ASSIGNMENT 1&2

Analysis and design

Team 1:

An Khoa Nguyen

Adrian Ching Cheng

Dongzheng Wu

Link our git: https://git.infotech.monash.edu/fit2099s221/CL_Lab1Team1/project

Note that:

- **The red font is the modified version of our UML after implementing Assignment 2**
- **The black font is the old one from Assignment 1**
- **The yellow highlighted font is the thing that should change**

Requirement 1: Player and Estus Flask

Because one player can hold only one Broadsword so the relationship between Player and Broadsword should be an association. (change from association to dependency, because we have figured out that our game should have a function to give a player a broadsword so that it will be more scalable and changeable if the player wants to drop down a broadsword or do anything else with the broadsword, instead of keeping broadsword as an attribute of the player which is so tight to be changeable later on)~~To be more specific, we treat the Broadsword as an attribute of the Player class that the Player must have when starting the game~~
~~(Broadsword will be created inside the constructor of Player class ---> Association)~~

(Also, we should do the same thing with Estus Flask, therefore, the relationship between Player and Estus Flask should be changed to dependency)
~~For Estus Flask, because each player has only one Estus Flask, we treat it as an attribute for the player.To be more specific, we treat the Estus Flask as an attribute of the Player class that the Player must have when starting the game and therefore, the multiplicity should be 1 to 1~~. Also, the Estus Flask should be inherited from the Abstract Item class (@see engine package). Because it might need to implement some methods that the Item Class has. Moreover, we choose to not inherit the Estus Flask from the PortableItem because we have looked through all of the requirements and it never said there is the time that the player might drop the Estus Flask. So Item class should be the best choice for the Estus Flask to be inherited.

Requirement 2: Bonfire

After looking through all of the code, we have figured out that our given code is already made by the floors and the walls for the Bonfire (@see Application.java), therefore when designing the class Bonfire, we don't need to inherit it from Wall and Floor to build it anymore, because from now, Bonfire Class is just only drawing a letter 'B' in the game map. ~~As a result, the Class Bonfire only needs to interact with the class Game map. And we treat them as an association because it is obvious that Bonfire is the child that is drawn inside the Game map~~.(After implementing Assignment 2, we have figured out that there is no need to have a relationship between Bonfire and the GameMap, therefore, Reduce Dependency and Follow the rules "Low Coupling and High Cohesion, Why is there more high cohesion ? Because we have decided to have a new relationship between Player and Bonfire that is an association, that means the Bonfire is always attached with the Player at the Start of the Game) ~~Therefore, Bonfire should be an attribute of the game map. Also, we treat the relationship between Player and Bonfire as a dependency because at the start of the game, the Player will implement the method that locate them to the Bonfire, which means the Bonfire should be treated as an instance that will give the information of their position to the Player to be located at the beginning of the game. And the Bonfire will have an association relationship with the Game map.~~ And the Bonfire also has an action that is “Rest at Firelink Shrine bonfire”, so there is the dependency between Bonfire and Rest.

For the “RESET” features, when the player chooses to rest, the Rest Class (is a type of action so we inherit from Abstract Action class) will implement the ResetManager that will (send a message to the Resettable Interface that will run all of the reset Instance that had provided before in each class that need to be reset later ! Because we need to provide the reset Instance in each class before, so each class should implement the Resettable Interface which are the Undead, Skeleton, Player, and Estus Flask. Also there will be an association between ResetManager and Resettable Interface, because the ResetManager need to have a LIST of resettable instances [any classes that implements Resettable] So now, we deleted the relationship between Rest and Player because the ResetManager will be the one that take responsibility to reset the Player) reset all of the enemies position, health and skill and for the Undead Class, the Reset Manager will remove them from the map and refill Player’s health and the Estus Flask. Therefore, there is the dependency relationship between Reset Manager point to Player, Estus Flask, and all types of enemies class.

Requirement 3: Souls (a.k.a Money)

When looking thoroughly at the code, we recognize that we already have the Attack Action Class, therefore, we just reuse that class for reducing duplicate code. As a result, we treat it as a kill/slay enemies method. As there are three types of enemies (@see enemies class) that can be killed by the Player , there will be three dependencies from Attack Action Class point to them. Also each type of enemy will have a specific soul reward, therefore, each type of enemies will correspond to specific souls reward that the player will gain after killing them. (We have figured out that we need to replace the SoulsReward by SoulsManager [reuse the given class, reduce duplicate code] to handle the transfer souls message such as deleting and adding souls. And there will be a dependency between AttackAction and Soul Interface. Because each time a player defeats one enemy, AttackAction will call the Soul transfer message then the Soul of the enemies will transfer to the player. Also Player should have a SoulsManager as an attribute to receive these messages)~~Therefore, when the enemies are killed, it will inform the Souls Rewards Class to increase the number of souls to the Player. As a result, when the Souls Rewards Class gets the information that there are specific enemies that have just been killed, they will increase the specific number of souls to the Player. Therefore, it should be a dependency relationship from Enemies to Souls Rewards Class and from Souls Rewards Class to Player~~. Also, the ~~Souls Rewards~~ SoulsManager is a type of Soul so it should implement the Soul Class Interface. So there will be an Implementation/Realisation relationship between Souls Rewards Class and Soul class.

