DSA Report

A simple game written in Java applying dijkstra Algorithm

Members

Lê Trần Phong - ITITIU 19180

Trịnh Quang Anh - ITITIU 19002

Nguyễn Ngọc Minh Nhật - ITITIU 19172

Demo Day

07/06/2021

I. Introduction

In this report, we will present our brand-new version of the Boom game. This is a well-known game having lots of variations but still keeping the classic gameplay, so it is quite challenging to create this game from scratch as a unique variant. We will give you a brief and terse overview of our game. The following sections below are categorized as follows:

- Section II provides the link to the github repository.
- Section III presents the firstlook of the game, its rules and extra features.
- Section IV explains the design with UML class diagram.
- Section V discusses the application of Dijkstra Algorithm.
- Section VI illustrates the design pattern(s), structures and principles.
- Section VII contains our self-evaluation.
- section VIII discusses the prospect for the game advancement.

II. Github repository:

Click the link over *here* to visit our github repository for having more details about the project by viewing the source code.

III. Game Overview, Rules and Extra Features:

Gameplay and Rules:

As we are approaching to design the gameplay as simple as possible for the player to comprehend but still keep the classic play style, so we created the Dungeon mode which consists of three phases. Before getting

started, players must choose one of six characters on the character selection screen in order to start the game. The rules of the game are simple, the player needs to place bombs to kill all the monsters without being touched by them. We developed dropping-item features with the purpose of boosting the character stats whenever the monsters are killed or blocks are destroyed and they drop some power-up items randomly, then he/she picks up one of those potions to gain power for the character.

The first phase is designed with numerous different pokemon monsters, as we are creating a unique version, so we just came up with the idea of making those cute pokemon into scary monsters of our game. The map of this phase is quite enormous as we planned to make the players feel as excited as possible.

The second phase is extremely tough as we add an elite boss with many monsters and increase their basic stats such as speed, damage, maximum health. And for the elite boss, it is not easy to be killed as other normal monsters as it must dominate over all the basic stats, then with the special abilities to be the second-phase boss.

The final phase is even harder for those players who are struggling to overcome the second round as this is when we introduce to the player our final boss of this dungeon. The map is two times bigger and there will be a lot of traps located near the boss. We must make the game as tough as we can in order to give the players great experience as well as curiosity while playing.

Extra features:

We have recently added only one extra feature. That is the multiple-explosion mechanism. The player's bomb can have numerous kinds of

explosions whenever the player picks up an explosion-boosting potion. We developed eleven types of explosions that burst in two extraordinary ways.

IV. Class Diagrams and Design Explanation:

Firstly, we applied the MVC structure for approaching a clean codebase as we can easily maintain our game due to separation concerns.

- *Model:* Representing the data of the application and responsibility for interacting with the database
  - *MapModel*
  - *BlockModel*
  - *ObstacleModel*
  - *TrapModel*
  - *PlayerModel*
  - *MonsterModel*
- *View:* Responsibility for displaying the model data, or interfaces for the users
  - *CharacterView*
  - *GameView*
  - *GameOverView*
  - *PauseView*
  - *MenuView*
- *Controller:* Handling and contains the flow of logic for the application and determines what response to send back to a user via view whenever requests are sent.
  - *CharacterController*
  - *GameController*
  - *MenuController*

➔ As utilizing the MVC architecture framework, we benefited quite a lot. The two main advantages that we considered it is crucial for our project's development are:

- ◆ *Rapid and Simultaneous development*
- ◆ *Reusability*

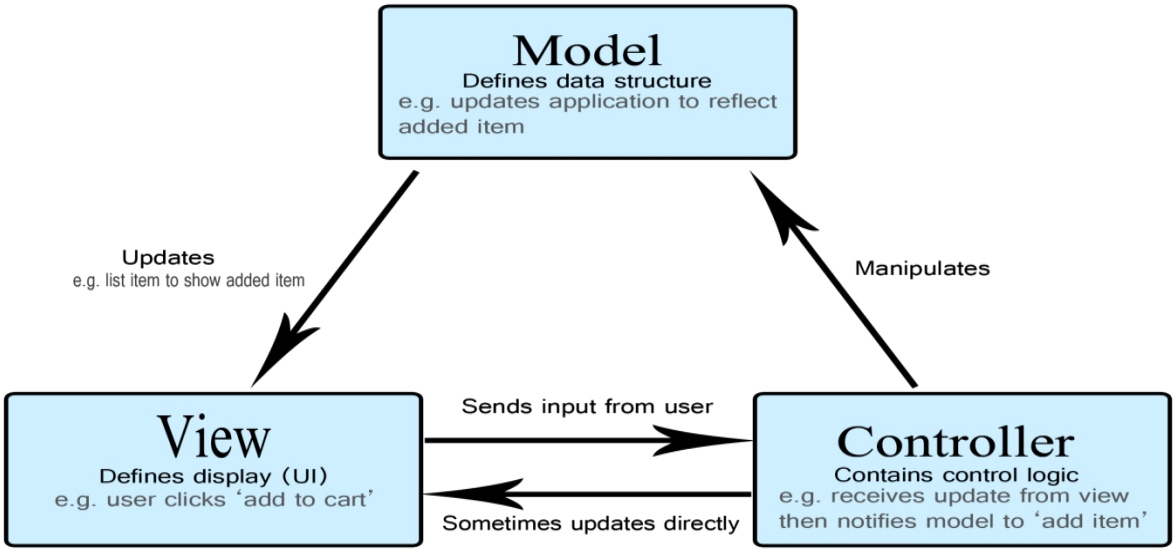

Figure 0. MVC Diagram

Secondly, we organized the whole game's entities and their behaviors inside a components folder.

In some essential classes, there will be an appearance of the **tick()** method which allows our game to update the data in every single time.

First of all, we created one basic abstract class to represent all game entities: *Entity*

This basic class will contain numerous variables and methods standing for the entity properties, and three main methods are:

- ❏ *LoadAllFrames():* Responsibility for rendering all images
- ❏ *initializeActions():* Actions triggering
- ❏ *setEntityParameters():* Setting default attributes(Damage, Hp, Speed….)

Then, from this base class, we separated the entities into two kinds of objects, dynamics and statics, as two abstract subclasses to categorize the entities further.

- ❏ DynamicEntity
- ❏ StaticEntity

About the **DynamicEntity**, we initialized an abstract class called DynamicEntity representing those entities capable of movement and created two more abstract subclasses to distinguish characters between players and monsters. Then, using polymorphism, we extend each abstract class above to create various concrete classes, which each class overwrites the common methods.

- ❖ Player
  - ➢ *Goku*
  - ➢ *Kid*
  - ➢ *Kirito*
  - ➢ *Satoshi*
  - ➢ *Shadow*
  - ➢ *Monk*
- ❖ Monster
  - ➢ *AlolanPersian*
  - ➢ *Bulbasaur*
  - ➢ *Emboar*
  - ➢ *Scorbunny*

