UML Class Diagrams

The UML class diagrams for Assignment 3 are separated with Assignment 1 & 2 as it will be really messy to combine both the classes together and it will be hard to read and understand the line properly. Hence, the second class diagram only consists of the new classes created for Assignment 3. There will be another class diagram that contains the classes for Assignment 1 & 2.

FIT2099 Design rationale

Task 1: Going To Town & Bonus Task: Vehicle Battery & Vehicle Key

Information about Bonus Task:
As for now there will be a vehicle in both of the maps that allows the player to move between different maps. Therefore, we are planning to make the game more interesting by asking the player to collect the car key and battery around the map before reaching the vehicle and travel between the maps. In other words, the player will require a key and a battery in order to move between the maps using the vehicle. We will be creating a new Battery class which extends from Item class that will place the battery around the map to allow the player to collect it. We will also have a Key class which extends from Item class that will be placed around the map for the player to pick up and put inside the player's inventory. The player will not be allowed to travel to the other map if it doesn't collect both the vehicle key and vehicle battery when it's standing on the vehicle. The classes below are implemented and will explain what it does to make the task works:

1. Vehicle

Vehicle class will be an extension of the Item class. Although the vehicle is extended from Item class, I have implemented it with portable=false, which does not allow the actor, standing on the vehicle to pick up the item. It will override the getAllowableActions() method in the Item class to check if the player is standing on the vehicle item and the player's inventory contains vehicle key and vehicle battery. Then, it will return an action option that allows the player to move to another map if the player contains both the items. It will then remove both the key and battery item in the player's inventory within the getAllowableAction() method and add the MoveActorAction for the player to move to another map. The MoveActorAction() class is created in the engine class to move the actor between different maps. Therefore, the reason why I chose to implement it this way is because I don't have to

create another class to move the actor to the other map as the MoveActorAction class is already implemented in the engine.

It will remove the vehicle battery and key by calling the method in the player class from the vehicle class. Vehicle class's constructor will have the gameMap and townMap location where the vehicle is placed and also the player as the input parameters. When the player is standing on the vehicle, it will then check if the player's inventory contains the vehicle battery and key before allowing the player to move to another map. Then, it will remove the item using the two methods removeKey() and removeBattery() created in the player class so that the player will not be able to move back to the map after travelling to the map. The player will need to find the key and battery again in the newMap in order to move to another map.

2. Player Class

Player class is extended from the Human Class which is extended from the Actor class inside the engine. Player will implement methods to check if it carries the vehicle key and battery item in it's inventory so that other classes such as vehicle class can use the method to check if the player is holding the item to allow it to move to another map. It will also create the removeBattery() and removeKey() methods in the player class to remove the item so that it can remove the items from vehicle class when the player is standing on the vehicle and contain those two items.

It determines if the player is carrying the vehicle battery and key, remove the battery, key items inside the player class instead of doing it in the vehicle class because it will be easier to extend the game next time if the developer decides to allow other actors to move around the maps as well. As the developer can just put the method in the actor interface to allow other actors to use the method and allow them to move between the maps. This proves that the design is future proof as it can be easily further developed by the developer in the future.

3. Vehicle Battery

The vehicle battery is extended from the Item class which is portable which allows the player to pick up the item. It will add a capability to the vehicle battery item with BATTERY type, so the player can check if the item is a battery later when it's standing at the vehicle. The reason to implement this class is that it will reduce the dependency between vehicle and vehicle battery and key as the vehicle just has to know if the player is holding the battery and key in its inventory. This is because it

has done the check in the player class, so it doesn't have to know about the vehicle key/battery to know the item.

4. Vehicle Key

The vehicle key is extended from the Item class which is portable which allows the player to pick up the item. It will add a capability to the vehicle key item with KEY type, so the player can check if the item is a battery later when it’s standing at the vehicle. The reason to implement this class is that it will reduce the dependency between vehicle and vehicle battery and key as the vehicle just has to know if the player is holding the battery and key in its inventory. This is because it has done the check in the player class, so it doesn't have to know about the vehicle key/battery to know the item.

