FIT2099
Assignment 2: Design Rationale
Applied Session 2 - GROUP3

Members:

1. Sandra Lo Yii Shin (33338604)
2. Tang Wei Yao (32694849)
3. Wong Kien Wen (32886195)
4. Wong Jia Xuan (33411808)

REQ1 The Ancient Woods

Tasks:

- Implement new AncientWoods map
- Add new Hut that can spawn ForestKeeper
- Add new Bush that can spawn RedWolf
- For both enemies, modify their default attack and make them drop healing vials
- Implement new FollowBehaviour for enemies on this map

Maps GameMapBuilder

Moving between maps has been implemented in A1 REQ4, in which the game map string for Ancient Woods is stored in *Maps*, while map creation and object placement is handled by *GameMapBuilder*.

This implementation however introduces some connascence of meaning when accessing maps in different classes, ie. consistent map naming across classes. Incorrect spellings are identifiable by the compiler, but if more maps were to be used across different classes, constants may be used to mitigate this issue.

*LockedGate*s have been added to the required maps such that the player can travel back and forth between them.

Hut, Bush, ForestKeeper, RedWolf

Deviating from the implementation of spawning enemies in A1 REQ2, the enemy spawner (*Spawner*) is reintroduced as an interface implemented by various enemy spawner classes: *RedWolfSpawner, ForestKeeperSpawner, HollowSoldierSpawner, WanderingUndeadSpawner*. This change aims to clarify the responsibility of each spawner class, enhancing code organisation through separation of concerns.

Hut, Bush, Graveyard now take in a spawner in their constructors for spawning enemies. The dependency injection intends to improve the transparency of the code. This also allows the same ground type to spawn different enemies, at different rates.

As more enemies are implemented, however, the spawners can have code repetition among themselves. If necessary, a manager/factory class can be used to abstract the spawning mechanism, in turn this does add to the class hierarchy, but is consistent with the implementation of the other interfaces in the code.

It has been observed that all enemies, so far, will wander around if the player is not in range, and will attack the player if the player gets too close, leading to the conclusion that ‘wandering around and attacking the player’ is part of the general functionality of any enemy. Therefore, the overridden *allowableActions* and *playTurn* methods, and the creation of the behaviour map have been abstracted from the individual enemy classes to the *Enemy* abstract class (DRY).

This refactor aims to enhance maintainability where all enemies can have a general behaviour once its constructor is defined. Even if the enemy does not follow the aforementioned general behaviour, the methods can be overridden again in that enemy's class to implement additional/different features of the enemy without breaking the existing code (OCP, LSP). However this does lead to complex overrides if the deviation is significant, limiting flexibility for unique behaviours in favour of code organisation.

Again, as stated in the implementation in A1 REQ5, both enemies create an instance of *HealingVial* in their overridden *unconscious* methods, containing a call to the *Droppable* and *Consumable* interfaces and their manager classes to implement the drop and consumption of the *HealingVial* item. No additional code has been added for this requirement.

FollowBehaviour

To address the continuous following behaviour of enemies towards the player until an elimination occurs, a *PROVOKED* status is introduced to signify lasting behaviour. Due to this being an NPC trait, a new *FollowBehaviour* class is used to encapsulate the logic. (SRP)

In the *Enemy* abstract class' *allowableActions,* once the player is checked for their hostility, this enemy can be *PROVOKED* to follow the player, if this enemy can follow the actor (validated through *FOLLOW* ability).

Once provoked, the enemy persistently follows the player as long as they share the same map. Valid exits are determined by calculating the Manhattan distance between actors.

This feature is implemented in *Enemy* rather than the individual enemy subclasses. This decision stems from the recognition that 'following the player' is considered a common trait among NPCs in games. Thus, this design choice ensures scalability, accounting for the addition of future maps or enemy types with this behaviour outside of Ancient Woods.

REQ2 Deeper into the Woods

Tasks:

1. The enemies drop “Runes” (represented by “$”)
    - The “Forest Keeper” drops 50 runes
    - The “Red Wolf” drops 25 runes
    - The “Wandering Undead” drops 50 runes
    - The “Hollow Soldier” drops 100 runes.
2. The player can drink from the nearby puddle of water.
    - Direct drinking
    - Drinking from the puddle of water heals the player by 1 point and restores the player’s stamina by 1% of their maximum stamina.
    - The player can only drink water from the puddle that they are standing on
3. New item: Bloodberry
    - represented by “*”
    - After consuming the bloodberry, the player’s maximum health is increased by 5 points permanently

<u>Runes</u>

The player has a wallet that stores their collected runes once consumed.