Requirement 4: Enemies

In order to prevent the enemy from entering the floor (and valley), we need to add a method in Floor class to keep the enemy out of it.(I used Abilities like ENTER and FALL to forbid actors from entering the floor and valley except the player, and the checking method will be implemented in each terrains' class.) We will use HOSTILE_TO_ENEMY status to distinguish the target so that enemies won't attack each other. (In addition, I provide BOSS status so that it can be distinguished from other enemies so that it can have special features like displaying dead messages or weak to storm ruler. Those abilities/status can reduce the dependencies by avoiding extra checking between objects)

To follow the DRY principle, an abstract Enemies class was created which is extended from the Actor class. For different enemies, some of them have the same behaviours like attack and follow the player, walk around, (wander behaviour is done in skeleton and undead class instead of abstract Enemies class to avoid providing the lord of cinder wander behaviour.) active skills randomly, and these can be created by implementing from the behaviour interface. The enemies should have multiple behaviours as attributes. (To be more specific, souls(souls manager), souls transferred method, soft reset method like register instance/ reset instance are all done in Enemies class because each enemy should use those attributes and methods.)
In this way, those behaviours (attributes, and methods) can be reused for different enemies by accessing (Extends) from abstract enemies class.
However the UniqueBehaviour will only belong to the LordOfCinder class since it is the unique behaviour of Boss. This should be implemented based on Boss's weapon, hitpoints, and locations (I thought the skills needed to be done in this class, but that should be provided from the weapon actually.) to achieve the required feature. The attack behaviour should be implemented based on AttackAction.

The SoulsReward class is implemented from Souls interface which will be used to gain souls after the enemy is defeated. (I didn't create souls reward class, instead, enemies will be implemented from Souls interface and that allows players to gain souls after defeating enemies by calling souls transferred method (handled by souls manager *see below*). These operations will be done after the enemies truly die.

In order to follow the Single Responsibility Principle, I created a SoulsManager class to store and handle the number of souls for players, enemies and token of souls. All the objects that have souls need to be implemented from Souls and use the specific method provided. Also, upcasting will be used when transferring souls. This kind of encapsulation avoids exposing private attributes and keeps data safe. To follow the same principle above, all the features of enemies are done in separate behaviour classes to avoid adding too many codes directly on each enemy class.)

After an enemy is defeated, it should be removed from the game map by calling a removeActor method from GameMap class. (This will be done in attack action or play turn accordingly. If a player has been killed, it will not be removed, instead, the game will be soft rest *see requirement 6.*)

All the particular attributes like hit points / maximum hit points, reward souls, locations, and some individual methods like revive, automatic death, and spawn from cemetery(Revive, instant die will be implement as behaviour, and spawn from cemetery is done in cemetery, in addition, the weapon will be created and added into the inventory, so it is not an attribute for enemies, should be dependency relation instead.) will be built in child class respectively since they are not going to share those values and functions.

To answer the question from base codes: According to our design, the LordOfCinder class cannot be an abstract class, because once it becomes an abstract class it cannot be initialized.

(All enemies' features are tested, some features would need to be based on the weapons so I created several dummy weapons for testing. All features are passed tests with those dummy weapon (MagicWand & HeavySword) Gun is only used to kill enemies)

Requirement 5: Terrains

Valley and Cemetery are part of terrains so they are extended from Ground abstract class. Once the player walks into the valley they should be killed immediately, therefore valley class should call the hurt method in play class which represents a dependency relation with Player class. Undead will be created around the cemetery, so it can be an attribute of the cemetery (I thought that we need to store the undead objects in cemetery before, but after implements the codes cemetery just create the undead object and place them on the map, so the relation should be dependency instead of association, and the undead will not become the attribute of cemetery.)

