Rationale FIT2099 Assignment 1

Unified Modified Language(UML)

TokenOfSouls (new class)

A new class named TokenOfSouls will be created which will represent the object dropped when the player dies. The TokenOfSouls object should hold the amount of souls the player had prior to dying and be dropped at the exact location where the player dies. In which case the player falls into a valley and dies, the token of souls would spawn the player's last location before he fell into the valley. TokenOfSouls inherits Item Abstract class because it is considered an item therefore it needs to inherit from the Item Abstract class to access methods such as getPickUpItemAction for the player to pick up the token of souls.

This class should implement the interface class Soul so that it is able to store the player's soul inside once the player dies. This means that the player should also implement the interface class Soul to track its current number of collective souls so that the Souls can be transferred to the TokenOfSoul once the player dies. The token of soul should consist of static instance variables and static methods because there will only be one token of souls existing in the game per instance, but the variables can not be final because the soul will be different each time the player dies. This means that the player can use the transfer soul method to transfer its soul from the player to the token of soul without having to create an instance of token of soul. The player should also be associated with the TokenOfSoul

FireKeeper (new class)

The fire keeper will be a new class that extends the abstract class Actor since the fire keeper is a vendor that is an NPC (non player character).Since it is an NPC, it would need to use the abstract methods in the Actor class.

It should be associated with Player, with the fire keeper having a one to one relationship with Player. The firekeeper would have an instance of Player because he needs to access the player's current soul to check if Player meets the requirement to make a purchase, the fire keeper also needs access to the Player attributes such as max HP to increment his max hp.

The fire keeper does not need to be associated with the classes of the weapon he is selling such as BroadSword and GiantAxe (the weapons he is selling), because these objects should contain static methods that does not require an instantiation of the class so that the Player and fire keeper does not need to hold instances of all the weapons. If the methods are not static then all the weapons would need an association with the fire keeper or the Player (more explanation on new weapon classes in Weapons section).

The Player would need to implement the Soul class to keep track of the souls collected so that it can purchase items from the fire keeper if the soul requirements are met. For example a broadsword costs 500 souls, the player would request to buy the weapon from the fire keeper, and in the FireKeeper class it will check if it has enough souls, it will return the weapon object if it returns true and false if otherwise.

BuyAction (new class)
A new class would also be created, BuyAction. The BuyAction class would be associated with FireKeeper because it would need an instance of the FireKeeper. In the FireKeeper class there would be a method that gets the buy action , returns a BuyAction type and stores it as an allowable action so that in the console when the player is near the FireKeeper, there would be a buy action available. The BuyAction class should inherit the Action Abstract class because it is considered an action and would require the methods implemented in the Action class. In turn, the FireKeeper would also need to be associated with the Actions class so that the getBuyAction method can be added to the list of actions which would then be available in the allowable actions. The BuyAction class is needed because otherwise the Player can't interact with the vendor.

Lastly, the FireKeeper would be associated with the Location class. The fire keeper is a vendor that will be in a constant spot of the map the entire game without moving. Therefore, it does not need to be a resettable instance but it still requires Location to store its coordinates and place its display character (F) in the map. The coordinates of fire keeper can be declared and initialized in ActorLocations instead but we decided to declare its Location in all of the actor classes because there would be multiple instances of an Actor, say like Undeads, then we would be able to create multiple instances of Undead with different Locations for each, which for continuity sake we also put Location in fire keeper.

Weapons
BroadSword, GiantAxe, StormRuler, GreatMachete (new classes)

All the weapons in the game would require new classes of their own. This is because each weapon is unique in its own ways and although it has some similar attributes such as hit rate and damage, they all have their own personalised abilities. With that said, all weapon classes should inherit from the class of MeleeWeapon which extends WeaponItem, which extends Item and gives the weapons properties of an Item.