Task 2: Mambo Marie

Mambo Marie is extended from the ZombieActor class which extends from the Actor class. It’s allowed to spawn 5 zombies in every 10 turns and will disappear from the map after 30 turns. The classes implemented below shows how it makes the task works:

1. GameWorld Class

In this new GameWorld class which extends from the World in the engine class. It will override the run() and processActorTurn() methods. It will then check through all the actors in the actorLocations class inside the GameWorld class. It will see if the mambo marie is on the map. If the mambo marie is on the map, then it will not check to summon the mambo marie. Else, if the mambo marie is not in the map, each turn when the gameMap ticks, it will have 0.05% chance of spawning mambo marie into the map. The reason to implement this class to check if mambo marie is in the map is because gameworld class contains all the actors in the map and will process it’s turn for every actor. This means that it will be easy to not only check if mambo marie is on the map, it can also check if it’s dead/alive too. By extending the GameWorld class will make it easier for developers to implement extra functions in the future without modifying the engine World class. The developer can allow the player to end the game quickly by overriding the run() method in the GameWorld() class.

2. Mambo Marie Class

Mambo Marie is extended from the ZombieActor class which extends from Actor class in the engine. It will override the playturn in the actor class to allow the actor to perform a list of behaviours which executes some actions. At each turn, mambo marie class will go through chantBehaviour and increase its turn by 1 and check if it can spawn any zombies in the map. I have created some methods such as isAlive() in the mambo marie class to check if the mambo marie is alive or just disappear from the map. I also created another method called disappear() to change the mambo marie status from true to false when it disappears from the map. This allows other classes such as GameWorld to check if the mambo marie has disappeared or been killed. The reason to implement mambo marie class is to fulfill the design, Don't repeat yourself as the actor mambo marie can be created by just creating the new instance and also each instance represents a separate mambo marie. The developer can just change or add the behaviours that mambo marie can perform in the future by adding the behaviours into the behaviour list stored inside mambo marie class. This proves that future proofing for the mambo marie class.

3. ChantBehaviour Class

ChantBehaviour class is implemented from the Behaviour class. It overrides the getAction class to check if the actor, mambo marie can spawn any zombies. As mambo marie is allowed to spawn 5 zombies in every 10 turns, and also vanish after 30 turns. This class will check the behaviour of whether the actor can perform such action. The reason that this class is implemented is because it has a don't repeat yourself principle as the developer can reuse the class if it wants to implement a new actor to check for similar behaviours. This proves that why the chant behaviour is not checked in the mambo marie class but instead created a new chantBehaviour class to check the behaviour.

4. SpawnZombieAction Class

SpawnZombieAction is extended from the Action class in the engine. It overrides the execute() method in the Action class to spawn 5 zombies randomly over the edge of the map. There will be another method inside the class to summon the zombies randomly over the edge of the map so that it will have don't repeat yourself design principle as the method can be called again in the class to perform similar actions. This class will be able to be used by other classes as well to spawn zombies at different locations.

5. VanishAction Class

VanishAction is extended from the Action class in the engine. It overrides the execute method in the Action class to remove the actor from the map. It will change the mambo marie's alive's status to false so that in the game world class, it can be checked to know that mambo marie has just disappeared from the map instead of being killed. By implementing the vanishAction class, this class can be reused by the developer to implement a new Actor that needs to disappear/vanish from the map for a temporary period of time. This shows that the design principle of Don't Repeat Yourself as the developer does not have to repeat the code to perform similar actions.

