Changes from A2 Code (Sarviin)

The changes made from A2 to change the implementation of consumption due to the comments from the marker as below:
As the definition of consume, drink, eat have been intepreted as different types of "consumption", design of "consuming" items are not particularly extensible.
To solve this issue, we have changed the implementation similar to playerpurchase and playersell, where we have one ConsumeAction class and one interface called ConsumeAbility which will be implemented by consumable items which are BloodBerry, Healing Vial, Refreshing Flask, Runes and the ground of water (Puddle). The implementation has a consumeItem method which has the implementation of the specific items consumption (how it increases stamina or health etc.) and new COnsumeAction is created with the instance of the item itself stored as ConsumeAbility.

Pros:

- The implementation of the ConsumeAbility interface and ConsumeAction class provides a flexible way to define consumable items and their effects. You can easily add or remove consumable items and adjust their effects without significant changes to the code structure. This approach adheres to the Single Responsibility Principle, with each item determining its consumption effect, and the ConsumeAction class facilitating the consumption process based on specified conditions.

- The ConsumeAbility interface abstracts the consumption behavior of items, allowing various classes to implement it. Any class that implements this interface contains the action that the specific item does when it is consumed, enabling different types of consumable items to have it as it is very close to the SRP Principle. This is a superior approach compared to creating multiple interfaces for each item, which is not suitable for extension and may require multiple if-else or instanceof checks.

Cons:

- The implementation assumes that each item is associated with a single consumption type and approach only. If the game later requires dynamic effects based on various factors, this simplistic approach may not be sufficient.

    - However, since the requirement never specifies the changes required in different instances of the same item, the current approach is still a better approach.

Requirement 1:(Raynen)

Implementation of gates leading to multiple locations (I acknowledge the use of ChatGPT):

To implement gates being able to lead to multiple locations, the LockedGate class now has additional constructors that takes in an ArrayList of LocationMaps or a single LocationMap depending if the gate leads to multiple locations or just one. The LocationMap class takes in a Location and a String, which associates a given location on a certain map with a string that describes the act of moving to that map. It has the methods getLocation() and getMapToMove() which returns the location and string associated with that LocationMap. Now rather than just making a single MoveMapAction in the allowableActions() of LockedGate, we iterate through the ArrayList consisting of LocationMaps and make a MoveMapAction based on each LocationMap iterated through using the methods getLocation() and getMapToMove(). This then enables the player to move to multiple maps once the LockedGate has been unlocked.

Why:

Creating a new LocationMap class provides a convenient way to manage and organize the Location and Strings within an ArrayList. Instead of handling a collection(ArrayList) for Locations and another one for Strings, we can encapsulate them together using the LocationMap class which enhances code readability and maintainability. If we were to handle a single collection(ArrayList) for Locations and another one for Strings (A Design Alternative), this would likely increase the complexity of our code as we probably have to use nested for loops in the allowableActions of LockedGate when making MoveMapAction's.Moreover, using LocationMap's introduce some uniformity in our code which would be absent in having a single collection(ArrayList) for Locations and another one for Strings which is why the LocationMap design was chosen. Regarding LockedGate, it can easily be extended in the future as the number of Locations that the LockedGate can lead to is completely dependent on what is passed in upon instantiation. This would mean that if a LockedGate were to move to 3 different locations instead of one, the LockedGate can simply be changed to take in 2 additional LocationMaps.

Pros:

- The LocationMap class adheres to the single responsibility principle as its main purpose is to represent the association between a Location and a String that describes the movement to the map of the Location (LocationMap encapsulates the association between the Location to move to and the String that describes the movement).
- LocationMap reduces code complexity as nested loops for generating MoveMapAction instances are avoided
- Scalability as extending the LockedGate class to lead to multiple locations is completely dependent on what is passed upon instantiation
- Adherence to Open-Closed Principle as gates can move to more locations without altering the LockedGate class (Dependent upon LocationMaps in the passed ArrayList)

Cons:

- The LocationMap class adds an extra class to the codebase which introduces a level of indirection and complexity
- The locations that a LockedGate can lead to is fixed upon instantiation. Indicating that if a LockedGate should be able to move to a new location depending on certain conditions in the game, this could not be done as there are no methods which allow us to add a LocationMap to LockedGate's ArrayList of LocationMaps

