FIT2099 Assignment 3

UML Diagrams & Design Rationales

Group name: MA_Lab06_Group6

Group member:

1. **Che'er Min Yi 32679939**
2. **Chong Ming Sheng 32202792**
3. **Lam Xin Yuan 32245211**

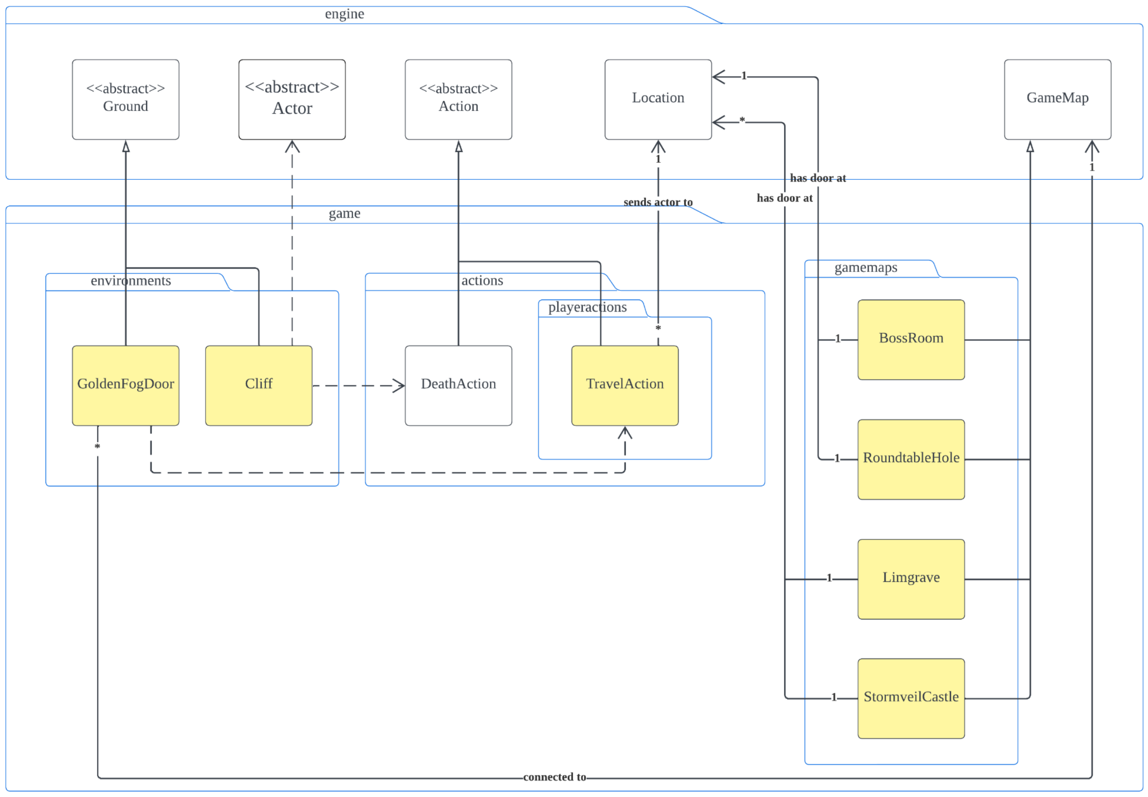
Link to group's contribution logs spreadsheet:
https://docs.google.com/spreadsheets/d/1vDWOI0y7OB4fWNsCS8IczHFDFUI4hhW5-LMqDeUrgBY/edit?usp=sharing

Below is the list of keywords and abbreviations for the design principles that our team used in the design rationales:

1. Don't repeat yourself - DRY
2. Single Responsibility Principle - SRP
3. Open-closed Principle - OCP
4. Liskov Substitution Principle - LSP
5. Interface Segregation Principle - ISP
6. Dependency Inversion Principle - DIP
7. Excessive use of literals

Note: The flow of this report goes from Requirement 1 to Requirement 5. For each requirement, the UML diagram is presented, followed by the design rationale. The new classes added into the game system are highlighted in yellow in the UML diagram.