- *ShinyEmboar*
- *ShinyZygarde*

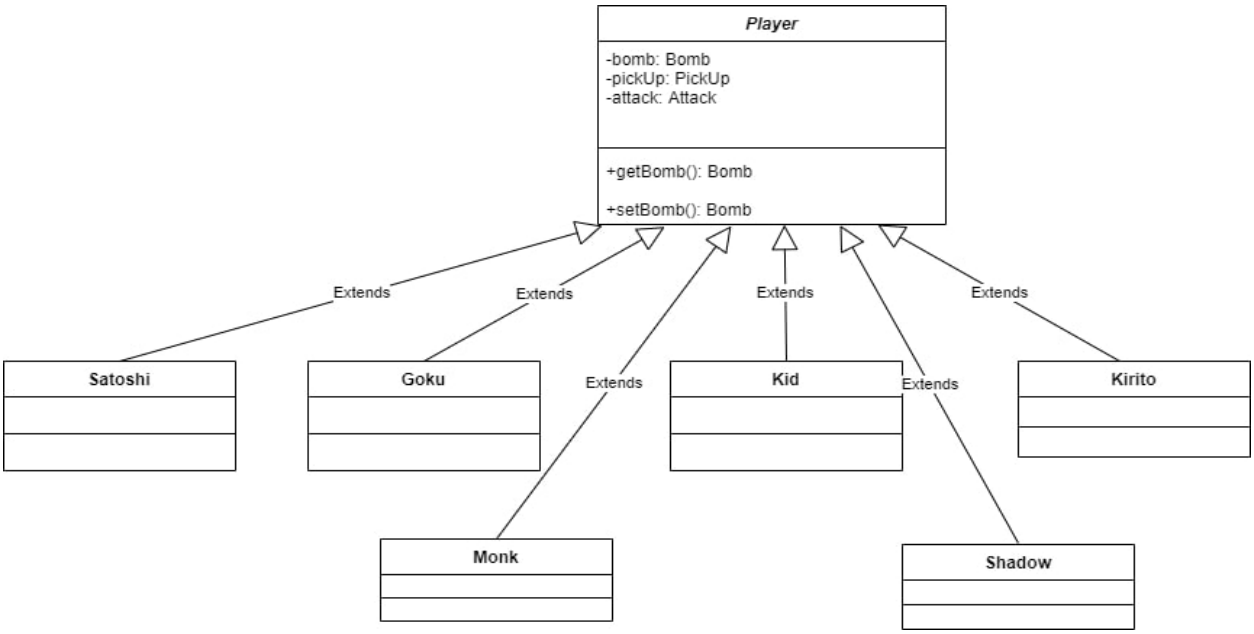

Figure 1. Players Diagram

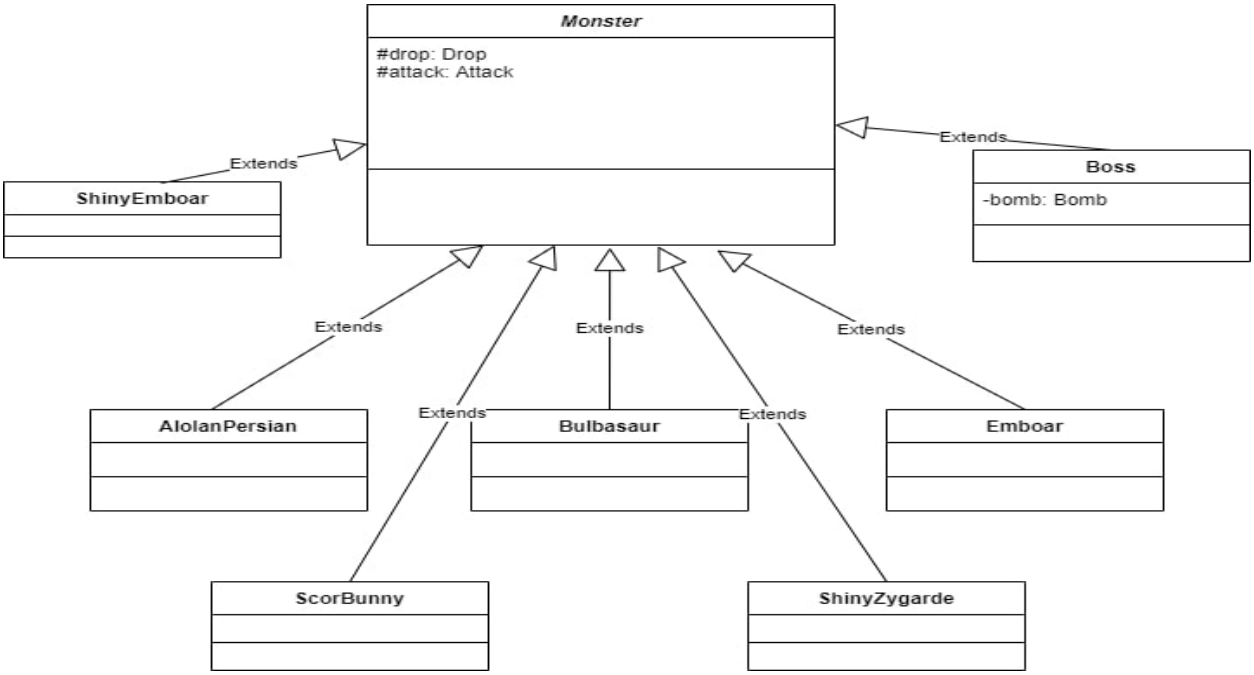

Figure 2. Monsters Diagram

Same as the **StaticEntity**, we also initialized an abstract class called StaticEntity representing those entities which are not capable of movement, we included three abstract subclasses called bombs, explosions, items, obstacles, and traps. And we also applied polymorphism by expanding each of these abstract classes below to create numerous subclasses allowing them to override unique methods.

We applied the **Prototype Design Pattern** in the **Bomb** and **Explosion** class with the purpose of avoiding creating new bombs or explosions objects during the game process which is considered harmful (Using **new** operator)

- Bomb
  - *BombA*
  - *BombB*
- Explosion
  - *ExplosionA*
  - *ExplosionB*
  - *ExplosionC*
  - *ExplosionD*
  - *ExplosionE*
  - *ExplosionF*
  - *ExplosionG*
  - *ExplosionH*
  - *ExplosionJ*
  - *ExplosionK*
  - *ExplosionL*
- Item
  - *DamagePotion*
  - *HealthPotion*
  - *Life*
  - *SpeedPotion*