Change in implementation of the common Behaviours of Enemies (I acknowledge the use of ChatGPT):

Before LivingBranch, all enemies in the game had the behaviours AttackBehaviour and WanderBehaviour but with the introduction of LivingBranch the Enemy abstract class had to be modified. Previously, the Enemy abstract class added the WanderBehaviour and AttackBehaviour to the Enemy's TreeMap in the constructor of Enemy. However, since LivingBranch cannot wander, a new method called addBehaviours() was made in Enemy which adds AttackingBehaviour and WanderBehaviour to the Enemy's TreeMap of behaviours. The addBehaviours() method is called in the constructor of Enemy to add the respective behaviours into the Enemy's TreeMap. To prevent LivingBranch from having the WanderBehaviour(the LivingBranch class extends SpawnableEnemy which extends Enemy), the addBehaviours() method could be overridden in LivingBranch to only add the AttackingBehaviour into LivingBranch's TreeMap of behaviours instead. Doing this prevents LivingBranch from having the WanderBehaviour.

Why:

This strategy prevents the repetition of code as no new abstract classes are made for enemies that only have WanderBehaviour. Moreover, if enemies were to now have either a wanderBehaviour or attackBehaviour but not both, the enemy classes could just override the addBehaviours method to only add the desired behaviour in the Enemy's TreeMap. Regarding future extensions, if Enemies were to be capable of more Behaviours, whether common for all Enemies or not, they could be added in the Enemy abstract class's addBehaviours method given that most Enemies were to have that Behaviour or they could just be added in the overridden addBehaviours() method of that specific Enemy. This allows us to customise the Behaviours of the Enemy abstract class and classes that extend Enemy conveniently.

Regarding a design alternative, a new abstract class called VersatileEnemy could have been created which would be extended by Enemies that do not have the common behaviours of an Enemy (WanderBehaviour and AttackBehaviour). Enemies that extend VersatileEnemy would then be implemented by interfaces like Wander, Attack and Follow depending if they exhibit those behaviours. Each of these interfaces would then have a corresponding method that should be implemented, similar to that of the followBehaviour() from the Follow interface which creates a FollowBehaviour. However this design alternative would be poor as it increases the complexity of the codebase and results in an inefficient use of inheritance due to the need of creating separate Interfaces for each behaviour like Attack and Wander. Having to create an interface for each behaviour seems redundant as it increases unnecessary dependencies and possibly creates a lot of overhead in terms of code. Above all, this design alternative is redundant as it could be achieved by the addBehaviours() method in Enemy, reducing code duplication as new classes and interfaces do not need to be created.

Pros:

- Modularity: the addBehaviours() method encapsulates the adding of Behaviours which prevents the constructor of Enemy from having an additional responsibility (Adhering to the Single responsibility principle), this simultaneously increases readability of code as the adding of Behaviours is found in the addBehaviours() method
- Flexible: Allows customization of Behaviours in classes conveniently
- DRY is not violated as for Enemies that have AttackBehaviour and WanderBehaviour do not need to override the addBehaviours() method but rather rely on the addBehaviours() method of Enemy (No code duplication for Enemies that can Attack & Wander)
- Promotes adherence to the Open-Closed Principle by allowing for the extension of behaviours without modifying existing code

Cons:

- Increase in complexity as developers need to be aware of which methods to override and which not to, making the code harder to understand and maintain
- Connascence of Name as addBehaviours() can be overridden in classes that extend Enemy, indicating that addBehaviours() can appear several times in the codebase. This can lead to maintenance issues as if the name of addBehaviours() were to change in one place, it needs to be changed in all to maintain consistency.

Requirement 2 (Jun Hirano)