Game Requirement 1



1. Created new grounds “Cliff” and “GoldenFogDoor” that inherit from abstract class “Ground”
   - OOP principles: SRP, OCP and DRY
   - Pro: Since this ground does not spawn enemies, we decided to inherit the “Ground” abstract class instead of the “SpawningGround” abstract class. This is to ensure that both classes have the proper responsibilities defined from its parent class and does not give the wrong responsibility such as spawning an enemy.
   - Pro: Defining “Cliff”’s unique functionality which is to kill the player instantly when the player steps on a cliff. This ability is defined in the “Cliff” class itself in the *tick()* method. This allows the system to check at every turn, if the player is standing on the location, if yes, then the player’s maximum HP will be reset to 0 and death action will be executed.
   - Pro: Defining “GoldenFogDoor”’s unique functionality which is to allow the player to travel to other parts of Limgrave. This ability is defined in the “GoldenFogDoor” class itself in the *allowableActions()* method. If there’s any “GoldenFogDoor” at the player’s surroundings, the door will give the player a “TravelAction” so that the player can choose if he wants to travel to a specific game map.
   - Pro: This new ground is easily extended from the “Ground” abstract class and uses all the common functionalities from the parent class. Their unique functionalities have demonstrated our game system is opened for extension, but closed for modification.

- Pro: Any unique ability or responsibility of the “Cliff” class can be added into its own class directly without modifying the parent “Ground” class and thus not affecting other ground classes
- Pro: By extending the subclasses “Cliff” and “GoldenFogDoor” from the “Ground” abstract class, repetitive code for the common functionalities that all types of ground share is prevented in the subclasses.
- Future extension: Any new functionalities that are introduced in the future (for instance increasing player’s maximum HP) can be easily implemented in the subclasses itself without affecting its parent class and other existing “Ground” subclasses.

2. Created new class called “TravelAction”
    - OOP: OCP and SRP
    - Pro: The functionality of moving the player from one game map to another game map should not be implemented in “GoldenFogDoor” class as this will cause the “GoldenFogDoor” to have too many responsibilities and thus violate the SPR principle. “GoldenFogDoor” should only know the destination of the location that itself connects to but it should not know how to travel the player to the destination. This responsibility should be handled by another class, which is the “TravelAction” in our case.
    - Pro: Since the player is given an option to choose if he wants to travel to another game map, “TravelAction” class extends from the “Action” abstract class to share the common functionalities that a “Action” should have such as printing specific descriptive strings to the console.
    - Pro: Extra or unique functionalities could be added to the “TravelAction” class easily in the *execute()* method.
    - Future extension: Any new functionalities that are introduced in the future (for example increasing player’s current HP before travelling) can be easily implemented in the “TravelAction” class itself without affecting its parent class and other “Action” subclasses.

3. The “TravelAction” will be added to “Actor”’s allowable action for each turn at the “GoldenFogDoor” ground’s *allowableActions* ActionList if the ground is at the surroundings of “Actor”.
    - An instance of “TravelAction” will be added into the selectable actions list of the player, so that in the console the player can choose to execute the “TravelAction” in order to travel to some specific parts of Limgrave when the user is at the surroundings of a “GoldenFogDoor”.
    - As mentioned above, “GoldenFogDoor” should not handle the functionality of travelling an actor to another location. Hence, “GoldenFogDoor” should pass this functionality to “TravelAction” where appropriate.
    - Pro: Whenever the player is in the surroundings of the “GoldenFogDoor”, he will be able to choose if he wants to execute the action via console.