- *Obstacle*
  - *Rock*
- *Trap*
  - *Lava*

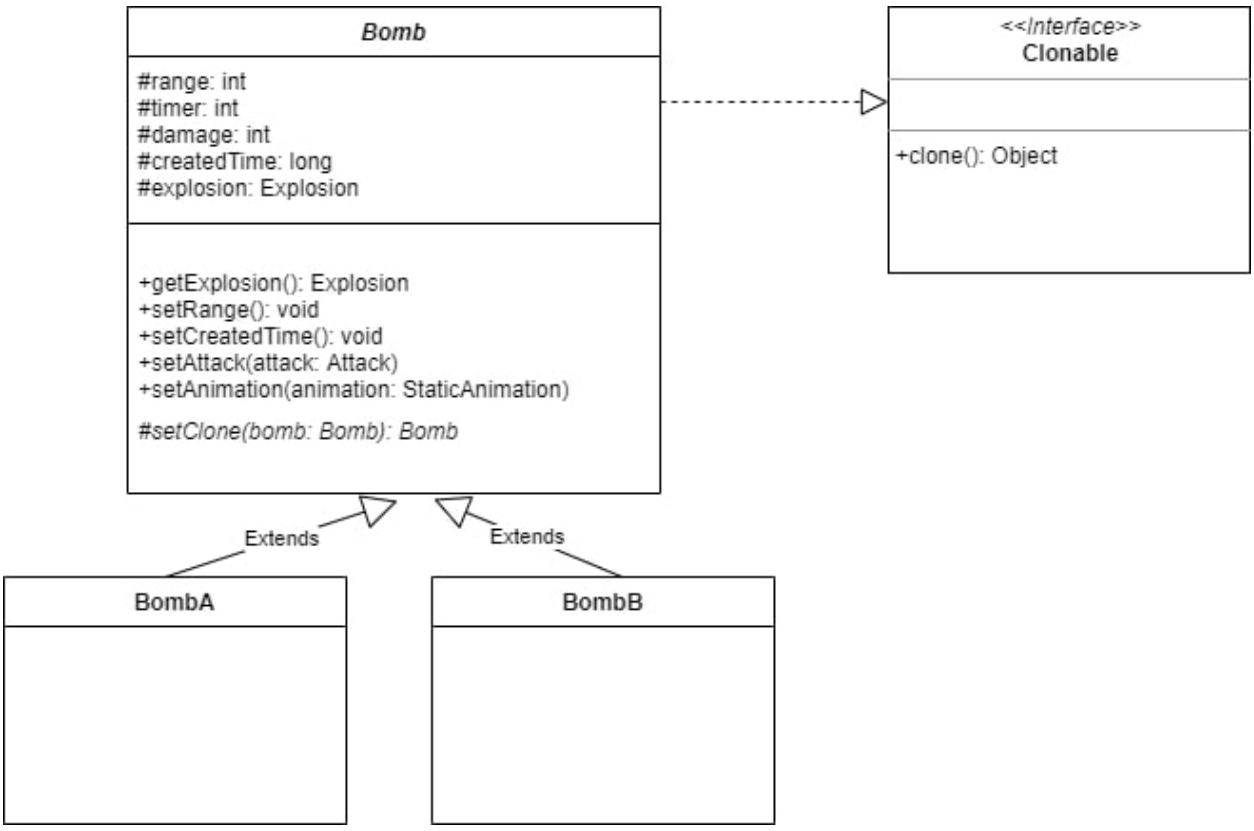

Figure 3. Bombs Diagram

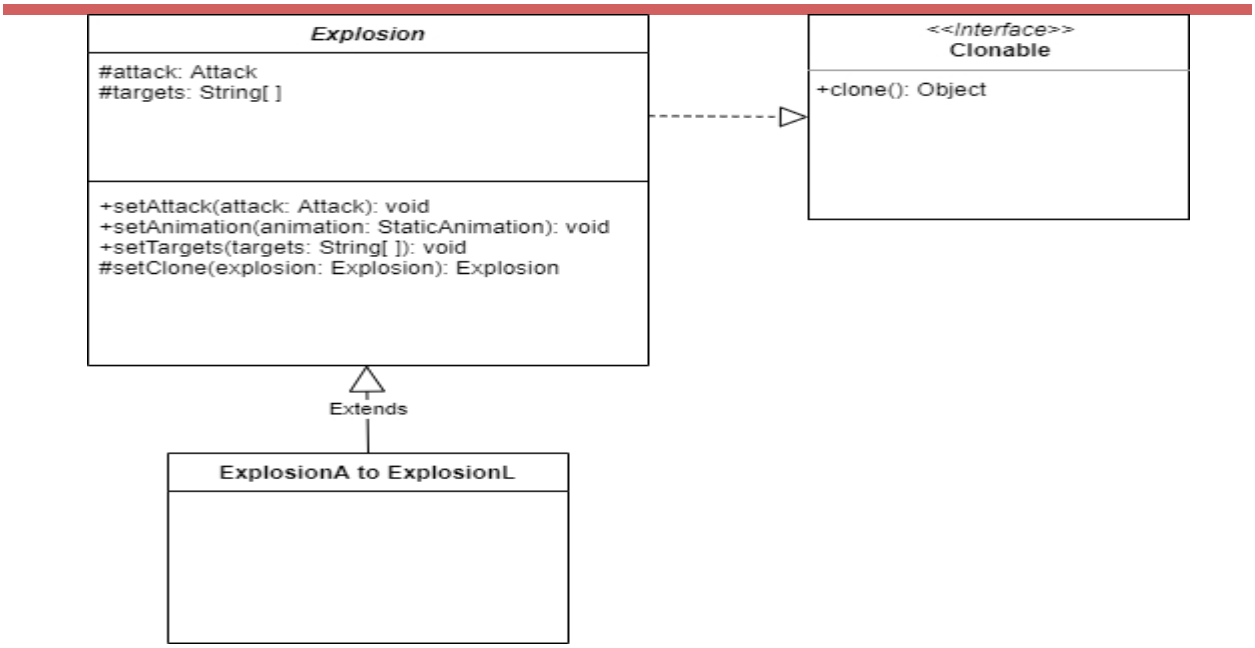

Figure 4. Explosion Diagram

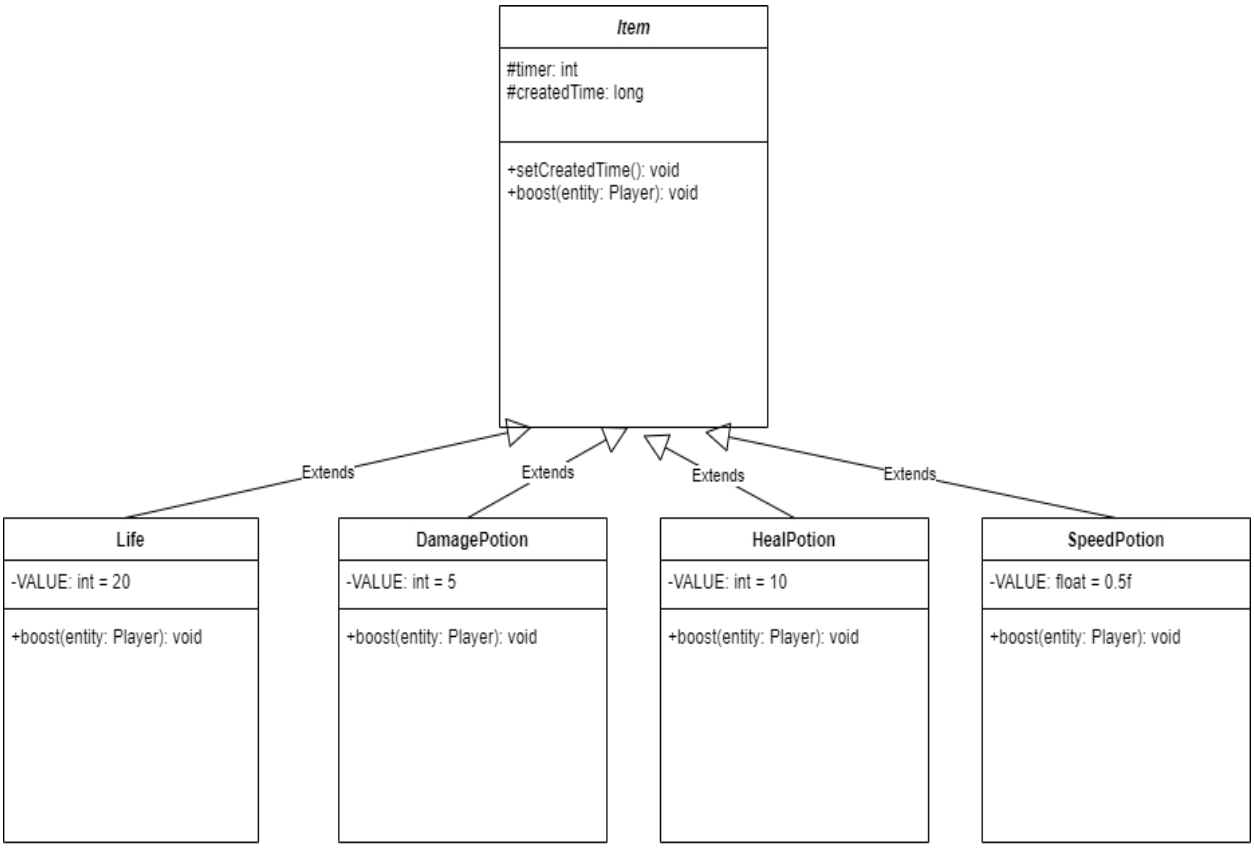

Figure 5. Items Diagram

Besides those entities, we also created an abstract class called Tile and its subclasses so as to construct different obstacles in the game. The most important method of this abstract class is the isSolid() boolean method as it will check whether the *"children"* of this class are impenetrable obstacles or not.