All the weapons should be associated with the Actions class, because in each of the weapon classes, there would be methods to get their abilities action. For example, StormRuler would have a getWindSlash method. So then the getWindSlash method needs to be added to the list of actions so that it would appear in the list of allowable actions. There is an exception for the weapon StormRuler because it is available to be picked up on the map itself, which requires player interaction. Therefore StormRuler would be associated with Location to store its fixed coordinates on the map.

Abilities
WindSlash, Charge, SpinAttack, EmberForm (new classes)

Back to the abilities of the weapon, we would require new classes for each of the abilities that each weapon has. This is the same concept as PickUpItemAction and Item. The abilities would be like the PickUpItemAction and the weapons would be like the Item. So there would be 4 new classes namely EmberForm, WindSlash, Charge and SpinAttack. In each of their respective weapon classes, there would be a method that gets these abilities and returns the class of the ability. For the example of StormRuler, there would be a getWindSlash method that returns a type WindSlash and adds this getWindSlash method to the actions list so that it would be seen as an allowable action. But to add the getWindSlash method to the actions list, it would first need the WindSlash class to be of type Action if not it can not be added. Therefore, all of the abilities classes should inherit the given WeaponAction class in the engine code which in turn inherits Action abstract class. In each of the weapon abilities classes, they would be implementing the active abilities while the passive abilities can be done within the weapon class since the passive abilities are considerably the properties of the weapon and needs no action but active abilities on the other hand is an action and requires the user to choose to use the ability or not.

Player (Existing class)

The Valley class is directly associated with the Player class because we need to know the location of which valley the player drops into for the TokenOfSoul class. The player class needs to implement a soul interaction class to change the soul value content within the player when the player defeats a NPC or Non Playable character. The player attributes also need to implement a Resettable interface class so the player's attributes can be reseted.

BonFire (new class)

BonFire class is a location where the player will spawn in the map under three conditions. The first condition is where the player has died either the player's hp become zero, the player falls into the valley or the player is resting at the bonfire.In this case,Bonfire class is directly associated with ResetManager class with a one to one relationship where the BonFire class has the object of ResetManager where we can reset the location of player with it. In order to reset the player's location, the BonFire class is initialised with a fixed location in the map and the player's location will be reinitialised with the BonFire's location in BonFire class.

Furthermore, the Player class implements Resettable interface class where player instances are created in the Resettable interface class to change player attributes. Resettable interface class and ResetManger class have a dependency relationship where ResetManager class iterates the list of resettables when the game undergoes soft reset. BonFire class is directly associated with Actions class with one to one relationship because whenever the player chooses to rest in the BonFire the action list will be iterated where the RestAction method will be executed. Hence, Actions class is directly associated with Action abstract class to iterate the list of actions in Actions class.

Moreover, the Non Playable character or NPC such as LordOfCinder and Skeleton is reset to its initial state in the game once the BonFire class is executed except a condition where LordOfCinder will not respawn if it has already been defeated by the player. In order to do so, the BonFire class is directly associated with the ActorLocation class. In the ActorLocation class there will be an actor and location object that is able to access the current location of each NPC when the BonFire class is executed. We need the current location of each NPC because the symbol of each NPC shown in the game map has to be reset into its original map's floor symbol. Furthermore, the BonFire class is inherited from the Ground abstract class because the BonFire class is considered to be a type of ground. We can access Location class to find each individual ground location, thus the Location class is directly associated with Ground abstract class.

RestAction (new class)

A RestAction class is created where the player decides to rest at the BonFire and the game will undergo soft reset. Hence,the RestAction class is directly associated with the BonFire class with a one to one relationship where the RestAction class has the object of BonFire that reset the whole game through ResetManager class. RestAction class is inherited from Action abstract class because RestAction class needs to inherit functionality from Action abstract class so the player can choose to rest in the BonFire when the player is near to the BonFire's location.

EstusFlask (new class)

The EstusFlask class is inherited from the Item abstract class where EstusFlask is one of the item in the game.The Player class is directly associated with the EstusFlask Class where the Player class has an EstusFlask object that shows the number of EstusFlask is left within the player. For the player to use the EstusFlask the EstusFlask class needs to be directly associated with the Actions class where it can execute the method of DrinkItemAction in the Actions class.