The game code introduces an item-upgrading mechanism through the Upgrade interface, which allows specific game items to possess their own logic for upgrades. When an item implements the Upgrade interface, it must define the upgradeItem() method detailing its upgrade process.The UpgradeItem class represents the action of upgrading an item. It checks if the actor performing the upgrade has sufficient balance for the transaction. Upon a successful transaction, the item's upgradeItem() method is invoked to apply the upgrade logic. Notably, it appears that the code has commented out portions suggesting capabilities of upgraded items, hinting at potential future enhancements. When a HealingVial is upgraded, the fraction of health it restores (HEALTH_REGAIN) is modified. Also, it uses the UpgradeStatus capability to track its upgrade state. Specific checks ensure that items like the HealingVial cannot be upgraded multiple times. The upgrade process also considers the type of actor (e.g., MERCHANT or SMITH) interacting with the item, providing flexibility in game dynamics. Overall, the code framework allows for interactive game actions tied to runes transactions and item capabilities, with specific attention to the upgrade process. An alternative approach I had considered was centralizing the item-upgrade logic within the BlackSmith class. The rationale behind this was the potential future introduction of multiple smiths, each possessing unique upgrade logics for items. However, this method posed a challenge. Given that we already have five upgradable items, introducing a new smith would mandate re-implementing the upgrade logic for all these items, even if the logic remains consistent across smiths. To address this redundancy and ensure scalability, we chose to embed the upgrade logic within each individual item class, leveraging the Upgrade interface.

Pros:

- One of the most evident principles followed in the UpgradeItem class is the Single Responsibility Principle. This class is solely tasked with handling the upgrade of items. By assigning the responsibility of upgrading an item to this class, we ensure that the code is modular and that each class in the game architecture serves a distinct purpose. Such modularity facilitates easier debugging and future changes.
- The implementation adheres to the Open/Closed Principle. The use of the Upgrade interface in the design ensures that the UpgradeItem class is open for extension but closed for modification. If, in the future, we wish to introduce a new type of item upgrade, we would just need to implement the Upgrade interface without needing to modify the existing UpgradeItem class.
- The code demonstrates low connascence. For instance, the upgrade behavior is achieved through an interface (Upgrade), implying a connascence of type.

Cons:

- Lack of Flexibility in Price Model: The current design assumes a fixed price for upgrading every item. This might not offer flexibility if, in the future, we want different items to have variable upgrade costs. This design decision could lead to a rigidity code smell.

- The current design of the item upgrade mechanism is tightly coupled to the item class itself, rather than being dependent on the blacksmith (or other upgrading entities). In a dynamic game environment, we might encounter various blacksmiths with distinct abilities or methods of upgrading items. The present structure, however, falls short in accommodating such variability. Specifically, the upgrade logic is embedded within the item suggests that the item dictates how it's upgraded, regardless of which blacksmith performs the action. Ideally, different blacksmiths might offer unique enhancements or different upgrade paths. Thus, the current design lacks the flexibility to adapt the upgrade logic based on the type or capabilities of the blacksmith, making it less extensible and adaptable to diverse game scenarios.

Requirement 3 (Lee Ann)

To allow the player to listen to the Blacksmith's monologues, a ConversationAction class (extending Action) and a MonologueAction interface was created. Blacksmith will then implement the MonologueAction to include all the dialogue options. There are 2 methods inside of MonologueAction, one for standard dialogues which do not require special conditions, and another one for conditional dialogues. For conditional dialogues, we created a few Status enums to check for the conditions. The dialogue options are stored in an ArrayList and the ConversationAction class will get a random number bounded by the length of the list to return a dialogue option. Using an interface instead of an abstract class is a better method as different actors have different dialogue options and also because Blacksmith is already extending Actor and therefore cannot extend another abstract class. Furthermore, the approach of using a new class for monologue list does not make a difference as it is just adding extra dependency and we still have to create a new list from the new class which is what our current implementation is

Pros:

- Action classes adhere to Open-Closed Principle so for future extensions, other classes can use ConversationAction and MonologueAction without needing to modify the existing code for them
- ArrayList is dynamic so we do not need to worry about the size of the list if we want to add more dialogue options in future
- Two methods were created under the MonologueAction interface to maintain easy visibility and prevent one method from being overcrowded and messy.

Cons:

- If a future actor needs to implement MonologueAction without having any standard dialogues (all conditional) then there will be a redundant block of code (standardMonologue()). This could be resolved by creating another interface for dialogues that do not have standard monologues.
- May need to create a lot of different Status types if there are many conditions which needs to be met
- Uses many if-else statements due to the many different conditions

Requirement 4 (Lee Ann)

Requirement 4 has a similar nature to requirement 3. We will be extending the ConversationAction and MonologueAction classes to be used with IsolatedTraveller. Since our implementation from requirement 3 is easily extensible, it was easy for us to integrate the implementation with IsolatedTraveller.