- The first types of tiles are **corner**, comprised of:
  - *LowerLeft*
  - *Plus*
  - *UpperLeft*
- The second types of tiles are **floor**, including:
  - Floor01
  - Floor02
  - Floor03
- The third type of tile are wall, consists of:
  - Wall
  - UnderwaterWall
  - Fluorescent
- The fourth type of tile are water:
  - Drain
  - HorizontalBridge (Not Solid)
  - VerticalBridge (Not Solid)
- The fifth type of tile are horizontal
  - RightBoundedHorizontal
  - Horizontal01
  - Horizontal02
- The sixth type of tile are vertical

- Vertical01
- Vertical02
- TopBoundedVertical
- BottomBoundedVertical

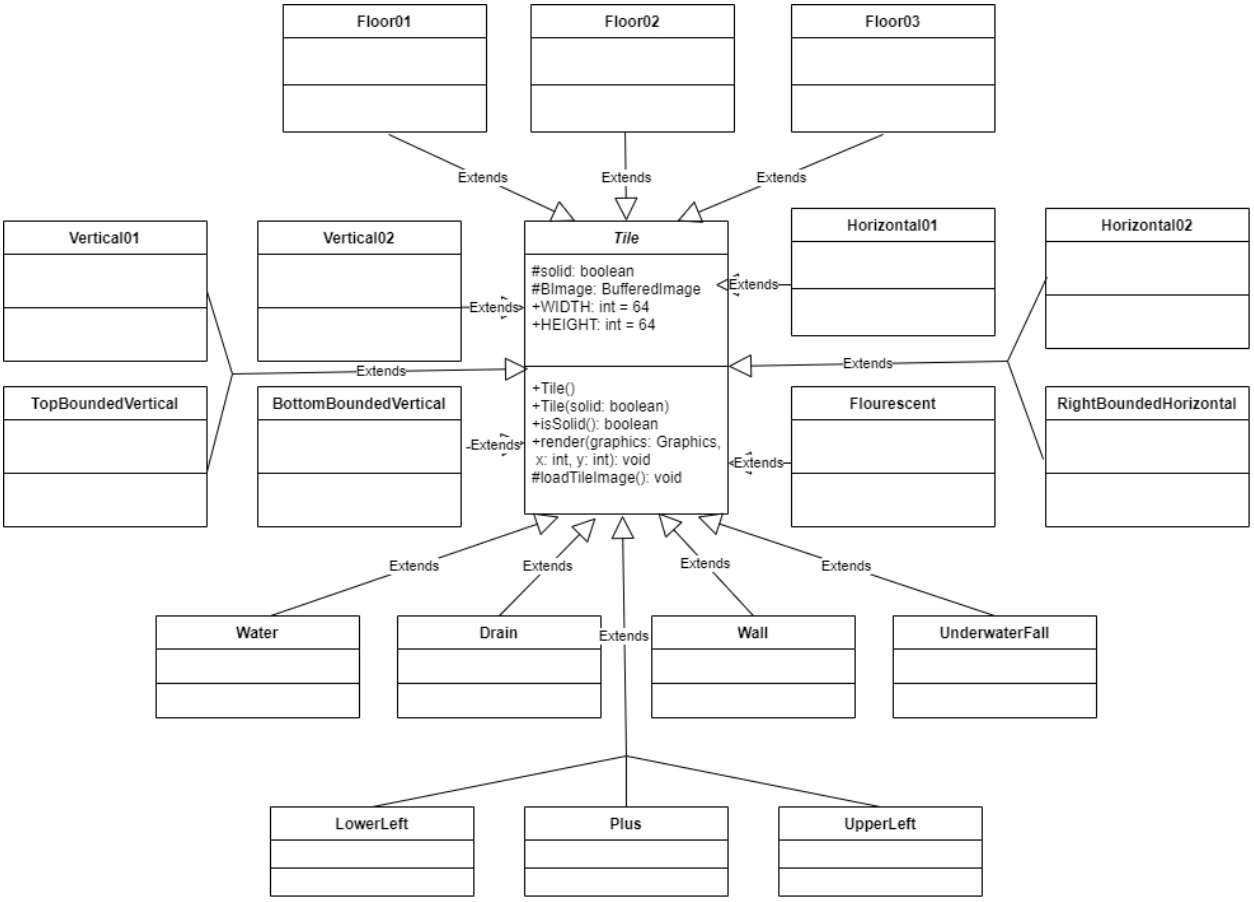

Figure 6. Tiles Diagram

In order to manage the actions of all entities in our game, we decided to implement seven distinct actions by creating them as an interface class and put them inside a package called actions. This is where we try to apply the **Decorator Design Pattern Principle** in the whole **actions** package except for the package **move** in order to increase the reusability of our codebase as well as improve the flexibility and

versatility of our code in every purpose without repeating so many code sentences having the same characteristics or functions.

The first one is **Animation abstract class**: Allow all entities to have their own motions in every second during the gameplay. We also splitted out two kinds of animation including **MoveAnimation** class and **StaticAnimation** class extending from the original one.

- MoveAnimation: Update motions every second based on four directions (Up, Down, Left, Right).
- StaticAnimation: Update the position of static objects

The second is all about the attack mechanisms which is implemented as an interface class and has three methods:

- *attack()*
- *setAttacker()*
- *getAttacker()*

We created two subclasses implementing this interface class so as to override the method attack in a versatile way.

- *AttackAction*
- *AttackDecorator*

We then initialized six subclasses of **AttackDecorator** so as to distinguish the attack mechanism operation between the 3 types of attack and 3 creations of entities.

- BlockAttack
- MonsterAttack
- PlayerAttack
- ExplosionCreating

- BombPlacing
- PlusExplosionCreating

We also implemented three subclasses from three different original classes so as to control the "attacker" and the targeted victims.

- ControlledBombPlacing
- ControlledMonsterAttack
- ControlledPlayerAttack

Besides, we created the **RandomBombPlacing** class extending from **BombPlacing** to override to decorate() method so as to place the bomb randomly.

- RandomBombPlacing

Thirdly, about the **Collide** , this is constructed as an abstract class which is implemented by two subclasses called **CollisionAction** and **CollisionDecorator** owning a boolean method to check collision between targets in every direction and define every type of entity during the collision.

- *isCollied()*
- *isColliedTop()*
- *isColliedBottom()*
- *isColliedLeft()*
- *isColliedRight()*

We first implemented an abstract class named **EntityCollisionDecorator** and created six subclasses extending the original one with the purpose of checking and defining the collision impact of all the entities in every direction as well as distinguish whether there are damages or not.

- *BlockCollision*

- *BombCollison*
- *ItemCollision*
- *MonsterCollision*
- *ObstacleCollision*
- *PlayerCollision*

Then, we distinguish the collision between the solid tile by creating the **TileCollisionDecorator** abstract class and allowing the class **SolidTileCollision**

- SolidTileCollision

Fourthly, **Display** interface class is created with the purpose of rendering health bars and the name of monsters and players in the game.

- *Display()*
- *getEntity()*

Also, we created a concrete class and an abstract class to implement the interface in order to display the actions and "decorate" of entities.

- *DisplayAction*
- *DisplayDecorator*

So, we constructed the HealthDisplay inside the **nonstop** package as a subclass of DisplayDecorator so as to display text, health, and box.

- HealthDisplay

Fifthly, we created an abstract class known as **Move** and created many functions to check the moving animation based on direction as well as avoiding collision.

Moreover, we decided to make our dynamic entities move in two ways: randomly and being controlled. So, we build up two more subclasses

extending from the **Move** abstract class and override the move method based on logical algorithms.

- *KeyboardBasedMove: Using keyboard as a support service to control entities*
- *RandomMove: Allows entities to move randomly*
- *Follow: Especially for the monsters with the path-finding skill*

Sixthly, we initialized the Item-dropping features inside the **drop** package. We first created an interface class named **Drop** and allowed a concrete and and an abstract class to implement the interface.

- DropAction
- DropDecocartor

We then made a class as a subclass of DropDecorator so as to categorize the items as well as set the position where it will appear in the map.

- ItemDrop: Life, DamagePotion, SpeedPotion, HealPotion

We also constructed another subclass of ItemDrop with the purpose of setting random item-dropping mechanisms.