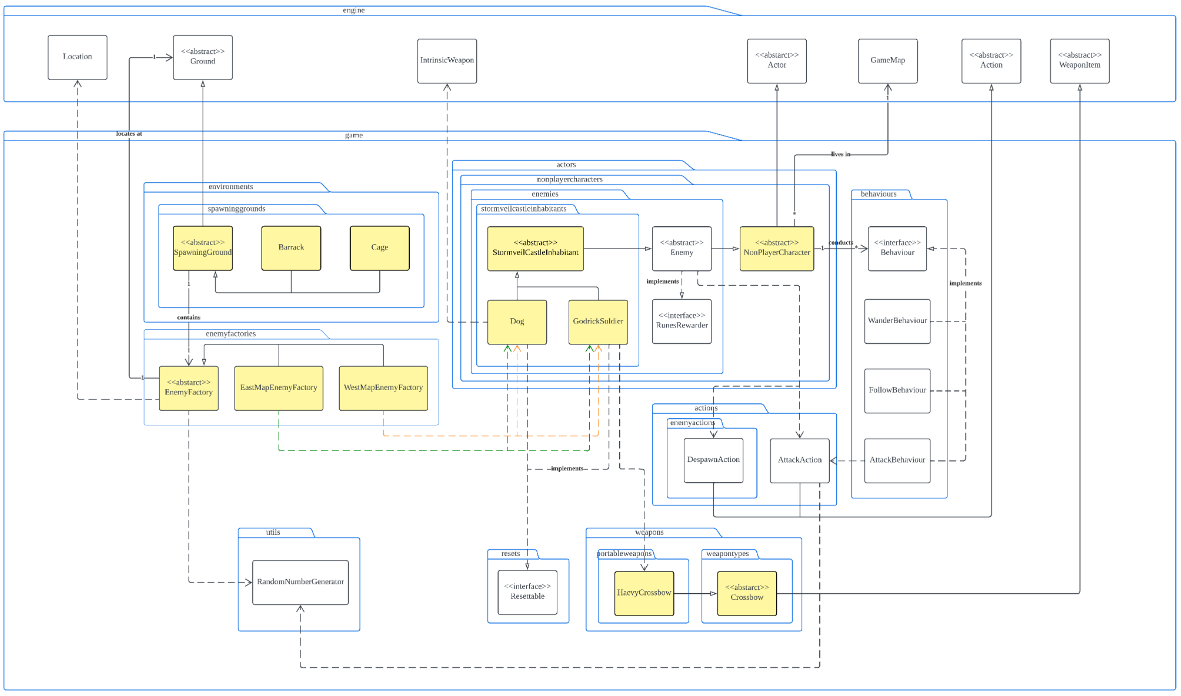
4. Created new game map classes “Limgrave”, “StormveilCastle”, “RoundtableHold” and “BossRoom” that inherit from “GameMap” class
   - OOP principles: SRP and DRY
   - Pro: Since each game map mentioned above can have different functionality, new classes for each of the game maps are created so that it is easier to handle in the “Application” class.
   - Pro: In “Application” class, an instance of the specific game map of its own type (eg: *Limgrave limgrave = new Limgrave()*) will be created so that any specific functionality from each game map class can be accessed easily such as the location of “GoldenFogDoor” which indicates the arrival location of actor from other game map.
   - Pro: By extending these classes from the “GameMap” class, each of these new classes do not have to again implement the common functionalities that a game map should have, thus preventing repetitive code in their classes.
   - Alternative: Instead of creating the new classes for each game map, we can define them in the application class itself.
   - Con of alternative: This can violate the SRP principle as the “GameMap” class will now have to handle all types of different game maps via polymorphism, especially different game maps that have their own functionality such as the game map that an actor can travel to from the current game map.
   - Future extension: If another new game map is introduced to the game (such as Peninsula map), a new class can be created and inherit the “GameMap” class to handle any unique functionalities.

5. “GoldenFogDoor” class has instance variables of a destination game map that the door can travel the actor to, the x-coordinate and y-coordinate of the arrival location at the destination game map.
   - Since this class's main functionality is to relate the travelling of an actor from one game map to another game map, each instance of “GoldenFogDoor” should know where they can link to when travelling the actor.
   - Pro: the location of the destination can be easily accessible via each instance of “GoldenFogDoor”
   - Future extension: This class can track the destination map and corresponding arrival location clearly whenever more traversals are introduced in the future.

