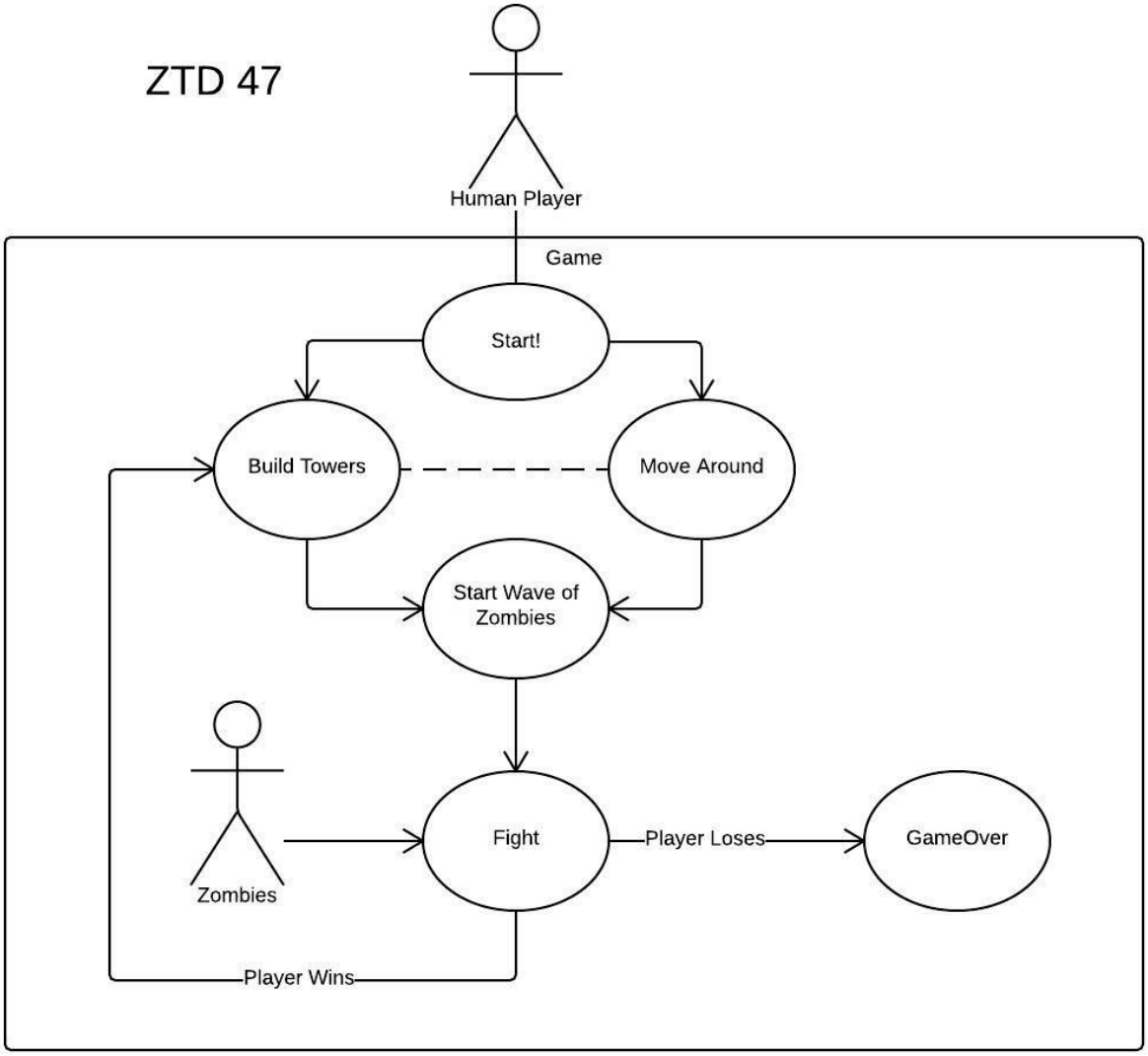


Ctrl Package:



UML Case Diagram:

ZTD 47


Human Player
Game
Start!
Build Towers
Move Around
Start Wave of Zombies
Zombies
Fight
Player Loses
GameOver
Player Wins