- RandomItemDrop

Lastly, we created the items-picking up mechanism with the purpose of supporting the power of our characters during the game. The pickup Package contains the **Pickup** interface class including two methods called **getEntity()** and **pickUp()**. And we implemented the **PickUpAction** class as well as **PickUpDecorator** abstract class to override this method so as to create the item-picking up features. We also set up a **ItemPickUp** concrete subclass extended from **PickUp** to boost the stats of character whenever each item is picked up. Also, we controlled the

items by constructing the ControlledItemPickUp extending the ItemPickUp class.

- *PickUpAction*
- *PickUpDecorator*
  - *ItemPickUp*
    - *ControlledItemPickUp*

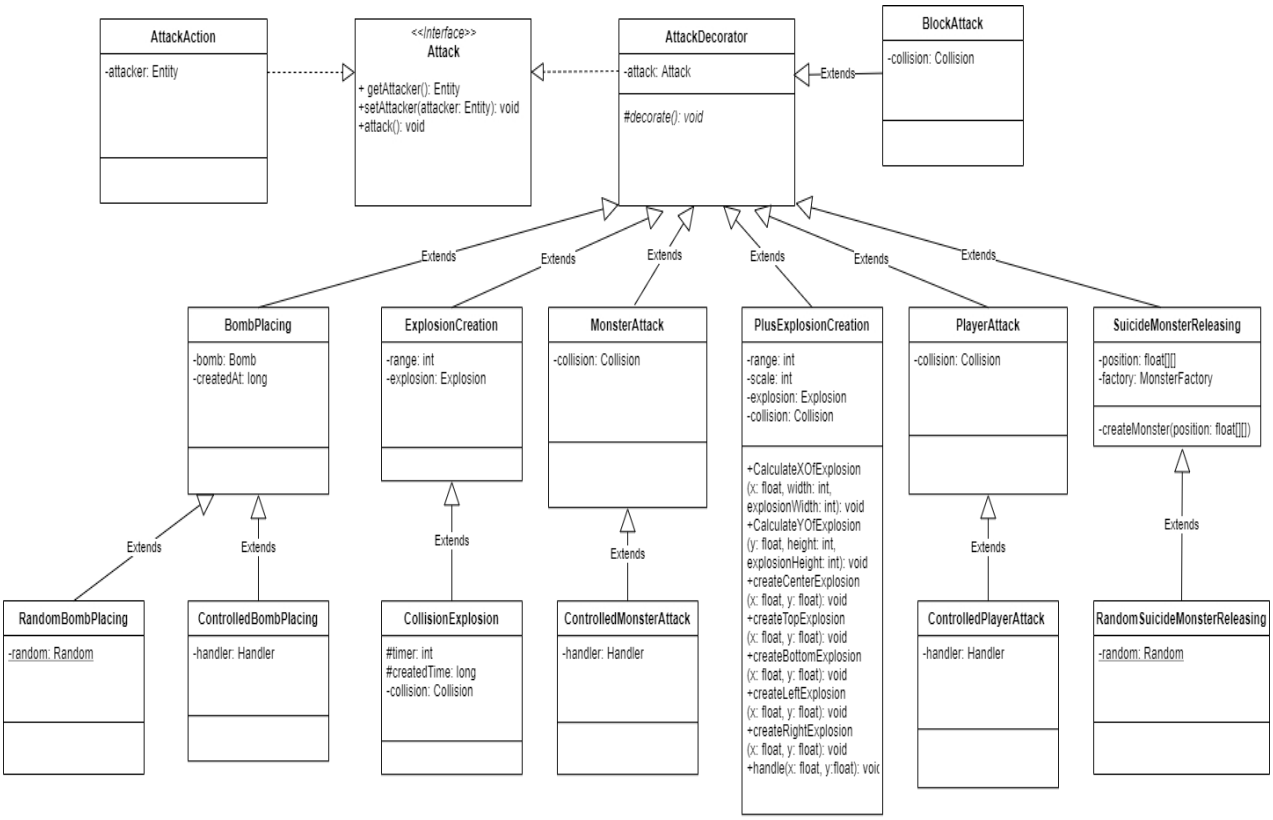

Figure 7.1 . Attack Diagram

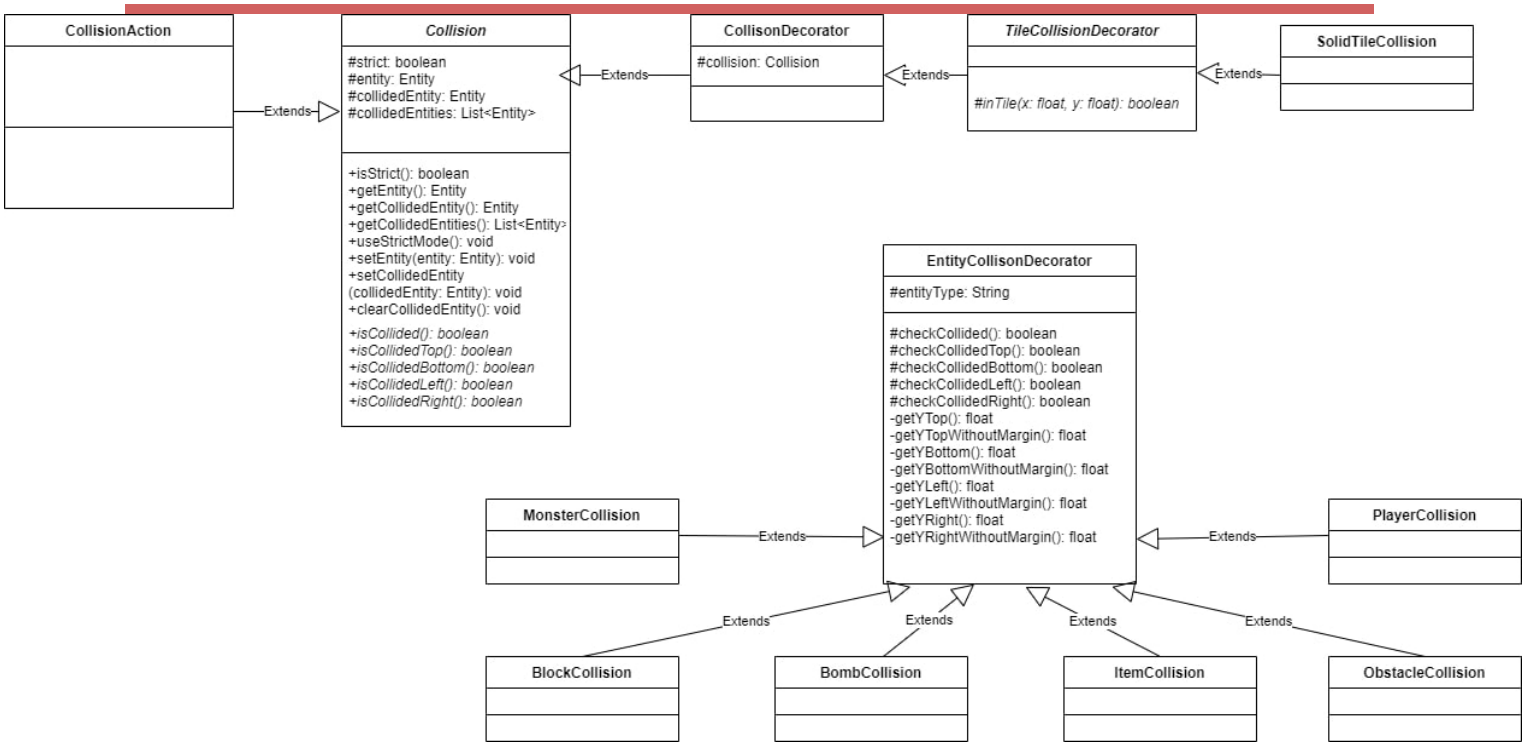

Figure 7.2. Collision Diagram

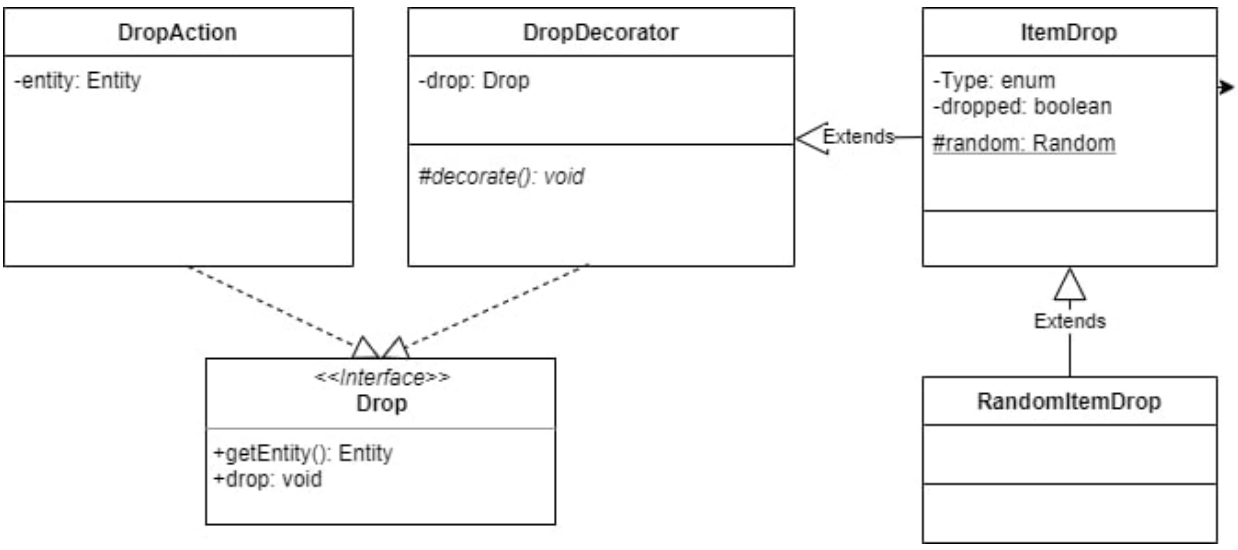

Figure 7.3. Drop Diagram

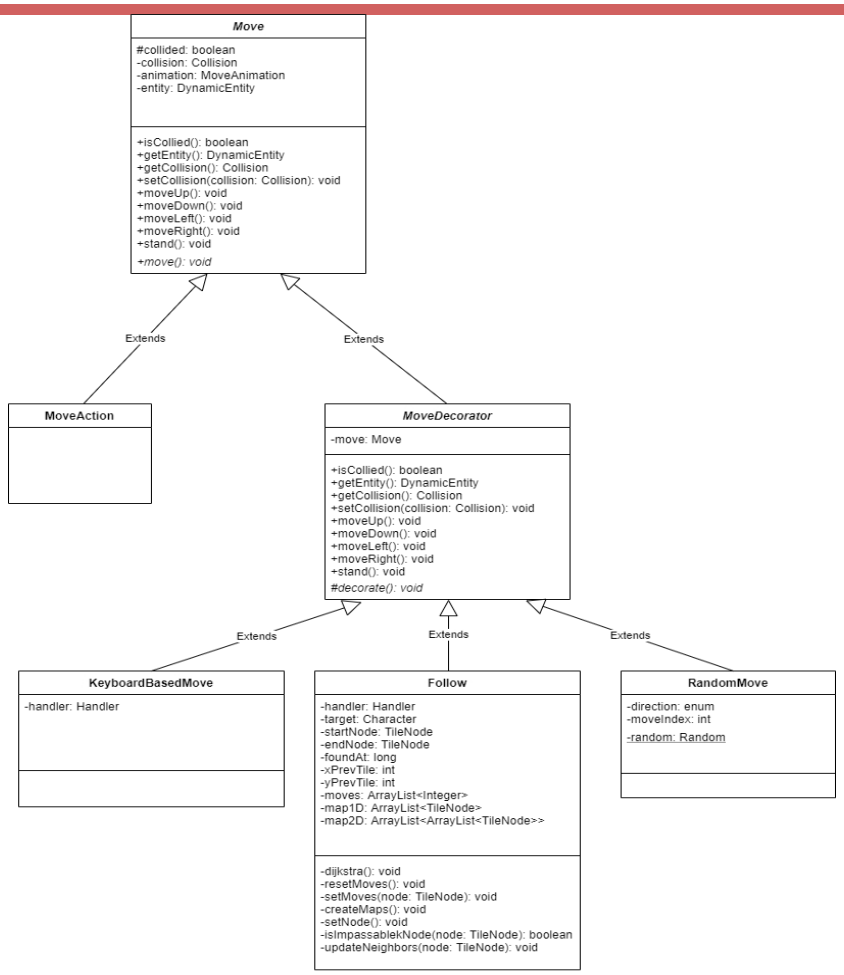

Figure 7.4. Move Diagram

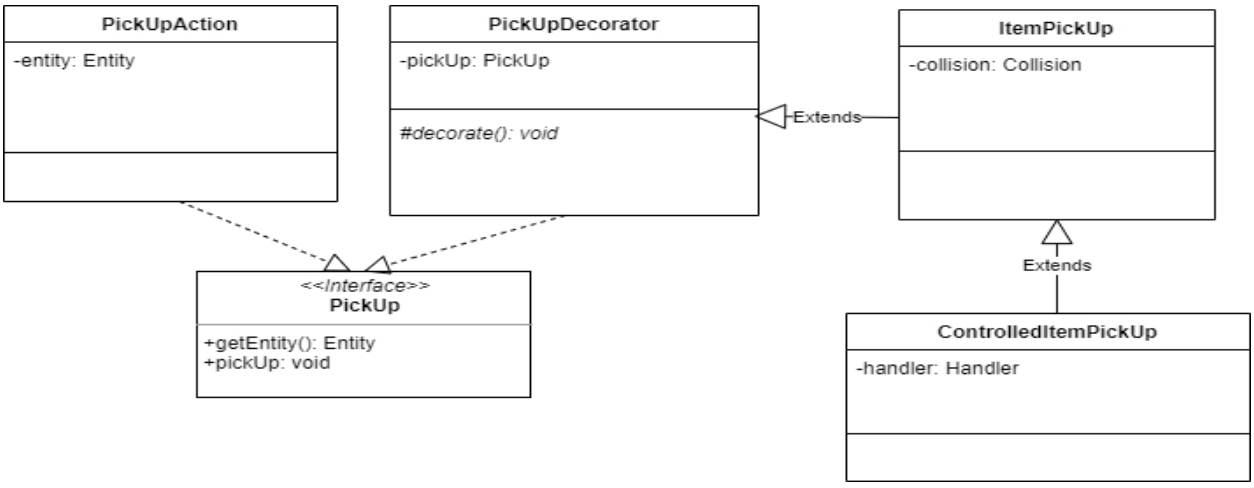

Figure 7.5. PickUp Diagram

Besides the components, we also rearrange our source code by creating various supportive classes with the purpose of managing the game data as well as a great number of attributes easily.