Similar to how droppable and consumable items are implemented in A1 REQ4, *Runes,* extending *Item,* implement both *Droppable and Consumable* interfaces with a manager (ISP, SRP) to implement this feature. Enemies are said to have a 100% drop rate, thus the Enemy abstract class sets this as a field to be used across all enemies to pass into their respective managers’ *dropped* method. Individual enemies on the other hand, have the amount of runes they can drop as an attribute, as these values are not expected to change throughout the game.

The consumption of Runes is done through a new *consumeRunesItem* method in the *ConsumableManager*, which adds to the player’s wallet balance.

Consistent with previous implementations of HealingVial and RefreshingFlask, this implementation has proven to be effective as the code does not have to change significantly to extend this feature (OCP).

<u>Puddle</u>

Puddle is a subclass of *Ground* and implements the *Consumable* interface since it can be consumed. The interface’s manager is extensible to accommodate for the consumption of the entity even if it is not an *Item* type. A *consumeWater* method has been implemented to add to the consumer’s HP and Stamina. It is also possible to generalise the behaviour of direct consumption from ground, like *consumeHealingGround, consumeStaminaGround* once the pattern and effects of ground consumption have been identified, to reduce code redundancy in the future.

Bloodberry

Once again, as with the previous items, this is implemented as a *Consumable* that is added to the map through the *GameMapBuilder*. The item increases the max health, instead of healing the player, therefore a new method, *consumeMaxHealthItem* is added into the *ConsumableManager* without changes to previous code. This method accommodates any item that can affect max health.

All items stated above override *allowableActions* to return a new *ConsumeAction* to allow the player to interact with the item.

REQ3 The Isolated Traveller

Tasks:

- Create a suspicious traveller (represented by “ᘐ”).
    - Inventory:
        - Healing Vials
        - Sells Refreshing Flasks
        - Sells Broadswords
- Implement PurchasingPurchasing & Selling mechanism:
    - PurchasingPurchasing:
        - Healing Vials (100 runes per 1)
        - Refreshing Flasks (75 runes per 1)
        - Broadswords (250 runes per 1)
    - Selling:
        - Healing Vials (35 runes per 1)
        - Refreshing Flasks (25 runes per 1)
        - Broadswords (100 runes per 1)
        - Bloodberries (10 runes per 1)

Suspicious Traveller

The Traveller serves as an NPC (Non-player character) that allows the player to purchase and sell items, it is an essential component of the game's economy and interactions. So it has *allowableActions* to let the player interact (purchase and sell items) with him/her.

TheIsolatedTraveller is implemented as a child/subclass of the *Actor* class. It is also implemented with items in their inventory, including *HealingVials*, *RefreshingFlasks*, and *Broadswords*. Since it is implemented as an NPC, it does not move around the map (*DoNothingAction*). It stays in one spot inside a building within the forest and allows the player to do *PurchaseAction* and *SellAction* once the player reaches the Ancient Woods and is next to the traveller.

In line with A1’s Player implementations, this one has shown to be successful since it just requires minor code modifications to enhance the functionality of the new NPC (OCP), which means a different feature is implemented for a different Actor.

Implement Purchasing & Selling mechanism:

The *PurchaseAction* is associated with the *Purchasable* interface where we can bring the *Purchasable* item into the *PurchableManager* to perform item purchasing. In the *PurchasableManager* it checks if the player’s wallet/balance has enough runes for the item the player wants to purchase or not. It is implemented with a 50% chance to double the price of *HealingVial*, a 10% chance of getting a markdown for *RefreshingFlasks* and a 5% chance of not getting the *Broadsword* after paying 250 runes.

Similar to the *PurchaseAction*, the *SellAction* is associated with the *Sellable* interface where we can bring the *Sellable* item into the *SellableManager* to perform item selling. In the *SellableManager* it is implemented with a 10% chance of selling the *HealingVial* for 2x the

original price and a 50% chance of getting no runes after selling the *RefreshFlasks* to the traveller.