DrinkItemAction (new class)

A DrinkItemAction class is inherited from the Action class where the player will use the EstusFlask once a specific hotkey is identified in the Action class. The DrinkItemAction class is then directly associated with the EstusFlask class to reduce the number of EstusFlask left within the player followed by the player's hp will increase so DrinkItemAction class is directly associated with Player class. The player's hp will increase whenever the DrinkItemAction class is executed.

Cemetery (new class)

Cemetery is a newly added class which inherits Ground abstract class because cemetery is a type of terrain so we need the constructor of Ground for its unique display character and we need its methods to add, remove and check the capability of cemetery.

Cemetery and Undead has a directed association relationship, and it's directed from Cemetery to Undead because Cemetery is a place to spawn undeads, one cemetery can have one or more undeads. Cemetery implements Resettable. The idea behind this is,we create an instance variable of ArrayList in Cemetery to store the Undeads created,and if an undead is dead,it's removed from the ArrayList. Therefore,when the player is dead or rests, the instance variable of Cemetery is reset, hence the ArrayList becomes empty hence all the undeads are removed from the map.

Cemetery has Location, the relationship between these 2 classes is association. Cemetery needs to create an object of Location to implement its method in order to set its location.

Undead (existing class)

Undead class also implements Soul interface class,so that the soul of undead can be added to the player who kills it.

Furthermore,undeads need to wander around the map after it spawn from cemetery so Undead class needs to have WanderBehaviour so that Undead can implement the method of getAction which returns a MoveAction for undeads to wander to a random location if possible.The relationship between these 2 classes are association,directed from Undead to WanderBehaviour.Undeads also need to follow the player until they die when they detect the player so Undead Class has FollowBehaviour.The relationship between these 2 classes are associated,directed from Undead to FollowBehaviour so that inside Undead class,we can create FollowBehaviour object and implement the getAction method for Undead to have the follow behaviour towards Player.

Undead class has Actions class,the relationship between them is association,directed from Undead class to Actions class.It is because we have to create an object of Action class,and implement its method to append the AttackAction,and the Action returned by the getAction method inside WanderBehavior class and FollowBehavior class so that all the actions can be added to the ArrayList actions,so that all the actions of Undead has will be available in allowable actions.AttackAction is necessary to be added to the allowable actions so that undeads can be attacked by the player.

Skeleton (new class)

Skeleton is a newly added class that inherits the Actor abstract class as skeleton is a type of actor and we need to use the attributes and methods of Actor class.

Skeleton has Location, the relationship between these 2 classes is association. Skeleton needs to create an object of Location to implement its method in order to set its location so that when the world resets, the skeleton can be placed back to its initial location.

Skeleton class also implements Soul interface class, so that the soul of the skeleton can be added to the player who kills it. Skeleton class implements Resettable interface class, so that skeleton can reset its abilities and attributes such as health and position when the player dies or rest.

Furthermore, skeleton will walk around after it spawn Skeleton class needs to have WanderBehaviour so that Skeleton can implement the method of getAction which returns a MoveAction for undeads to wander to a random location if possible.The relationship between these 2 classes are association,directed from Skeleton to WanderBehaviour. Skeletons also need to follow the player until they die when they detect the player so Skeleton Class has FollowBehaviour.The relationship between these 2 classes are associated,directed from Skeleton to FollowBehaviour so that inside Skeleton class,we can create FollowBehaviour object and implement the getAction method for Undead to have the follow behaviour towards Player. Skeleton class has Actions class,the relationship between them is association,directed from Skeleton class to Actions class.It is because we have to create an object of Action class,and implement its method to append the AttackAction,and the Action returned by the getAction method inside WanderBehavior class and FollowBehavior class so that all the actions can be added to the ArrayList actions,so that all the actions Skeleton has will be available in allowable actions.AttackAction is necessary to be added to the allowable actions so that skeleton can be attacked by the player.

