Design Rationale - Assignment 2

Requirement 1 (Done by Amadea Theola Benedicta)

Requirement

Aside from HuntsmanSpider, several types of enemy are added with different spawn rates. The moon's crater should be able to spawn the enemy following its spawn chances. Despite the variety of enemies, each crater instance should only spawn one certain type of creature.

Changes to Design

In this implementation, the enemy class takes in an integer as a parameter for the constructor method representing the spawn rate of the object. This addition of attribute to the class eases the spawn of each enemy type depending on its spawn chance. This implementation ensures abstraction for actors that fall into the enemy category. As they are going to be spawned from the crater, setting spawn chance will allow the enemy instantiation to happen only for a certain amount of times.

Moreover, a spawner class is created to avoid the crater class being responsible for spawning the enemies, implementing delegation as its principle. This implementation eases modifying the behaviour.

Design Goal

The crater class is a ground object which overrides the tick method to spawn a certain type of enemy. Each crater object comes with a spawner object each instantiation. The spawner will be responsible for spawning the enemy and its function call of spawning enemies exists within the tick method. Since HuntsmanSpider, SuspiciousAstronaut, and AlienBug has different spawn rates, including an integer as a spawn rate for each instantiation of each enemy sub class enables the enemy to be spawned in certain occurrence. Encapsulating the attribute of an enemy class and a method to instantiate a new enemy object makes the enemy class responsible for its own properties. Moreover, the spawner class is created to ease the maintenance of the code for spawning enemies.

Pros

- Encapsulation: Separating the spawning logic and crater into different classes keeps the system modular and maintainable.
- Flexibility: New types of creatures and spawn strategies can be added without altering existing code, adhering to the open/closed principle.
- Scalability: The design supports easy scaling in terms of adding more creatures or changing spawn probabilities without major disruptions to existing functionality.

Cons

- Lack of Single Responsibility Principle: The method generateSpawnExit does more than one thing: it filters exits and then randomly selects one. This violates the Single Responsibility Principle (SRP), which suggests that a class or method should have only one reason to change. The method is responsible both for determining which exits are suitable and for selecting one of them, which could be better structured by separating these concerns into different methods or classes.
- No Error Handling: The method assumes that the exits list and the enemy are properly initialized and valid. There is no error handling to deal with potential null pointers or empty lists beyond what is implicitly handled by returning null if no suitable exit is found. This can lead to fragile code that might fail at runtime under unexpected conditions.

Requirement 2 (Done by Tong Zhi Enn)

Requirement

This requirement includes two new entities that will appear on the moon where the Player lands. This two entities are AlienBug('a') and SuspiciousAstronaut('ඞ'), where both are hostile enemies to the Player.

AlienBug is capable of picking up scraps on the ground, wandering around the moon and able to enter the spaceship. When Player is near its surroundings, it will follow wherever the Player goes. Since Player's job is to retrieve scraps on the moon, Player will be required to defeat AlienBug to retrieve stolen scraps.

SuspiciousAstronaut, is similar to the hostile HuntsmanSpider we implemented in the previous assignment, it wanders around the moon. But SuspiciousAstronaut is far more dangerous, it can instantly kill the Player, with 100% accuracy if it is within their surroundings.

Changes to Design

New status: CAN_ENTER_FLOOR is added into Status enum class. The existing Floor class is also modified with a new method canActorEnter() that returns the capability of CAN_ENTER_FLOOR, which allows certain actors that have the said capability to enter the spaceship.

New ability: PICK_UP is added into Ability enum class. This is used to indicate that certain enemy or creature(e.g. AlienBug) may have the ability to pick up scraps other than the player.

Existing AttackAction class is modified so that when a target(e.g. AlienBug) died from the player's hand, it will drop the scraps on the ground that it was carrying. This is done by performing a for loop to iterate through the target's inventory ensuring that everything it was carrying is fully dropped.