Pros:

- Easy integration of implementation from requirement 3 without having to modify the existing code. We only had to implement the interface and add the dialogue options
- Using ArrayList allowed for IsolatedTraveller to have more dialogue options (compared to Blacksmith) without needing to modify existing code
- Use of interfaces allowed for IsolatedTraveller and Blacksmith to share the same methods while having completely different dialogue options and conditions

Cons:

- Some similar code (such as clearing the list, calling standardMonologue(), and returning a conversationAction) had to be repeated which does not adhere to DRY principle.

Requirement 5 (Sarviin Hari)

1. To handle the resetting of all the actors, items and grounds in all of the Maps, we create a new class MapReset to handle this resetting of all the game maps in the application.

Pros:

- Having a new MapReset class adheres to Single Responsibility Principle as this design segregates the responsibilities of resetting all of the game maps to one class. Although for the current design only the Player has to MapReset and not implementing all the resetting of the game maps in the class of a Player reduces unwanted association and dependency of Player with a list of GameMaps and other classes. This not only ensures SRP it also ensures the modularity and maintainability of the code and prevents tight coupling between classes.

- Having a MapReset in a separate class also allows the design to adhere to DRY principle. This is because having a new class that handles MapReset avoids redundancy by isolating map reset logic into a separate class, promoting code reuse and maintainability. As for future extensions, we can use the same code if the same MapReset has to occur if another actor dies.

- The dedicated reset methods for each actor, ground, and item in the MapReset class offer future flexibility. In case the assignment requires resetting the state of a specific actor, item, or ground, these methods can be easily accessed through the default non-parameterized constructor of the MapReset class. This approach aligns with the DRY principle by enabling code reuse, eliminating the need to duplicate code. Furthermore, this also reduces unnecessary dependencies and associations with individual objects (actors, grounds, items) by centralizing the reset logic within the MapReset class.

Cons:

- Introducing a dedicated class for map reset adds complexity to the codebase. While it modularizes the reset logic in one class, it also means developers need to manage additional set of methods and dependencies
    - However, the complexity is still a better tradeoff to the OOP Principles as having these methods defined in Player class defeats the purpose of OOP as we have unnecessary association and dependency from Player class to other classes

2. For the implementation of the death of every enemies that dies, due to the extensibility of our system based on our coding practice in A2, we managed to solve this problem by adding a new status called status.SPAWNED to every new spawning enemy (in SpawnableEnemy class).

Pros:

- Alternate implementation of using instance of checks if the enemy is in the class of any of the 5 different classes of Spawnable enemies is not a good approach and does not deal well with the OOP principles as it goes against the Open-Closed Principle as whenever a new enemy is introduced, we have to add a new instance of the class checking thus resulting in poor extensibility of the code

- This design adheres to the Open-Closed Principle as we don’t need to have multiple if-else or instance of checks since we already standardise to remove the capability from all the enemies that are spawning by setting them to have a capability of status.SPAWNED. Thus, this also allows future extensions as the new enemies being added just need to have the status.SPAWNED capability to allow this functionality of the code.

- The use of a capability of status.SPAWNED, results in low coupling as this minimises the direct dependencies between classes, as we don’t need to know the type of actor class, as we only need to know if that actor has a capability. This enhances code flexibility and maintainability.

Cons:

- As for the cons, if we were to add more capabilities, managing multiple capabilities for an actor can become complex their number increases, but it is still a good trade-off to maintain OOP.

3. For the removal of the items from the ground, the same concept of the status.SPAWNED is used. When we want to remove all Runes on the ground, we will add the capability status.DESTROY_IF_ON_GROUND which specifies to the MapReset class that if the item has this status then the item should be removed from the ground. As mentioned in the previous explanation, this way of coding allows the adherence to the Open-Closed Principle and low coupling. This is also extensible as if we have a new item, we just have to add the status DESTROY_IF_ON_GROUND, so any items that are being on the ground with that capability will get removed. In addition, items without this status will not be removed from the ground or if it is in the inventory of the player adhering top the requirement.

To update the player actions after the player dies, i.e. Update Stamina, Update Health and Drop Runes, we will implement the code in the Player class itself.

Pros:

- This adheres to the Single Responsibility Principle as Segregating the player's responsibilities into multiple functions while aligning with SRP, also enhances code maintainability and manageability. By having distinct functions for updating stamina, health, and dropping runes, each function is responsible for a specific aspect of the player's post-death actions. This separation of concerns promotes clarity and makes the codebase easier to navigate, understand, and maintain. It also reduces redundancy by ensuring that each function addresses a unique responsibility.

