Design Manual

Joe Elvin, Yuxuan Huang, Jon Li

Introduction

We used Swing to develop this simple 2D game. The reason for using Swing over JavaFX is that although JavaFX is newer ,more advanced and more capable, Swing is more than sufficient for the scale and complexity of our game. Moreover, when we started learning about game development using Java, we were able to find more online help with Swing than with JavaFX . For the design of this game, we followed a very object-oriented approach. The entire game was broken down into most basic meaningful functional blocks. Polymorphism is also heavily used throughout the design in order to maximize the benefit of code-reusing and encapsulation. For the design of the GUI, we tried our best to follow the Model-View-Controller(MVC) design approach.

User Stories

Here are the user stories we initialize determined for this project:

- ***As*** *As a video game player, I want…*
    - *Start menu*
        - *Have buttons for “Story mode”, “Endless mode” and “exit”*
            - *The ability to play a “story mode” with 10 waves and a final boss*
            - *The ability to play a “endless mode” with a continuous wave of enemies until death (so the game is more playable as the player can still have fun after beating the game in story mode)*
    - *To defend a castle against enemies (So that the game pose challenge)*
        - *Enemies come from the right side of the screen*
        - *The castle has a HP bar*

    - *The castle is stationary on the left side of the screen*
    - *The player only have one life*
- *The player can move the main character on the terrain*
    - *“Left” key to move to left*
    - *“Right” key to move to right*
    - *“Up” key to jump(with gravity down)*
- *To upgrade the weapons my player can use (more playable)*
    - *Weapons have different damage and attack rate*
- *The game to get incrementally harder as it progresses (in waves)*
- *A range of enemies to fight(more fun!)*
    - *Different types of enemies come with different routes*
    - *Different types of enemies have different HP, damage and attack rate.*
- *To be able to pause the game anytime and able to resume where I was just before pausing*
- *After the game end.*
    - *“Story mode”: Return to start menu.*
    - *“Endless mode”: Display player’s score before return to start menu.*

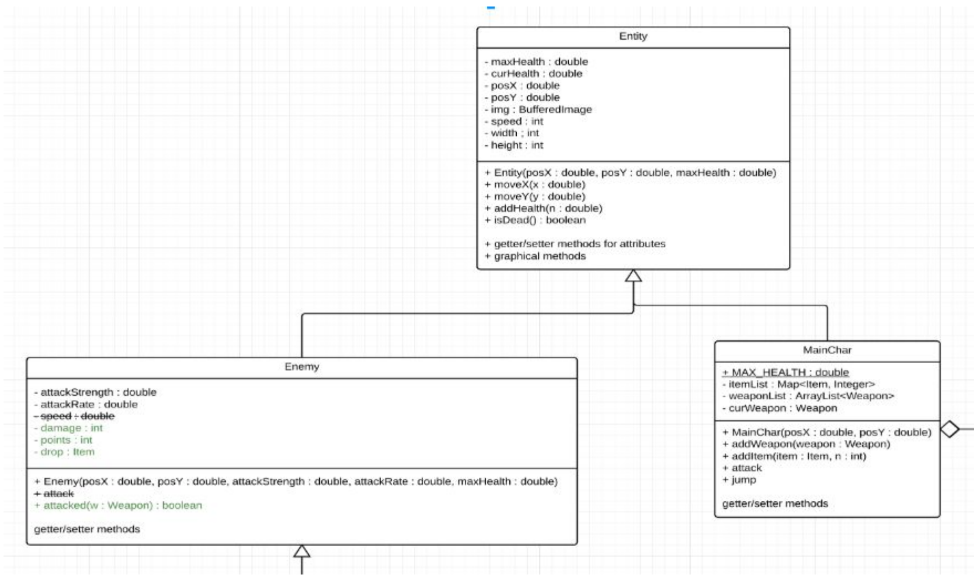
In order to accomplish these user stories, we extracted all the important nouns(such as “Start menu”, “player”, “enemies”, etc) from the user stories and create class object for each one of them. We also used inheritance, abstraction and polymorphism wherever make sense. For example, we have an abstract class called GameState where several other classes such as MenuState, PlayState, Paused State extend from. We have a superclass called Entity which has subclasses Enemy and MainChar. Such practice of OOD and agile development helped us achieve our user stories.