6. Each game map class has an instance variable(s) of Location that saves the location of “GoldenFogDoor” which is the location where an actor can approach to travel to the other map.
   - Instead of adding the ground of “GoldenFogDoor” at some specific location of x-coordinate and y-coordinate in “Application” class, the value can be set in the constructor of each game map class and then to be retrieved via the use of engine code which is *gameMap.location.x()* and *gameMap.location.y()* when we are adding the ground. For example: *roundtableHoldMap.getDoorToLimgrave().setGround(new GoldenFogDoor(limgraveMap, limgraveMap.getDoorToRoundtableHold().x(), limgraveMap.getDoorToRoundtableHold().y()));*

- The above line of code demonstrates how we can make use of the engine code *.setGround()* to add a “GoldenFogDoor” into the game map. This would be able to relate different game map (which is the source and destination) easily via the new instance of “GoldenFogDoor”.
- For example, the above code indicates clearly with the source map and destination map when setting the “GoldenFogDoor” ground. The “RountableHold” game map will set a ground of “GoldenFogDoor” at the location that allow an actor to start travel; while the initialization of “GoldenFogDoor” indicates the destination of travelling reaches Limgrave game map at specific x-coordinate and y-coordinate location
- OOP principles: SRP
- Pro: The game map has given the promise that the location to travel to another map has been fixed and promised. This is to prevent other random “GoldenFogDoor” to be created at random locations.
- Future extensible: If there are more game maps introduced in the future (such as Peninsula map) and more intertraversal are allowed to happen (meaning we will probably have more “GoldenFogDoor” instances in a single game map), the system is more extensible as we can declare the location to travel to another game map in each game map class.

Game Requirement 2



1. Created new specialised grounds “Cage” and “Barrack” that inherit from abstract class “SpawningGround”.
    - OOP principles: OCP and DRY
    - Pro: Prevent repetitive code as the “SpawningGround” class already defined all the attributes and methods that are common to all types of specialised spawning ground.
    - Pro: This implementation follows OCP principle by creating new ground classes simply by extending the “SpawningGround” abstract class without modifying the existing code. All the special functionalities of a specialised spawning ground can be implemented in the specialised spawning ground class itself without modifying the existing abstract class “SpawningGround”.
    - Future extension: If there are any specific features or functionalities that need to be added to "Cage" or "Barrack" in the future, they can be easily implemented by making modifications only to their respective class, without impacting the implementation of the parent abstract class.