- *Package asset:* Initialize the process of getting and loading images
  - Asset: Categorized various types of images by binding them with name along with the String path in order to get the images as sprite sheets
  - AssetMap: Modify the String data of images
- Package config: Evaluate the configuration of various elements
  - *AppConfig:* Setting default Game characteristics
  - *GameConfig:* Setting default Game configurations
  - *KeyConfig:* Setting default keys
  - *MonsterConfig:* Setting default attributes
  - *PlayerConfig:* Setting default attributes
- Package factories: We applied the **Factory Method Design Pattern** which is capable of enhancing the logic processing and defining relevant objects where the client needed. Some entities are divided into small packages and owning only one class:
  - *PlayerFactory*
  - *BlockFactoty*
  - *ObstacleFactory*
  - *TrapFactory*
  - *MonsterFactory*
  - *TileFactory*

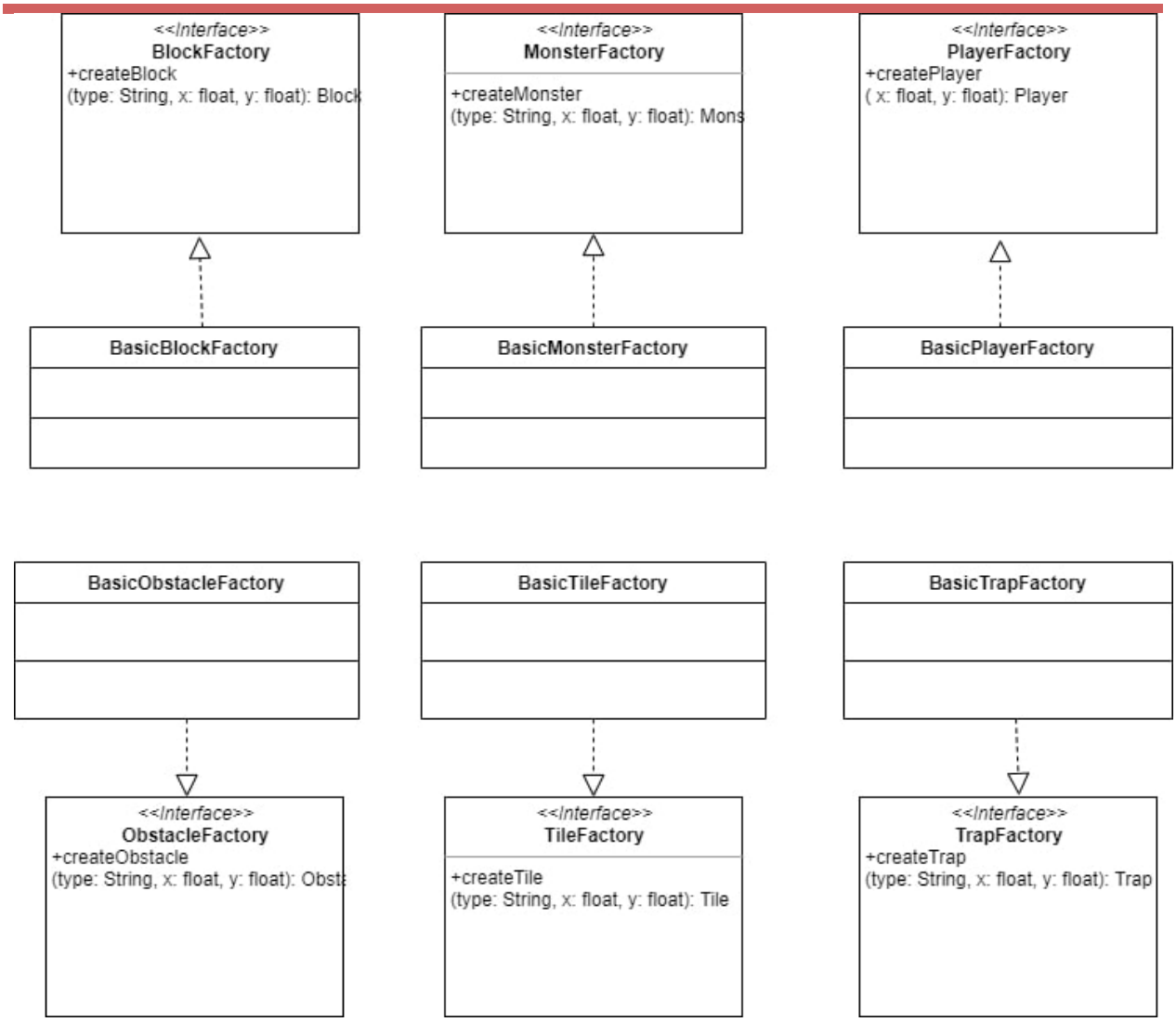

Figure 8. Factories Diagram

- *Package helper: Contains several supportive methods and allows them to be called everywhere in the project with the purpose of optimizing the codebase and preventing repeated code sentences.*
  - *Event and Listener: For every extraordinary action that happens, we consider it is an event and it is initialized inside the* ***Listener Class.*** Whenever an event is called, an action is going to take place.
  - *File:* Managing all the string path of the files
  - *Sheet: Responsibility for cropping images of sprite sheets*

- ❏ *Package routes: Redirect the client view based on the controller's requests, "Menu", "Game", "PauseMenu", "Character"*
- ❏ *Package core: Where the game is first initialized as well as at the stage of data-preparing. We also applied the **Singleton Design Pattern** in the two class Handler and Router in order to create only one unique instance in the whole game processing.*
  - ❏ *Map: Rendering and Loading the map based on which phases are completed by the players.*
  - ❏ *Window: Applying the JFrame as well as Canvas so as to draw and display the screen.*
  - ❏ *Game: Run the game, render and update its graphics, images every second.*
  - ❏ *Router: Direct the views based on the controller requests and register via **RouterRegistration** class*
  - ❏ *Handler: Basically, this is a class created for launching the game with all the data being prepared including images, sounds, window, and maps. This is also where all the services registered such as mouse, cameras and keyboard.*

Finally, the last diagrams describe the most challenging part in our project, the UI of the game including two interfaces: Home Screen and CharacterSelection Screen. We implemented an abstract class called **Element** storing numerous variables and methods. This class implements two other interface classes called **constracts.** These classes determine the status of the UI based on the users' mouse, so we treat it as the main characteristic of each UI segment.

- ● Listenable: Features of some UI elements based on the mouse position
  - ○ *isDisable()*

- *isClicked*
- *isHovering()*
- *onHover()*
- *onClick()*
- *onWaiting()*

- Sharable: Special feature for the Character Selection Buttons, as it will receive the character data and display to the screen.
  - *receive()*
  - *shareWith()*
  - *getSharedElement()*

By that, we also created an abstract class **Button** as a subclass of **Element** then we constructed various concrete subclasses representing the users' selection on the screen.

- *PlayButton:* Move to the CharacterSelection Screen
- *QuitButton:* Leave the game
- *PauseButton: Pause the game*
- *ContinueButton: Continue the game*
- *ReturnButton*: Return to the Home Screen
- *StartButton:* Begin the game

In the CharacterSelection Screen, we initialized the **Radio** abstract class as a subclass of **Element** and the **CharacterSelectionRadio** abstract class extending from **Radio** so as to recognize the user's selection of each player depending upon the six blocks. In other words, as long as a player is selected, there will be unique effects in the border of the character's block as well as the appearance of the character's image. So, we also create a **tick()** and **render()** method for assisting this mechanism.

- *GokuRadio*

- *KidRadio*
- *KiritoRadio*
- *MonkRadio*
- *SatoshiRadio*
- *ShadowRadio*

Besides, we also have a **Text** abstract class extending from **Element** with the purpose of making our UI design more detailed by adding text to it. And there will be some subclasses representing each part.