Object-Oriented Design

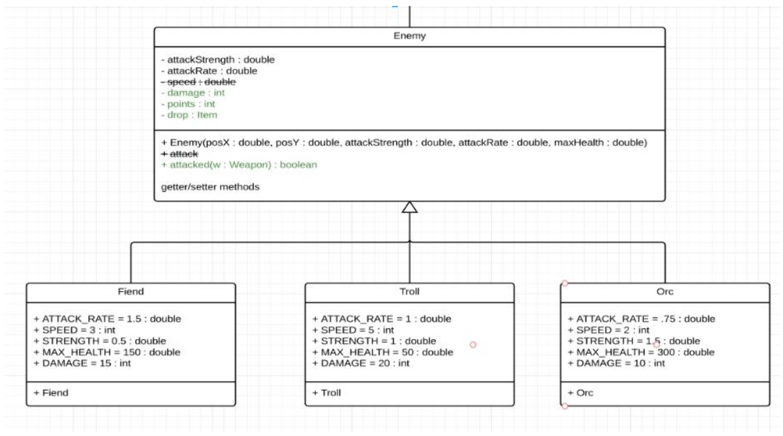
For this project, our team used a very object-oriented approach to accomplish our goal. The base game (non-GUI class files) operate on five main parent class files, for handing the Entities, Enemies (which are themselves extended from Entities), Items, Weapons, and Buildings. All classes which represent real objects which are instantiated in the game are child classes which inherit one of the five parent classes. For the GUI and game framework, we also followed the object-oriented approach as well as the Model-View-Controller architecture as much as possible.

Non-GUI Classes Design

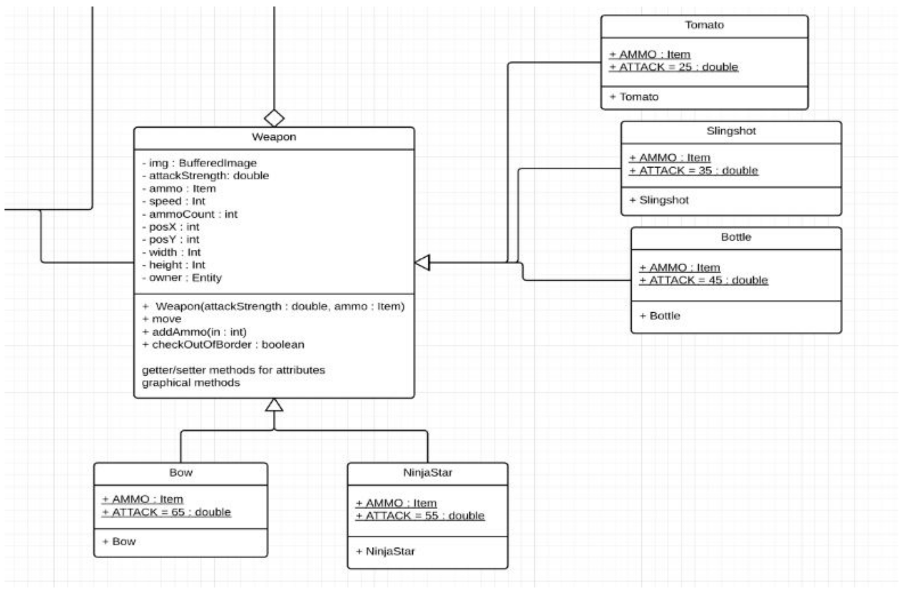
Our UML class diagram goes over these classes in detail. The first of these classes is the Entity class, which handles anything that can be considered a “living thing” within the game. Every entity is given things that any living thing might have, like a position (separated into x and y coordinates), a max health and current health, and a movement speed, and many more. The two main classes which derive from the Entity class are the Enemy class (which is a parent class itself) and the MainChar class, which is a special class purely designed for handling the main character which the user controls.



The Enemy class extends into three child classes; the Fiend, Troll, and Orc classes. All three of these child classes are similar; each has class variables which denote the enemy’s relative speed, damage, and health, which is different for each enemy. By setting these variables as class variables, we don’t have to worry about specifying an enemy’s parameters at every instantiation.