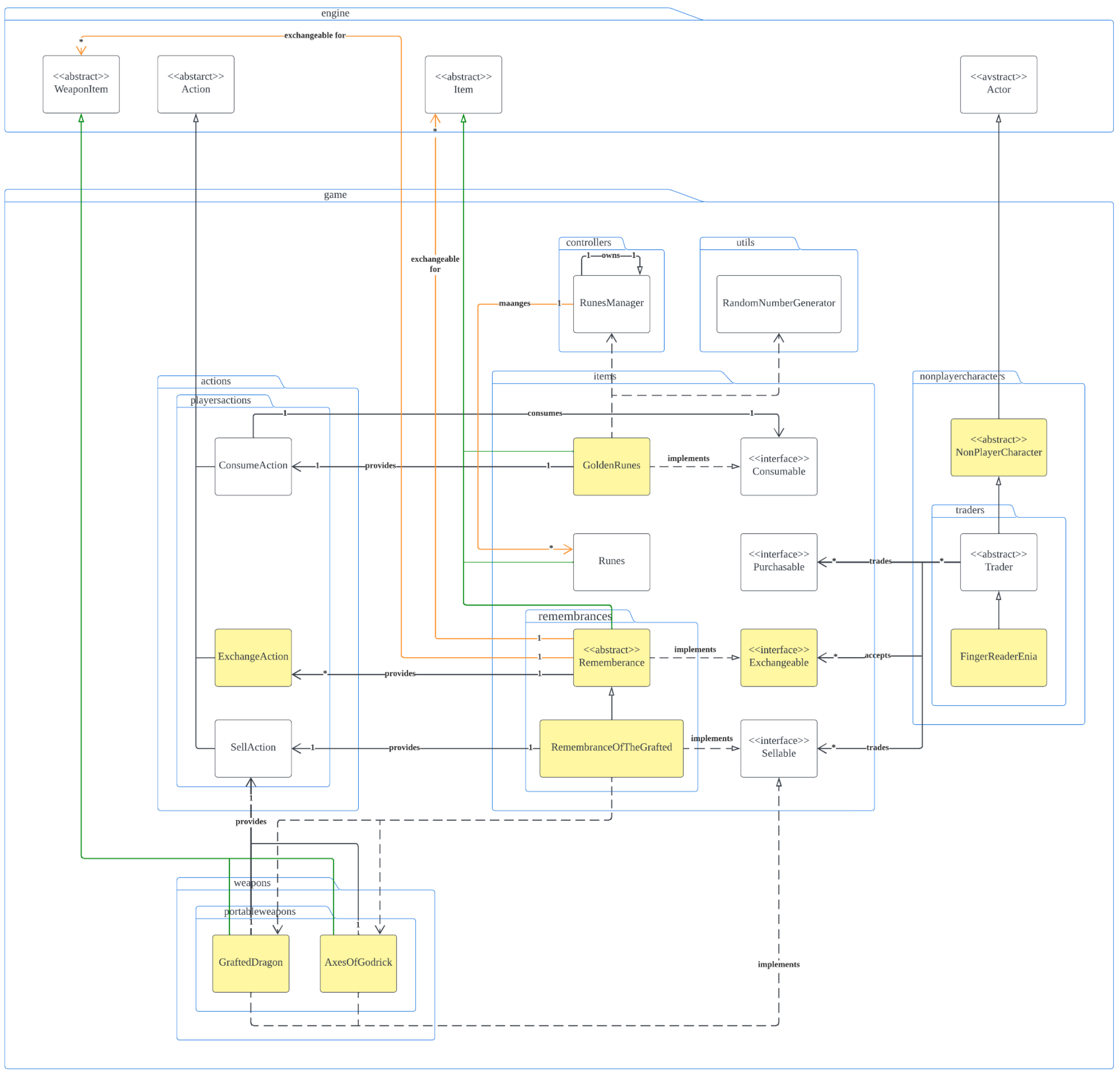
2. Created an abstract class “StormveilCastleInhabitants” extending “Enemy” abstract class and this new abstract class is extended by “Dog” and “GodrickSoldier”.
    - This new abstract class defines a new type of enemies in the game.
    - Decision: “Dog” and “GodrickSoldier” extend “StormveilCastleInhabitants” abstract class since they are both hostile creatures found in Stormveil Castle and share common attributes and behaviours.
    - Decision: Our group decided to add a new weapon item called “HeavyCrossbow” to the “GodrickSoldier”.
    - OOP principles: OCP and DRY

- Pro: Extending from this new abstract class means a particular enemy already has the characteristics provided by its type.
- Pro: By creating the “StormveilCastleInhabitants” abstract class, common attributes and methods can be defined once and reused by the “Dog” and “Godrick Soldier” enemy classes, avoiding code repetition.
- Pro: From the UML diagram’s point of view, it can be shown that both “Dog” and “GodrickSoldier” are enemies that can only be found in the “StormveilCastle” game map.
- Pro: This implementation follows OCP principle as “Dog” and “GodrickSoldier” classes are created simply by extending the “StormveilCastleInhabitants” abstract class without modifying any existing code. All the special functionalities of a specialised Stormveil Castle’s inhabitants can be implemented in the specialised inhabitant class itself (such as “Dog” and “GodrickSoldier”) without modifying the existing abstract class “StormveilCastleInhabitants”.
- Future extension: In the future, new enemies that belong to Stormveil Castle game map (for example “GodrickFighter”) can be easily implemented by creating new classes for them and extending these new classes from the “StormveilCastleInhabitants” abstract class without modifying the existing code.