This design adheres to the SRP by clearly defining responsibilities for items, manager classes, and character classes, and the OCP by using interfaces and manager classes that allow for easy extension without modifying existing code, making it possible to add new items, characters, or transaction rules in a modular and maintainable way. Although this design encourages extensibility and maintainability which adheres to the SRP and OCP, the increasing number of classes and interfaces, which at first sight could seem complex, this is one potential downside.

REQ4 The Isolated Traveller

Tasks:

- Create a new weapon sold by the traveller, Great Knife
- Create a new weapon able to be picked up upon reaching Abxervyer battle arena,Giant Hammer
- Create Great Knife's skill, Stab and Step
- Create Giant Hammer's skill, Great Slam
- Implement a gate to Abxervyer battle

Great Knife and Giant Hammer

The Great Knife and Giant Hammer extends the WeaponItem class, containing base weapon features and item features. This allows for addition and modification of the weapon without unnecessary code repetition adhering (DRY, OCP and LSP).

Further, they utilise the AttackAction class to do the attacking, this also reduces unnecessary code repetition as all weapons use the same class to inflict damage on other actors. Which adheres DRY, OCP and SRP.

Utilising the PurchasableManager or SellerManager allows for easy modification and extension to how the traveller interacts with the player as a new class can just created for each possible future item the traveller buys/sells.

The weapon's respective skills "Stab and Step" and "Great Slam" extends the Action class, containing base action features. The overriding allows for addition and modification of the skills without unnecessary code repetition adhering (DRY, OCP and LSP).

Stab and Step creates an AttackAction object to inflict damage on the targeted actor and then searches for safe spots by storing available exits around the player to step away in a list and then selects that as the location to travel to.

Great Slam calls the AttackAction class by creating an AttackAction object with the default constructor to inflict damage on the targeted actor and then creates another AttackAction object with another constructor that contains the new damage after being affected by the damage multiplier to inflict damage on the targeted actor's surrounding actors, including the player.

Both of the skills use proper OOP rules such as initialising values at only one place and passing that variable rather than hardcoding it, adhering DRY.

Advantages:

Modularity: The use of multiple respective classes with specific responsibilities allows a streamline flow ensuring that changes in one part won't affect the others heavily

Extensibility: Adhering to OCP, we can introduce new weapons and skills without making many unnecessary changes to the existing code.

Readability: The distinct responsibilities of each class helps enhance the cohesiveness between the other classes allowing for better clarity.

Disadvantages

Complexity: Utilising method overloading can sometimes be confusing as the code extends and becomes larger over time.

The abundance of classes makes implementations for future codes easier, however, it can also be confusing especially for people that weren't directly involved in that particular section of code without proper well done documentation.

REQ5 Abxervyer, The Forest Watcher

Tasks:

- Implement new boss, Abxervyer
- The boss can follow the player
- Boss is immune to voids
- Implement weather control ability of the boss
- When defeated, drop runes, print a message and become a gate

Abxervyer

The *Abxervyer* is a subclass of *Enemy,* having a new ability of *VOID_IMMUNE*, other than *FOLLOW*. (REQ1)

As for its weather control ability, this has been implemented as an interface, *WeatherControllable*, accompanying a manager class to handle weather related matters. Entities that can control/be controlled by the weather implement the interface which contains a *weatherEffect* method to implement what is done to the entity for a specific weather. The method is generalised to affect all instances of its implementing classes for simplicity, stemming from the assumption that enemies are local to their maps/regions.

This segregation of weather-related functionality into a separate interface/class (SRP) allows for a straightforward addition of more entities with weather-related behaviours in the future.

The *Abxervyer* uses this method to control the weather by incrementing an internal counter and alters the weather every three turns. As there are currently only two weathers defined, the weather is toggled between them both. If there are more weathers in the game, then a list to index the weather cycle would be more suitable.

However this may lead to encapsulation concerns whereby the weather-controlled entities can have access to the weather-controlling methods in the manager. This can be mitigated by introducing another interface, eg. WeatherController, if there were multiple entities that control weather differently, or follow a different cycle of weathers throughout different maps.

weatherEffect is called in all implementing entities' *playTurn* method to realise a passive response to weather changes. For instance, RedWolf's *weatherEffect* checks the current weather, and alters its damage if it is sunny, Bush spawns RedWolf at 1.5 times the original spawn rate if it is rainy.

The *Abxervyer* has a *LockedGate* as a field, set to travel to *The Ancient Woods*. When *unconscious,* the gate is set up at its current location. As with REQ2, runes are added as a droppable item in the method as well.