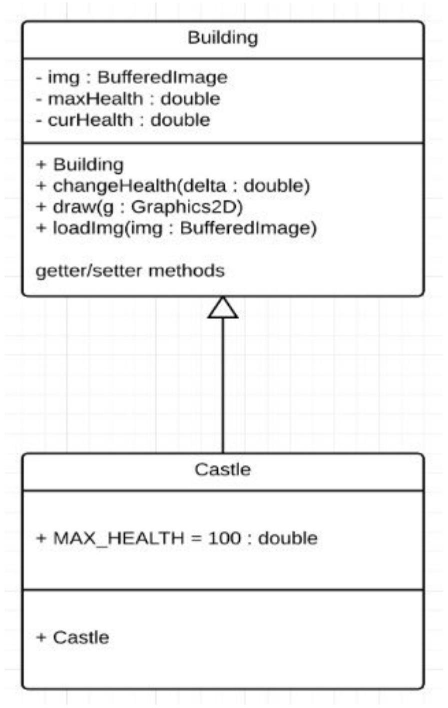

Enemy
- attackStrength : double
- attackRate : double
~~- speed : double~~
- damage : int
- points : int
- drop : Item
+ Enemy(posX : double, posY : double, attackStrength : double, attackRate : double, maxHealth : double)
~~+ attack~~
+ attacked(w : Weapon) : boolean
getter/setter methods
Fiend
+ ATTACK_RATE = 1.5 : double
+ SPEED = 3 : int
+ STRENGTH = 0.5 : double
+ MAX_HEALTH = 150 : double
+ DAMAGE = 15 : int
+ Fiend
Troll
+ ATTACK_RATE = 1 : double
+ SPEED = 5 : int
+ STRENGTH = 1 : double
+ MAX_HEALTH = 50 : double
+ DAMAGE = 20 : int
+ Troll
Orc
+ ATTACK_RATE = .75 : double
+ SPEED = 2 : int
+ STRENGTH = 1.5 : double
+ MAX_HEALTH = 300 : double
+ DAMAGE = 10 : int
+ Orc

The third parent class is the Weapon class, which aggregates into the MainChar class because the main character is assigned a weapon at every instantiation. From this class extend five child classes, representing the possible weapon upgrades the player can get. Like the Enemy classes, each weapon has class variables which represent the type of ammo the weapon uses and the damage the weapon does; this vastly simplifies the implementation of these classes into the all-encompassing GUI.



The fourth parent class is the Item class. These class act as placeholders for the physical items; all other item stats like damage are handled in the Weapons classes. This is why all items are nothing more than constructors, because they ultimately have very little to fulfill with respect to the project.

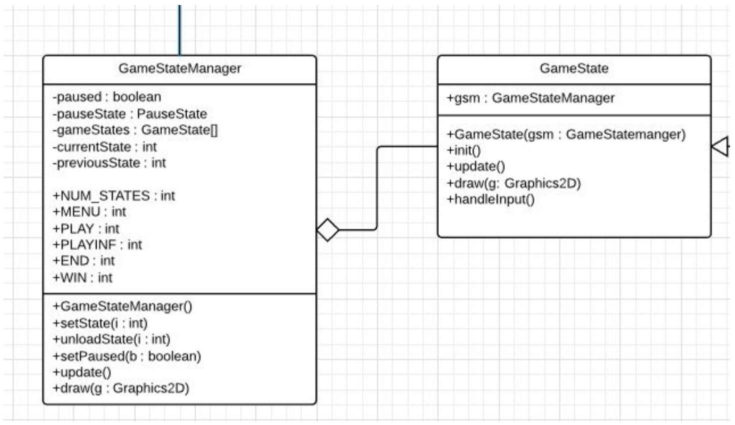
The fifth parent class is the Building class. Originally we were going to have upgradable buildings so the player would be able to fend off the enemies more efficiently later on, but this proved to be too tough of a challenge given the time constraint. Regardless, we left the Building class open-ended and easily expandable, so future updates could allow for more buildings with better features. Each building has maximum and current health parameters; they need little more than this, because this is all they represent in the game.



GUI & Game Framework Design

For the GUI of the game, all classes are organized under a main class which holds the JPanel. Under the main class, there is a Game class which update the frame by defined FPS and take key interceptions. The Game class also host the main controller of the game called GameStateManager.java which control the switch among different game states.

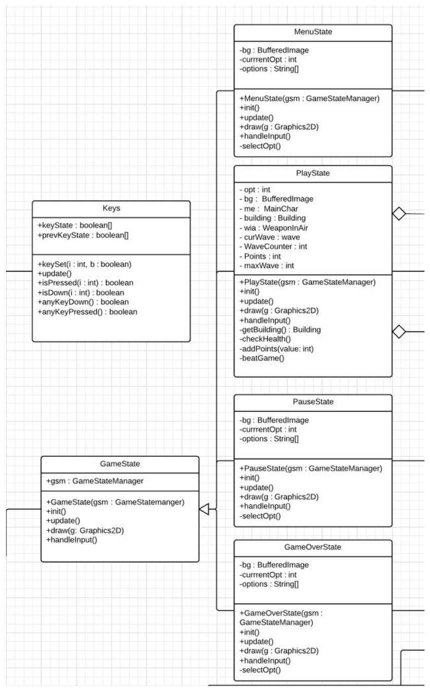
There are five different instantiable game states designed: MenuState, PlayState, PauseState, WinState and LossState. All the states apart from the PlayState are fairly simple, they all serve as some menu which allows user to choose desired actions, such as exit or change the game to another state. The PlayState holds the core of the game. The PlayState enable all objects(mainChar, weapons and enemies) to move frame by frame on the JPanel and interact with each other. Each of these states are designed to host both controller and view. Due to the nature of JPanel, we found that