Design Goal

The design choice was to create AlienBug and SuspiciousAstronaut into Enemy objects, where both classes inherit from abstract Enemy class, this ensures that the approach adheres to DRY principle as they have common attributes. There are some override methods in this design choice such as playTurn() and allowableActions(). These methods will return the action performed by the Enemy and Player.

AlienBug has some behaviours such as WanderBehaviour, PickUpBehaviour and FollowBehaviour. These classes are created from implementing Behaviour interface, allowing to override the getAction method. PickUpBehaviour and FollowBehaviour is newly created for AlienBug class, where PickUpBehaviour returns a PickUpAction which allows the AlienBug to pick up scraps from the ground or floor when it walks into it, meanwhile FollowBehaviour returns a MoveActorAction which allows or move the AlienBug to follow the Player. Whereas, WanderBehaviour is reused from previous Assignment which adheres DRY principle to avoid code duplicates. To also ensure that AlienBug is able to enter spaceship, AlienBug is given a capability of CAN_ENTER_FLOOR to do so, where this allows the said actor to pass a certain terrain, in this case entering the spaceship.

SuspiciousAstronaut has WanderBehaviour and AttackBehaviour. AttackBehaviour and WanderBehaviour are reused from Assignment 1, adhering to DRY principle. As the specification suggests, SuspiciousAstronaut will attack the Player therefore AttackBehaviour is added for it, allowing it to return AttackAction when Player stands next to the SuspiciousAstronaut. It was also stated that SuspiciousAstronaut will only attack Player as Intern. To ensure this requirement is fulfilled, SuspiciousAstronaut will only attack Actor that have the capability status of HOSTILE_TO_ENEMY.

Pros

- The design choice adheres to the Liskov Substitution Principle (LSP). AlienBug and SuspiciousAstronaut is a subclass of the abstract Enemy class, allowing us to create more Enemy type classes in the future by overriding the methods as well as allowing consistencies in code.

- Existing or newly created behaviour classes can be reused in the future for other newly created Enemy type classes that have the certain behaviour, without violating the DRY principle.

Cons

- In future, if there is a new creature/enemy that is able to change follow target, the current FollowBehaviour class may not be feasible, and modifications to the code may be needed to fulfil the requirement.

    - The current FollowBehaviour will only forever follow one fixed target if the target entered their surroundings.

Requirement 3 (Done by Yin Shi)

Requirement:

- The Pickles (n) can be consumed by the intern and has a 50% of being past its expiry date, if passed its expiry date, the intern will hurt by 1 point, otherwise, healed 1 point.
- The gold ($) can be taken from the ground and added 10 credits into the wallet.
- Puddle of water (~) this increases the intern's maximum health by 1 point permanently. Consume the water from the ground directly without having to add anything to their inventory.

Design goals:

1. Pickles (n)

Pickle should be an item which can be picked up and dropped, pickles should also be consumed by the player, so, In the design, a concert object class extends from the abstract class "Consumables" called "PickleJar" is created. This ensures the pickles can be consumed by the player. (Because of the consumables abstract class contains the "allowableActions()" method (DRY) principle). In "PickleJar" class, "getHealPoint()" and "getItem()" is overridden, cause the "getHealPoint()" method in the "PickleJar" returns the random heal points which has 50% chance to be reversed. The "getItem()" method will return the item itself.

2. Gold ($)

- "Sellables" is implemented as an abstract class which extends from the Item abstract class in the Engine package. This class is used to help implement the items later which has an ability of adding credits to the player's wallet such as the gold object (Dependency Inversion Principle (DIP)). The method "allowableActions()" is created in the "Sellables" abstract class which returns a list of actions that the actor can do.

- The "sellAction" class is created to be a class extending from the Action class. When the "execute()" method is called, the balance in the actor's wallet is added, with the sellable Item removed from the actor inventory.

- Finally the "Gold" class extends the "Sellables" abstract class (Dependency Inversion Principle (DIP)).

3. Puddle of water (~)

The puddle modifies the basic attributes of the player, so "SpecialGround" abstract class which extends from the "Ground" class is created. And the "Puddle" class extends from this abstract class. This abstract class has fields of the attributes of how a “SpecialGround” should change the player’s attributes (Open-Close principle).