3. Created new abstract class “Crossbow” and new weapon item “HeavyCrossbow”. “HeavyCrossbow” implements both “Purchasable” and “Sellable” interfaces.
    - Decision: A weapon “HeavyCrossbow” is created and extends the abstract class “Crossbow”, this “Crossbow” abstract class defines a new type of weapon items.
    - Decision: Similar to the previous discussion in Assignment 1, we designed to make the “Crossbow” to be an abstract class to prevent a direct instance of “Crossbow” from being created since this is not the real weapon that an actor should hold, but just a type (class) that groups all possible weapons in the game based on their features and similarities, which is “HeavyCrossbow” in this case.
    - Decision: “HeavyCrossbow” implements both “Purchasable” and “Sellable” interfaces as it can be purchased from traders and also sold to traders.
    - OOP principles: OCP and SRP
    - Pro: By creating the “Crossbow” abstract class, common attributes and methods can be defined once and reused by the “HeavyCrossbow” weapon item class, avoiding code repetition.
    - Pro: This implementation follows OCP principle as “HeavyCrossbow” class is created simply by extending the “Crossbow” abstract class without modifying any existing code. All the special functionalities of a Crossbow’s type weapon items can be implemented in the specialised classes themselves (such as “HeavyCrossbow” in this case) without modifying the existing abstract class “Crossbow”.
    - Future extension: If there is a new weapon type added into the game, a new weapon type abstract class can be created and let the specific weapon classes extend this new abstract class defining their type.

- Future extension: If a new “Crossbow” weapon is added into the game in the future (for example “LightCrossbow”), it can be implemented by creating a new class and extending this new class from the existing “Crossbow” abstract class without modifying it.

Game Requirement 3



1. "GoldenRunes" implements "Consumable" interface
    - OOP principles: OCP and ISP
    - Pro: As discussed in Assignment 2, the "Consumable" interface follows ISP as it promises its own method(s) that is/are to be implemented in every class that implements this interface, which is, the items that can be consumed.
    - Pro: OCP is followed as when "GoldenRunes" is consumed by the player, then a different effect will take place when its *consumedBy()* promised method is executed, and this effect is different from those we encountered previously.
    - Future extension: *(Reiterating the point from Assignment 2)* If we have another consumable item, say "PowerPotion", once drank by player, then he can be shielded and protected from any attacks in three consecutive rounds in the game. In this case, this "PowerPotion" will have to implement "Consumable" and code the implementation of "shielded" inside its own *consumedBy()* method, which is different from the effects of other consumable items.

2. Each “GoldenRunes” has an instance variable of type “ConsumeAction”
    - OOP principles: SRP
    - Pro: Each “GoldenRunes” instance is responsible for adding and removing the instance of “ConsumeAction” from their *allowableActions* ActionList, by overriding the *getDropAction()* in “GoldenRunes” class.
    - Pro: If an instance of “GoldenRunes” is added into the player’s item inventory, then in every round of the game, once the *tick()* method is executed, an instance of “ConsumeAction” (instance variable) will be added into this item’s *allowableActions* ActionList, to be printed out at the I/O console.
    - Pro: When this item is dropped by the player without being consumed, then in the “GoldenRunes”’s *getDropAction()* we can remove this instance of “ConsumeAction” (instance variable) from this “GoldenRunes” instance’s *allowableAction* ActionList, so that it would not be printed out at the I/O console.
    - Con: This association relationship between “GoldenRunes” and “ConsumeAction” may seem redundant and unnecessary, but this is the best and only workable alternative that we can think of as of now, in order to remove the “ConsumeAction” from the I/O console once it is dropped by the player without being consumed.
    - Alternative: Without keeping an instance variable of “ConsumeAction”, directly add a new instance of “ConsumeAction” into the “GoldenRunes” instance’s *allowableActions* ActionList if this item is in the player’s item inventory via the *tick()* method.
        - Pro: No association relationship between “GoldenRunes” and “ConsumeAction”
        - Con: Cannot remove the “ConsumeAction” from this item’s *allowableActions* if the player drops this item without consuming it. So, the I/O console may still print out this option for the player to choose although “GoldenRunes” has already been dropped.