Besides,Broadsword and GiantAxe are two newly created classes.**(More details explanation is provided above)**.They both inherit MeleeWeapon class since they both are melee weapon type and cannot be dropped.New classes of Broadsword and GiantAxe are needed as each weapon has unique attributes and skills,so it would be more manageable to have new classes for each of the weapon. Skeleton has Broadsword and GiantAxe class since skeleton carries either broadsword or giant axe randomly.The relationship between Skeleton and Broadsword is association directed from Skeleton to Broadsword. The relationship between Skeleton and GiantAxe is an association, directed from Skeleton to GiantAxe.

LordOfCinder (New class)

LordOfCinder is a newly added class that will inherit the abstract Actor class as Lord of Cinders because it is considered a ‘boss’ in the game, which is also considered an NPC (Non-player character). Since it is an actor, it will need to use the abstract methods in the Actor class.

LordOfCinder would be associated with GreatMachete (new class) which is the weapon LordOfCinder will be yielding. LordOfCinder will have one instance of GreatMachete and the GreatMachete is not droppable. The GreatMachete class extends from the class MeleeWeapon which in turn extends from the WeaponItem abstract class. GreatMachete needs to extend MeleeWeapon because it needs to initialise it’s attributes such as damage, hit rate, display character and its verb (hit, zap etc). A new class GreatMachete is created because all weapons in the game are different and unique, in a sense that all weapons have their own abilities (both active and passive) therefore each weapon has its own class.

LordOfCinder is directly associated with the Location class to initialise its original location. Meaning that it serves as a marker to remember the starting location of LordOfCinder. Everytime the player dies or rests, the LordOfCinder will be reset to its original location. Assuming that the player kills the LordOfCinder then LordOfCinder will be deleted from the game and does not respawn anymore. Every time LordOfCinder moves, its coordinates will be updated in the Bonfire ActorLocation variable. Location object is also needed to update the ground display character such as if LordOfCinder moves north, then the ground north of LordOfCinder will be changed to the character (Y) and the previous ground he was on will be changed to its original ground such as dirt with the character (.).

LordOfCinder has PortableItem,the relationship between LordOfCinder and PortableItem is association,directed from LordOfCinder to PortableItem.Since LordOfCinder will drop Cinders of a lord after he is killed,so an object of PortableItem cindersOfALord will be created by using the constructor in LordOfCinder.It has to be portable since it can be dropped from cindersOfALord and can be picked up by the player.

The player will receive 5000 souls if the LordOfCinder is defeated by him. Thus, LordOfCinder has to implement the method in the Soul interface class to change the soul value in LordOfCinder and player class. LordOfCinder will be reset if the player has died fighting the LordOfCinder. The LordOfCinder class has to implement the method in the Resettable interface class to reset the health point and phase of the LordOfCinder. The LordOfCinder will follow the player when the player is near to it. Hence, LordOfCinder class has to be directly associated with FollowBehaviour class so the getActor method can be executed where the LordOfCinder will follow the player’s location. LordOfCinder class has Actions class,the relationship between them is association,directed from LordOfCinder class to Actions class.It is because we have to create an object of Action class,and implement its method to append the AttackAction,and the Action returned by the getAction method inside

FollowBehavior class so that all the actions can be added to the ArrayList actions,so that all the actions LordOfCinder has will be available in allowable actions.AttackAction is necessary to be added to the allowable actions so that lord of cinder can be attacked by the player.

Ember (New class)

Ember is a newly added class which inherits Ground abstract class.Ember is a type of terrain which can only burn the Dirt ground.Ember needs a new class as it has an unique character V which represents it and we need some methods to subtract player's hitpoints when the player stands on the Ember type of terrain Ember has Player,the relationship between Ember and Player is association,directed from Ember to Player.Ember class needs to create a Player object so that we can implement Player class's method to subtract player's hit points when the player stands on the Ember type of terrain.