- *CharacterName*
- *CharacterSelection*
- *MenuTitle*
- *HealthStatus*

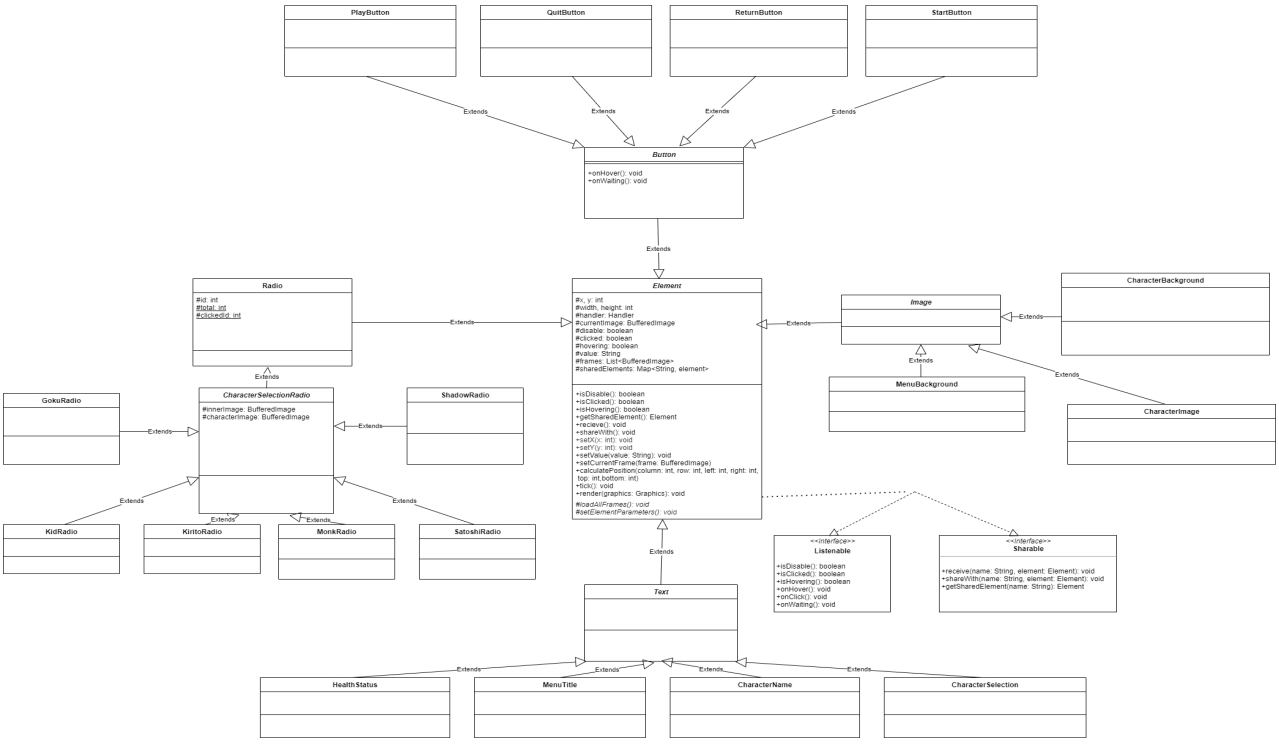

Figure 9. UI Diagram

V. Dijkstra Algorithm and its Application

The Dijkstra algorithm is fully implemented in our project as a special mechanism with a combination of monsters and explosions features.

Idea:

As we want the boss fight must be challenging as well as more difficult, so we decided to add four more monsters with incredible characteristics and only one of them to be summoned randomly on the map by the Boss at the final stage. These creatures will figure out the shortest path to the characters' position with the purpose of causing an explosion in midrange dealing tons of damage to the players. Moreover, after 5 seconds spawning on the map or suddenly being touched by the players, the explosion is also activated.

This is how the path-finding mechanism works on these unique monsters. Players must be aware of these threats as they will track the players' position to chase after in the shortest path before causing an explosion.

Implementation:

First, the monsters with the ability to automatically explode will own some attributes same as the bomb. As it is summoned by the boss during the boss fight so its operation just the same as the bomb whenever appears. The only difference is that it is movable and able to explode if detects a collision between them and players.

Secondly, the Dijkstra Algorithm, known as shortest path finding algorithm, basically starts at the node that you choose (the position of the summoning) and it analyzes the graph to find the shortest path between that node and all the other nodes (players' position on the map) in the graph. The algorithm keeps track of the currently known shortest distance from each node to the source node and it updates these values if it finds a shorter path.

Once the algorithm has found the shortest path between the source node and another node, that node is marked as "visited" and added to the path. The process continues until all the nodes in the graph have been added to the path. This way, we have a path that connects the source node to all other nodes following the shortest path possible to reach

each node.

The process continues until all the nodes in the graph have been added to the path. This way, we have a path that connects the source node to all other nodes following the shortest path possible to reach each node.Basically, the node traversal for this algorithm is circular if we take a look at the visualization of the Dijkstra Algorithm. So, if the players' position changes overtime, it does not matter for the monster to track the shortest path in order to chase after.

VI. Design Patterns/ MVC/ Principles

The following design patterns and principles are applied in this project, an explanation for their strengths is in Section V.

Fully implemented patterns:

- Factory Method Design Pattern
- Decorator Design Pattern
- Singleton Design Pattern
- Prototype Design Pattern

We also follow the *Model-View-Controller(MVC)* which is known as an architectural pattern to optimize our ability of handling clean code so as to get used with creating scalable and extensible projects. As applying the MVC structure, we try our best to approach the first five principles of Object-Oriented Design, *SOLID*, which stands for:

- Single responsibility (S)
- Open-Closed principle (O)
- Liskov substitution principle (L)
- Interface segregation principle (I)
- Dependency inversion principle (D)

VII. Evaluation

Overall evaluation

The project makes great employments of fundamental Object-Oriented programming concepts such as inheritance, interfaces, abstract classes to specify the relationship of components in the game. By splitting the project into several layers of abstraction using Strategy design pattern, it is possible to preserve the minimal improvement required for any potential mechanism/optimization as well as consistency of the game mechanism, dependencies, and resources. However, the lack of a shared set of standards is also reflected in the code and remains a concern as the code base broadens.

Pros:

- MVC
  - Separation of concerns
  - Reusability
- Singleton and Factory Method Strategy
  - Instance control
  - Flexibility
- Decorator Strategy
  - Flexible Alternative
  - Reusability
- Interface and abstract
  - Polymorphism and encapsulation

Cons:

- Redundant attributes and methods in Entity class

VIII. The perspective for future potential renovation

The following checkpoints are mentioned in critical order.

1. Update new features and mechanisms

Well, there will be lots of mechanisms to add to our game, so I just mention three main points that needed to be completed as soon as possible. The first task we need to do it right away is to build up the Player vs AI system in order to support the go-online purpose in the future. Secondly, new bosses, monsters, powerups, and maps will be added to the game and the most important thing is that the new crowd control effects impact the players so as to make the game tougher and more challenging. We are also working on the team-up system to promote collaboration in our game. Finally, the most essential feature in every game, the Quest system. We think that It will be extremely fascinating to receive and do the quest for earning experience to level up or gold.

2. Reinforcement of Game Graphics and other Gameplay features

Firstly, we desire to make improvements to the game Graphics in the future. To be specific, there will be a big update about the graphics design of all entities including characters, items, monsters, maps, obstacles, and UI. We plan to improve the game in order to increase its flexibility as well as smooth motions. Moreover, we recognized that our game has not run fluently on all platforms yet so there should be a small update about the game stability with the purpose of improving general frame-rate performance through optimizations made to multiple systems and content

Secondly, about the gameplay, we suddenly come up with the idea of the item adjustment which modifies the whole item mechanism. In other words, the drop-rate item will operate parallelly with the new item rate called "Rare rate" which categorized every single item separately based on rarity, "Common", "Epic", "Unique", "Mythic", and of course, we will add more items.

Furthermore, we also consider inventing a system that allows interactions between players and obstacles on the map. The first idea in our mind is that obstacles are no longer solid anymore, instead, there will be mechanisms that allow them to respond to players, bombs, items, and monsters.

Finally, about the monster and bosses, we perceived that our monsters and bosses are not that threatening, and it is a brilliant idea to improve their ability instead of giving them more health. So, we decide to recreate the monsters and give each of them a single unique ability, stat, or skill such as the ability to track players on the map and attack them actively. Monsters and bosses in the future are going to be extremely smart and they will be able to pick up items to use against players.

3. Approaching to an Online Multiplayer Game

As I said in the first key point, this game must have the Player vs Player mode in the future although this is the hardest part and time-consuming as well. It is difficult to say but we have not come up with any ideas for creating this mode, however, this will always be an on-going project and we will not give up. The first task that we plan to do is absorbing more knowledge and creating a to-do list for enhancing our codebase, graphic design, and

mechanisms. All of this must be done with the purpose of getting ready for the go-online mode, which is an extremely problematic task for us in the long-term.