- It helps the code to be modular and efficient to keep these operations inside the Player class. The need to navigate between different classes to develop and manage death-related logic is eliminated when all actions taken after the death of the Player are contained within the Player class. This makes the structure of the codebase simpler and more understandable. This eliminates the need to interface with numerous separate classes when using the Player class to efficiently access and alter these actions. This design decision optimises the organisation of the code and simplifies the installation and upkeep of the player after-death functionality. This design also indirectly supports the Open-Closed Principle (OCP) by allowing for extensions without modifying existing code by not altering the existing logic, and allows the code open for extension but closed for modification.

Cons:

- Strict adherence to SRP could lead to a high number of specialized, small classes, potentially making the codebase harder to navigate
    - However, since the current implementation doesn't involve complex coding practices, the current implementation in terms of reusability, modularity and navigability is still a good implementation.

4. To respawn the player back to the original position when the player dies, we pass in the original location's MoveActorAction instance that consists of the instance of an action to move a player from one map to the other as specified. So, when the player dies, the player will move to the location and the map specified in the constructor.

Pros:

- We opt to use the current player instance rather than creating a new instance for the Player at a new location as this will cause extra complexity and dependency to the code as every item from the player needs to be transferred to the new instance and might cause tight coupling between the classes. This implementation may also result in code smell as creating multiple new instances whenever the player dies might result in overuse of object creation and results in unnecessary memory consumption and decreased performance.

- By having the Player class handle its own respawn logic, this design adheres to SRP, as the Player is responsible for handling its own respawn to a new location. Keeping the respawn logic within the Player class simplifies the codebase. It avoids complex transfer of player attributes and items between instances and minimises code adhering to the DRY Principle

- This implementation is done in the player class rather than creating a new class or implementing in a different class so that this implementation adheres to the Single Responsibility Principle as the Player is responsible for respawning back to the new location.

Cons:

- Although reusing the existing player instance simplifies some aspects, it could potentially lead to tight coupling between classes. The respawn logic depends on the specific MoveActorAction instance passed in the constructor, which may introduce a form of coupling.
    - However, the current implementation seems better as it optimises the performance of the implementation while adhering to the OOP Principle especially Single Responsibility Principle as the Player can only have an instance of MoveActorAction, while the implementation is executed in MoveActorAction class itself rather than the Player class. Although there might be tight coupling it is still a better trade off to the advantages in terms of modularity and SRP of the current implementation

5. For the implementation of Locking back all the gates, due to the extensibility of our code from the previous requirements, to lock all the gates in all of the maps, we just have to remove the ability.MOVE_MAP from all of the grounds. This is due to the implementation of LockedGate in prior assignment, where the Player can only have the option to MoveMap when the LockedGate has the ability.MOVE_MAP, else, the code requires the player to Unlock the gate first. So, no changes were made to the LockedGate or UnlockDoorAction or MoveMapAction, as the only code specified in the MapReset to remove the capability ability.MOVE_MAP will make all the opened gates to be locked.

Pros:

- This is extensible as if we have a different type of Gate which cannot also be locked or Unlocked, the same concept can be used as above by just removing the specific capability to re-lock all the gates, thus this current implementation can very much be applied for most cases.
- This implementation is highly extensible. It can be applied to many kinds of gates that can be locked or opened without requiring significant code modifications as we only have to specify the capabilities to be added or removed to lock or open the gate. All gates can be re-locked by deleting the particular capacity (ability.MOVE_MAP in this example). It is a reliable option for managing different gate types because of its flexibility.
- This implementation also results in low coupling as we don’t really have any direct dependencies to any specific classes of Ground child classes, i.e. LockedGate or Puddle, instead, we only have dependency with the Ground class that removes the capability from every ground, thus improving the code maintainability.

Cons:

- This implementation carries no notable disadvantages. However, when removing capabilities from grounds, it's crucial to take into account the possibility of unforeseen consequences. The behaviour of other game elements may change if they make use of the same capability.

    - To ensure no such problems occur, we always make sure to only reuse the capabilities when the definition is aligned, thus why the capabilities used have very distinctive names rather than being too generalised