FIT2099 Assignment 3 Bonus Tasks

Bonus Task 2: New Enemy, Witch

To make the game extra more interesting, we are planning to add a new Enemy class to the game. The witch will have the ability to stun the player two turns. When the witch is standing next to the player, it will have 20% chances of stunting the adjacent player. It will not have any damage dealt to the player other than stunning the player for two turns. It will also check if the adjacent player is already stunned, so if they are stunned, it will wander around the map. This will make sure the new Enemy witch does not keep stunning the player. We will implement this by creating a new Witch class which extends from Actor class, StunBehaviour class that creates an Action to throw a stun to the target actor and stun the player in the StunBehaviour class too, modify the Player class to allow the player to be stunned for two turns when the stunt action is successful.

1. Witch Class

Witch class is extended from the ZombieActor class which extends from the Actor class in engine.

It stores a few behaviours such as StunBehavior, huntBehaviour and wanderBehaviour in the behaviour list inside the Witch class. This will allow the Witch to either stun the target player if it's beside it, or move closer to the player, or just wander around the map. During witch's playturn, it will check if the player is stunned by looking at the player's inventory and check if it contains the stunt item. It will then go through a list of behaviours inside the behaviour list to check if the witch is able to stun the player. Else, it will just move towards the human/player or wander around the map. If the player is stunned, then it will not check the stunBehaviour and

move on to check the other two behaviours in the list. When the player is stunned, it will move on to check if it has stunned the player on the previous turn or it has been stunned for two turns where the witch will be allowed to stun the player again. This is done by using a counter inside the class to check how many turns the player has been stunned in order to perform the stunBehaviour. The reason for this design is that the witch will be able to know which turns has the player been stunned and quickly decide to perform the stunBehaviour by just getting the player from the gameWorld class to check if it contains the stunt item.

2. Player Class

Player is extended from the Human class which extends from the Actor class in the engine. Inside the player class, it will have two methods to check if the player is stunned by the witch. One method isStunned() and the other updateStunnedStatus(). isStunned() method is checked to see if the player's inventory contains the stunt item, and it's checked in the player's playturn. If the player is stunned, it will add the DoNothingAction so that player will not allow to perform any actions for 2 turns. Another method updateStunnedStatus will update the stun count by 1 to keep track the number of times the player is stunned. If the stun count reaches 2 means that the player is stunned for 2 turns and it will be able to move again. It will remove the stunt item from the player's inventory so that when the witch is beside the player and checks for the player's inventory, it will be allowed to stun the player again. This design will have the Don't Repeat Yourself design principle as if the developer decided to make witch or a new actor to be able to stun other actors, then these two methods can be reused by other actor class to check if it's stunned and also update the stunned status without the need to implement more classes to perform such action. This can easily be done by adding the methods into the actor interface that allows all the actor to override the method and perform similar actions.

3. Stunt

Stunt extends from the Item class. It is not portable as the actor cannot carry the item. This class does not perform any actions but it's used by other classes to check if the actor contains/carries such an item, that means it is stunned.

4. StunBehaviour

StunBehaviour extends from Action class and implements the Behaviour interface. It stores the attribute of the Actor that represents the target to be stunned. This means that the constructor of StunBehaviour would need to have an argument of Actor to be instantiated and also the stunt item. Each behaviour of stunning can only stun one

player. So, the multiplicity of StunBehaviour to Actor is one. It overrides the getAction() method from the behaviour class and implements its own functionality, which in our case, to get the action of stunning an actor, it has to check whether the target is within 5 squares of the caller. In particular, the getAction method is going to check the list of exits the witch has and check if any of the exits contains the actor player. If the actor player is beside witch which means that witch will have the chance to stun the player. In our design, assuming that the witch is able to stun, then it will execute the StunBehaviour methods that are overridden from the Action class because StunBehaviour implements Behaviour interface and extends the Action class, so it has the ability to override the methods in Action class. The overridden execute method has a 20% chance of success. If succeeded, it first checks if the player is stunned. If it is not stunned, it will add an item of Stunt onto the player's inventory. This Stunt will have a stun effect that prevents the player from any actions for two turns, which will be checked by the player class.