EmberFormAction (New class)

EmberFormAction is a newly added class which inherits WeaponAction abstract class,it represents the active skill of GreatMachete.We need a new class for this because the active skill of GreatMachete is unique and works differently to other weapons.EmberFormAction has GreatMachete,the relationship between them is association,directed from EmberFormAction to GreatMachete.GreatMachete class has Actions class,their relationship is association,directed from GreatMachete to Actions because we need to create an object of Actions and append the EmberFormAction to the ArrayList of actions so that it will be available in allowable actions.EmberFormAction has Ember,the relationship between Ember and EmberFormAction is association,directed from EmberFormAction to Ember.Because EmberFormAction need to create an object of Ember to implement its method in order to set the Dirt type terrain to Ember when the active skill of GreatMachete is used.

Conclusion

Based on the principles we have learned for Object Oriented Programming, we should not repeat ourselves (**DRY principle**), and try to reduce dependencies as much as possible (**ReD principle**). We were also instructed not to touch/change the engine code and apply the do not overthink principle(**DOT principle**).

Based on our design implementation, there are no dependencies between classes at all as seen in the UML diagrams except for the ones which already exist in the base code provided to us which applied the **ReD principle**. We also never modified/changed the engine code and have only worked with the game code and mostly inherited classes from the engine code. This shows that we tried to reuse existing code as much as possible and avoided repeating the same code which applied the **DRY principle.** Overall we tried not to overthink too much and worry about the coding but just focused on having a rough idea of what it would look like and thinking theoretically if it would work or not which applied the **DOT principle**. All in all, we tried to minimize everything and only connected classes with each other if it was necessary.

<u>Interactive Diagram(Sequence Diagram)</u>

DrinkItemAction.execute()

The execute method of DrinkItemAction is called and it takes in the parameter of the actor (the player) and also the map. The execute method would first call the getEstusFlask() method from the Player class which would in turn return an object of EstusFlask. Next, the execute method would call the isEmpty() method from the EstusFlask class to check the remaining charges of the estus flask. It would return a boolean value to indicate whether the estus flask is empty or not. If the charge is empty then the console will print a message with “No changes remaining”. Other than that, a message will be shown in the console indicating the remaining charges of estus flask within the player. The DrinkItemAction class will call the useCharges() method in the EstusFlask class where the amount of estus flask remaining in the player will be minused by one since the Player would consume a charge. Moreover, the Estus flask class will increase the player’s maximum hp by 40% with the heal method called from the Player class which implements an abstract method from the abstract class Actor. After the healing is done, it will print a message of the remaining charges of Estus Flask along with a description saying the player has been healed.

SpinAttack.execute()

Execute method is called and it takes in the parameters of the actor and map.Then,in the execute method, getWeapon method is called from the actor of type Actor to return the GiantAxe object of actor holding.Then the damage method is called from the object of GiantAxe which returns the damage of giant axe and stored in a variable(damage).Then,verb method is called from the object of GiantAxe which returns String (“zap”) and stored in a variable(verb) so that it can be concatenated to a String and be printed when the enemies got hit.Then, locationOf method is called from the ActorLocation class which returns the location of the actor currently at and stored in a variable(here).

We would then want to check the adjacent squares of the actor calling the execute method to see if the SpinAttack is done with no other actor around or if there are other actors around. The method getExits() would be called on *here* which is the Location of the actor, and returns a list of Exits. For each exits in getExits(), exits.getDestination() will be called to check where the actor will be if it takes that exit. It will be stored in a variable named destination which is of type Location. Then, the method isAnActorAt(destination) would be called to check if there is an actor in the current destination. If there is an actor at the current location, then the method hurt from the target actor would be called to deal the damage done by the SpinAttack, which is 50% of its original damage. After the actor is damaged, it would print a string of description.

The actor acts as a target for the player when the target is unconscious a method called removeActor will remove the target from the game map.A message will print in the console that tells the player the target is dead.However, If the target is not found in the game map player will be notify with a string of description.