Requirement 6: Soft reset/ Dying in the game

(I used the static factory method for soft reset, in this way it doesn't need to be initialized when using it and we can use soft reset by calling reset manager anywhere in the program because the static attributes will share the same attributes value with each reset manager instance. This could be used when players rest at the bonfire as well.)
Playturn method should check if the player is conscious or not, once the player becomes unconscious, the soft reset will be triggered. And the soft reset should heal the hitpoints of the player to maximum, and place the player on to the bonfire. (The reset feature/functionality is done in every resettable objects' class either in playturn / tick or just in the reset instance method. Therefore, the SoftReset class is unnecessary.) It also needs to call the reset manager to reset all the resettable stuff. Those approaches would be considered using methods so they will be presented as dependencies.
Once the player is dead, the souls will be transferred to token souls which is implemented from souls interface and be placed where the player is defeated.

There is a special situation where the player dies from falling in the valley, in this case, the soft reset should check the last action of the player and use the last location of the player to determine the location step behind then put the token of souls on it. (My old design was checking the location of players each turn and once found there was a valley around, then placed a token of souls. New method could be much easier and more efficient. Instead of checking location each turn, now we only need to check where the player is dead, if there is a valley then we take one step behind and place the token of souls.)
Token souls will be extended from item class (extends from item class and implements from souls interface) so that we can use those methods like setting the capabilities, and I might use a special method to allow players to pick it instead of making it portable because it cannot be dropped except from death.

Requirement 7: Weapons

For the weapons, there are in total 4 weapons in this stage of the game which are broadsword, giant Axe, storm ruler and the Yhorm's great machete. When I was designing this part, I decided to categorize it into Axe and Sword based on the weapon type. In this case there will be 2 more abstract classes extending from the weapon item. The swords will be inherited from the sword abstract class and same situation for the Axe class. We would like to implement the burning ground in the Axe class so as to reduce redundancy of the code
About the active and passive skills, it would be stored into each weapon object as the skills are unique, therefore it will be better to store within the object.
For not dropping weapons when the player died and did something intentionally, we would be setting up methods so as to save the weapon that each actor has and when they died , we would call this method and keep the weapon when the actor respawns.

For the weapon class, I decided to create a weapon package which includes 4 weapons used in the game. As mentioned in the previous design rationale, we categorised the weapon into axe and sword class so as to inherit some common attributes and functions so as to simplify the whole code. The axe and sword class will be inherit from melee weapon in which melee weapon is inherit from weaponItem class.

For the Axe class, there isn't any common attributes and functionality to put in yet.

But for the sword class, as every sword in this development stage has this passive skill of critical strike, therefore I implemented the method in the sword class which could be override by both swords.

Broad Sword (Extend from sword): For this sword ,I can set all the values for this sword as it is fixed. The only method that I need to implement in this class will be the damage() method , as this weapon has the critical strike passive skill, therefore I need to make sure that it will be invoked every time the player attacks an enemy in which damage() method is the one to edit

Giant Axe (Extend from axe): for the giant axe, I first set the values for the giant axe object ,after that I start to develop the active skill which is spin attack . I created a new class called spin attack action for storing the function of the spin attack. as this spin attack action is also applicable for the enemies too.

Storm Ruler(Extend from sword): For the storm ruler, we created a charge action class and a wind slash class for it so that those two classes could be swapped once charges finished an after the wind slash attack is performed.

Yhorm's Giant Machete (Extend from axe):   for the boss weapon, we created two new classes which is ember form action and burning dirt class. The amber from action is inherit from weapon actions and the burning dirt is inherit from ground. burning dirt is responsible for hurting the player and the ember form action is to activate the burning dirt and increasing the hit rate .

Requirement 8: Vendor

For the vendor, we first inherit from the actor as it is an actor. After linking it to the actor class, we start looking into the requirements for the vendor.

1) We need to set the vendor in a permanent location , in this case we link it to the location class and set the X,Y coordinate for it.
2) It could trade with players using souls, in this case we will be creating a trade class and this trade class will be inheriting from the souls class so as to get the methods of adding and subtracting souls from players
3) It swaps the weapon for player when the player buys a new weapon, therefore we need to link to the swap action
4) Increase Max Hp is one of the goods sold by the vendor. When we were looking through the code, we saw that there's a method in each player object which allows him to increase the maximum Hp, as the result, we will be calling this method within the player object when he buys the increased maximum Hp.
5) There is a new class called getPurchasedWeapon, this weapon is to allow the vendor to create and get the weapon which the player purchases so as to complete the trade

For the vendor requirement, I first created a vendor class which inherit actor so as to display the vendor in the map. After that , I created three classes which is buy broadsword , buy giant Axe and increase maximum health function .I decide to let this free functions to be inherit from a class called by item action so that if there's any changes in the purchase function, we would be able to change the by item action class but not changing it one by one. By item action classes inherit from action which allowed us to inherit an override different methods .