FIT2099
Assignment 3: 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 Overgrown Sanctuary

Tasks: Create new map - Overgrown Sanctuary
Modify LockedGate so it can go to multiple destinations.
Create new enemies - Eldentree Guardian, Living Branch

The map is implemented using the same logic as A2 where the game map string for Overgrown Sanctuary is stored in *Maps*, while map creation and object placement is handled by *GameMapBuilder*.

LockedGate

The LockedGate class is modified to support multiple destinations per gate, aligning with the Single Responsibility Principle (SRP). This class manages gate-related functionality, including tracking destinations, unlocking, and allowing player movement. A new constructor accepting ArrayList is added to LockedGate to allow a single Locked Gate object to go to multiple destinations. By using an abstract class, gate functionalities. Three Array Lists are created in GameMapBuilder to act as the parameters for the locked gate to support multiple destinations. This adheres to the Open-Closed Principle (OCP) as it allows new destinations to be added without modifying the existing code. This extensibility is vital for accommodating future map expansions and varied gate functionalities, upholding the principle of open for extension but closed for modification.

EldentreeGuardian, Living Branch

Following the method of spawning enemies in A2, LivingBranchSpawner and EldentreeGuardianSpawner are introduced to manage the respective enemy spawning. This helps centralise the logic for enemy generation while mitigating code redundancy. Both these decisions contribute to a more organised and efficient codebase. A new Ability WANDER is added to indicate the ability to wander around. It is not added to be a part of LivingBranch's capabilities as it cannot move. This adheres to OCP as future enemy implementations can be done easily and reduces code redundancy at the same time.

Disadvantages
Complexity: Utilising method overloading can sometimes be confusing as the code extends and becomes larger over time.
Having many classes can make it simpler to add new code in the future. However, it can also be confusing, especially for those who didn't work on that specific part of the code, unless there's clear and well-done documentation.

REQ2: The Blacksmith

Tasks:

- Create a Blacksmith (represented by “B”).
- Implement Upgrading mechanism:
    - Healing Vial (250 runes per 1)
    - Refreshing Flask (175 runes per 1)
    - Broadsword (1000 runes per upgrade)
    - Great Knife (2000 runes per upgrade)

The Blacksmith

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

Blacksmith is implemented as a child/subclass of the *Actor* class. It is implemented differently from TheIsolatedTraveller where this *Blacksmith* won't have an inventory of items since it is not an NPC that sells items. Since it is implemented as an NPC, it does not move around the map (*DoNothingAction*), a small difference from TheIsolatedTraveller is that every round it will have a rhythmic sound of a hammer pounding against metal. And it stays in one spot inside a building within the building and allows the player to do *UpgradeAction* once the player reaches the Abandoned Village and is next to the traveller.

In line with A1’s Player implementations and A2’s TheIsolatedTraveller implementation, this one has shown to be successful since it just requires minor code modifications to enhance the functionality of the new NPC (OCP) and add a new mechanism to the game, which means a different feature is implemented for a different Actor (NPCs).

Implement Upgrading mechanism

The *UpgradeAction* is associated with the *Upgradable* interface where we can bring the *Upgradable* item into the *UpgradableManager* to perform item upgrading. In the *upgradableManager* it checks if the player’s wallet/balance has enough runes for the item the player wants to upgrade or not. It is implemented with item level where for *HealingVial* and *RefreshingFlasks* can only be upgraded once, but for the *Broadsword* and the *GreatKnife* can be upgraded multiple times. So, the item level is used to keep track of the item level to check if it is available for another upgrade and for the multiplier of damage/hit rate/healing amount/stamina recovery.

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, which is shown in this requirement/task. 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.

REQ3: Conversation (Episode I)

Tasks:

- Implement an option to listen to the blacksmith's monologue
- Some options can only be performed when conditions are met: ie. player has/has not defeated the Abxervyer, player holds a great knife

Monologue, BlacksmithMonologue

A *Monologue* abstract class is created to handle monologue related functionality (SRP). Thus far, this includes an abstract method, *getAvailableMonologue* which takes in the listening Actor as an argument and returns a list of monologues that the Actor can listen to. Declaring this as abstract ensures that each subclass provides an implementation for the method despite the diverse monologue options, adhering to LSP. Future extensions for the feature may introduce more complex monologue interactions in separate methods of the monologue subclasses.

Following this, the method is implemented in the subclass *BlacksmithMonologue* where all strings of monologue are initialised locally and added to an ArrayList to be returned to the calling method. The *Blacksmith* then has a *BlacksmithMonologue* object as a field. The extension allows new monologues and conditions to be added without modifying the existing code, aligning with OCP.