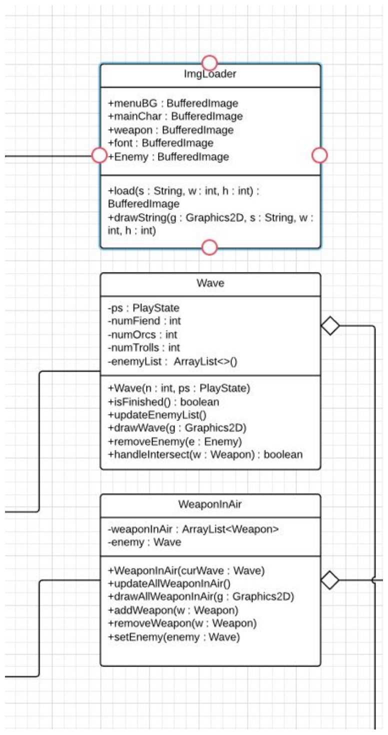


it is difficult and unnecessary to separate the controller and view for the GUI design. If you look into any of the states, you will find two overriding methods called update and draw. Basically, the controller is under the update method while the view is under the draw method.

For the models, basically all the non-GUI classes are used as models in the design. There are three additional classes called Wave, WeaponInAir and WeaponOnGround. These are simple classes designed to host lists of enemies and weapons to be easily instantiated and accessed in the PlayState.

We also have two major Utility classes, Keys and ImgLoader. Keys detects key interruptions and ImgLoader loads graphics images for all the objects. These two classes are non-instantiable.


MenuState
-bg : BufferedImage
-currrentOpt : int
-options : String[]
+MenuState(gsm : GameStateManager)
+init()
+update()
+draw(g : Graphics2D)
+handleInput()
-selectOpt()
Keys
+keyState : boolean[]
+prevKeyState : boolean[]
+keySet(i : int, b : boolean)
+update()
+isPressed(i : int) : boolean
+isDown(i : int) : boolean
+anyKeyDown() : boolean
+anyKeyPressed() : boolean
PlayState
- opt : int
- bg : BufferedImage
- me : MainChar
- building : Building
- wia : WeaponInAir
- curWave : wave
- WaveCounter : int
- Points : int
- maxWave : int
+PlayState(gsm : GameStateManager)
+init()
+update()
+draw(g : Graphics2D)
+handleInput()
-getBuilding() : Building
-checkHealth()
-addPoints(value: int)
-beatGame()
PauseState
-bg : BufferedImage
-currrentOpt : int
-options : String[]
+PauseState(gsm : GameStateManager)
+init()
+update()
+draw(g : Graphics2D)
+handleInput()
-selectOpt()
GameState
+gsm : GameStateManager
+GameState(gsm : GameStatemanger)
+init()
+update()
+draw(g: Graphics2D)
+handleInput()
GameOverState
-bg : BufferedImage
-currrentOpt : int
-options : String[]
+GameOverState(gsm : GameStateManager)
+init()
+update()
+draw(g : Graphics2D)
+handleInput()
-selectOpt()


ImgLoader
+menuBG : BufferedImage
+mainChar : BufferedImage
+weapon : BufferedImage
+font : BufferedImage
+Enemy : BufferedImage
+load(s : String, w : int, h : int) : BufferedImage
+drawString(g : Graphics2D, s : String, w : int, h : int)
Wave
-ps : PlayState
-numFiend : int
-numOrcs : int
-numTrolls : int
-enemyList : ArrayList<>()
+Wave(n : int, ps : PlayState)
+isFinished() : boolean
+updateEnemyList()
+drawWave(g : Graphics2D)
+removeEnemy(e : Enemy)
+handleIntersect(w : Weapon) : boolean
WeaponInAir
-weaponInAir : ArrayList<Weapon>
-enemy : Wave
+WeaponInAir(curWave : Wave)
+updateAllWeaponInAir()
+drawAllWeaponInAir(g : Graphics2D)
+addWeapon(w : Weapon)
+removeWeapon(w : Weapon)
+setEnemy(enemy : Wave)