“DrinkAction” is created to be a class extending from the Action class. When the “execute()” method is called, The actor’s base attribute is changed.

The puddle class overrides the “allowActions()” method in the Ground, and returns a list of actions that can be done by the player which will be accessed in the “World” class.

Pros and Cons

- Pickles (n):

  Pros: extends from “consumable” abstract class without modifying the class itself, open for extension but closed for modification. (OCP)
- Gold ($):

  Pros: The “Sellable” class is created for extension. Other classes should not depend on the Item in the engine package, but not depend on the “Gold” class which is a low-level module. High-level modules should not depend on low-level modules. Both should depend on abstractions (Dependency Inversion Principle). The “SellAction” class extends the “Action” class also DIP.

- Puddle of water (~)

  Pros: create “SpecialGround” abstract class which extends from the “Ground” class (OCP). This abstract class is specialised for the ground types that change the base attributes which is beneficial for future modification. The “DrinkAction” class is the action cooperate with the “SpecialGround” that helps the player to modify the base attributes.

  Cons: This “DrinkAction” class violates the Single responsibility principle, This “DrinkAction” is capable of changing all the base attributes of the actor instead of changing a special type of attribute.
  The “SpecialGround” abstract class only copes with the additional field, but focuses less on the extra methods.

Requirement 4 (Done by Woon Chong)

Requirement

For the requirement, a Computer Terminal is added and whenever the Player enters within the vicinity of the Terminal, they are able to purchase one of three items offered by the terminal. Items sold by the terminal have a random occurrence that could range from costing double its original price to taking the money and not giving the item.

Changes to Design

Changing the previous assignment, a Consumable abstract class is created to handle all items that can be consumed by Actors. This was done to ensure that future items that can be consumed by actors will have fixed methods that other classes can use to interact with consumable items with ease. This will help follow the Dependency Inversion Principle where other classes will only need to depend on the abstraction of consumable items and not the items themselves.

Design Goal

Terminal would be a Ground object where it will override the allowableActions method from the Ground abstract class. The overridden method will return a list of BuyAction actions an Actor is able to perform to buy items from the terminal.

Currently the three items are of unique types where EnergyDrink is a Consumable, DragonSlayerSword is a WeaponItem and ToiletPaperRoll is an ordinary Item. With this, I have decided to create a Purchasable interface where items that implement this interface are ones that can be bought from the terminal only. Interface methods include getting the item's original price, getting the item's price and getting the item. By allocating all possible items that can be bought from the terminal to one type, BuyAction will be able to handle managing items as Purchasable instead of items as WeaponItem, Consumables and Item. Doing so will allow ComputerTerminal to create new BuyAction actions by passing Purchasable objects into its parameters without the need to modify the BuyAction action class to accommodate new Purchasable items, allowing for extension of the class usage. This would adhere to the Open-Closed Principle where BuyAction is open for extension and closed for modification.

Since each Purchasable item has its own unique random occurrence, I have decided to let each individual item handle their own random occurrence as not all occurrences are the same. These random occurrences are calculated when the BuyAction class invokes the Purchasable methods. For example, when displaying the menu description for an EnergyDrink, it will get the original price first. When the player executes the action, the action invokes the price of the EnergyDrink where there is a 20% chance it would return double its original price.

Pros

- BuyAction can always be sure the items it is interacting with will have Purchasable methods
    - Methods such as getOriginalPrice(), getPrice() and getItem()
    - Items in the future that will be sold by the terminal will always have these methods implemented as well
- New purchasable items can be added easily by creating new BuyAction actions in the ComputerTerminal class and passing the new Purchasable item to the constructor

Cons

- Future Purchasable items may have similar occurrences to previous Purchasable items
    - May have duplicated codes
- Future Purchasable items may have completely different random occurrences that may require modification to Purchasable interface and BuyAction action to accommodate these new random occurrences