Status enums are used to indicate special conditions that generate different monologue options. Namely, *HAS_GREAT_KNIFE* and *ABXERVYER_DEFEATED*. These are added to the player's *capabilityList* when relevant, and are checked for within the same method to reduce code complexity.

The current design provides a clean and clear implementation for the current requirements of the game, and also adheres to OCP in which extensions can be made for more conditional availability checks. However, while effective in this context, such extensibility is limited, for example, if the game expands to include more complex interactions with different actors, or the conditions affect more aspects of the game (eg. actors reacting to the Abxervyer's death, not restricted to monologues), it would be suitable to transition the feature to follow the Observer pattern, whereby an interface is defined and implementing classes react to the condition accordingly, similar to the implementation of the weather feature in A2 REQ5.

ListenAction

A concrete *ListenAction* extending abstract class *Action* is created to facilitate the interaction, allowing for polymorphism through abstractions (DIP). The class is associated with a *Monologue* class and a *talkingActor,* allowing the class to effectively execute its assigned monologue library. In the execute method, *getAvailableMonologues* is called to obtain the monologues available, and randomly selects a line to display on the menu. This is

consistent with the use of Actions in the game. The Blacksmith adds a *ListenAction* to their *ActionList* if the interacting actor has the ability to *LISTEN*.

REQ4: Conversation (Episode II)

Tasks:

- Implement an option to listen to the Isolated Traveller's monologue
- Some options can only be performed when conditions are met: ie. player has/has not defeated the Abxervyer, player holds a giant hammer.

IsolatedTravellerMonologue, ListenAction

As this requirement is identical to REQ3, the section will only briefly describe the extension of the Monologue feature.

Following the implementation in REQ3, an *IsolatedTravellerMonologue* object is added as a field in *TheIsolatedTraveller*, containing an implemented method *getAvailableMonologue* that returns a list of default monologue, and additional monologue options if the player has the status of *HAS_GIANT_HAMMER* and/or *ABXERVYER_DEFEATED.* This is consistent with the previously established design pattern for monologues in the game.

The actor can then have the option to listen to one of these monologue sentences through the execution of *ListenAction* in *TheIsolatedTraveller's allowableActions* method. Adhering to OCP, the design has allowed the addition of a new character monologue without breaking existing code.

As with the *Blacksmith's* monologue feature, the approach aligns with the current context of the game while providing code clarity by using the enums to check for conditions.

REQ5: A Dream?

Tasks:

- Respawn player back to initial born place if the player dies due to any causes, maximum values of the player's attributes will be kept and will be reset to full. The items will remain in the inventory but the rune in the player's wallet will be dropped in the last position and reset to 0.
    - All spawned enemies not including bosses will be removed from the game map. The bosses' health will be reset to full if they have not been defeated.
    - Unlocked gates will be locked again.
    - All runes in the previous turn that are left on the map will be removed, including all runes dropped by enemies and the player.

ResetManager

The 'ResetManager' stores all the instances of classes adhering to 'Resettable' interface in a list. It provides the related management method for the list and a centralised control among the 'Resettable' (OCP). It provides singleton control over these 'Resettable' objects without any other modifications to the 'ResetManager' as it only focuses on reset (SRP).
Therefore, it raises a risk that the future implementation might violate the SRP if the implementation doesn't focus on resetting responsibility.
The 'ResetManager' provides a centralised approach for timely resetting all 'Resettable' objects in the list and makes it easy for the implementation of turn-based games. Furthermore, it allows the game to handle the list as the game continuously progresses.

Resettable

This is an interface for all game objects which can be reset (ISP). It required the related objects' class to implement the reset() method, which will be invoked by ResetAction() (DIP and LSP). Through this interface, the objects can have their own reset mechanisms without downcasting.

ResetAction

This is an action that will be used by the player only if the player has the Status.DEAD and Ability.REVIVE. It has the responsibility to handle all the related consequences after the player dies. It will runReset() function of the ResetManager class, subsequently invoking the reset() methods for all 'Resettable' classes in the list via abstract interfaces. Basically, the reset() methods for all 'Resettable' classes are adding Status(Enemy except the boss), adding ItemCapability(Runes), a special mechanism for Abxervyer boss, and a special mechanism for LockedGate. The player will be respawned through this action, this result was treated as a consequence of the player dying.

RemoveActorAction

This action has a dependency on the Actor classes. It will remove the Actor classes from the map. It was separated from the Actor as this action was designed to remove other actors rather than the player (SRP). Furthermore, it is one of the classes implemented 'Resettable', this action improving extensibility and adhering to OCP.

Further explanation:
playTurn() in game engine:
The enum that is added to the related ‘Resettable’ object from ResetAction() will be checked in this method. The related actions or mechanisms will be invoked.