3. We set the “GoldenRunes” instances directly at specific locations of the “Limgrave” and “StormveilCastle” game maps via the “Application” class.
    - Pro: Avoid unnecessary relationships between “GoldenRunes” and “Location”
    - Pro: Make use of engine code to put the “GoldenRunes” items on maps
    - Pro: Make the game system more flexible as the locations of “GoldenRunes” items can be adjusted and modified easily via the “Application” class
    - Pro: Easy in terms of implementation, reduce code complexity
    - Pro: Prevent “GoldenRunes” to be added onto the specific grounds that cannot be entered by the player. Since the list of strings representing the maps can be seen in the “Application” class, and therefore their x and y-coordinates can be calculated easily.
    - Con: “Application” class will become lengthy and somewhat turn into “God” class, but acceptable.

- Future extension: If we decide to add more “GoldenRunes” items at different locations, we can just simply add a new “GoldenRunes” instance and set its location to be added via the “Application” class, using *at()* and *addItem()* methods in the “GameMap” and “Location” classes provided by engine code.

4. For the purpose of simplicity, our group decided to not make the “GoldenRunes” resettable after the player died, so “GoldenRunes” does not implement the “Resettable” interface. Meaning, all the instances of “GoldenRunes” that have not been picked up by the player will not be removed from the game maps.

5. “GoldenRunes” keeps two final static variables indicating the minimum and maximum runes to be awarded to the player once consumed.
    - OOP principles: SRP and prevent excessive use of literals
    - Pro: static - these two variables are treated to be class variables as both the minimum and maximum runes amount to be awarded by every “GoldenRunes” instance are the same.
    - Pro: final - keep these two values at constant to prevent accidental modification to these values at somewhere else
    - Pro: SRP is followed as the “GoldenRunes” knows their own minimum and maximum runes awardable, so that a random number between these two amounts (accessed via the final static variables) can be retrieved in the *consumedBy()* method, and increment the player’s runes amount by this random number.
    - Pro: final static variables make the code more readable, intuitive and meaningful, as the other programmers can understand the meaning of these values easily.
    - Future extension: If there exists another consumable item that can as well award runes to the player, say “SilveryRunes”, but with a much smaller minimum or maximum awardable runes amount that are different from the “GoldenRunes”, then we can code another two final static variables inside this “SilveryRunes” class with their respective specified values.

6. “FingerReaderEnia” extends “Trader” abstract class and is added with both capabilities of *PROVIDE_EXCHANGE_SERVICE* and *PROVIDE_SELL_SERVICE*
    - OOP principles: OCP and DRY
    - Pro: “Trader” abstract class allows the game system to be easily extended as there could have more specific traders introduced into the game such as “FingerReaderEnia” in this case.
    - Pro: “FingerReaderEnia” can reuse the code and share all the functionalities given by the “Trader”, and therefore reduce repetitive code. This is because all the general functionalities of a trader have been coded in the “Trader” abstract class.

- Pro: The two elements of enumeration "*PROVIDE_EXCHANGE_SERVICE*" and "*PROVIDE_SELL_SERVICE*" are added to be capabilities of "FingerReaderEnia" to indicate that they allow the player to exchange "RemembranceOfTheGrafted" as well as sell their existing sellable weapons or items.
- Pro: In the *tick()* method within the classes of weapons and items, we are able to check if the player's surroundings has a trader with these capabilities.
- Future extension: *(Reiterating the point from Assignment 1)* If in the future there is a new trader introduced into the game such as "TraderMania", then this class can easily extend "Trader" and we can just add specific capability to this trader as if he allows the player to sell, purchase or exchange something.

7. Each "Trader" such as "MerchantKale" and "FingerReaderEnia" in this case, has an ArrayList of type "Exchangeable"
    - OOP principles: OCP, DRY and LSP
    - Pro: This association between "Trader" and "Exchangeable" allows every specific trader to know what are the things that they can accept from the player to be exchanged.
    - Pro: OCP is followed such that we can easily add or remove any "Exchangeable" items from the specific trader's "Exchangeable" ArrayList via their constructor, without having to modify the implementation done in other classes.
    - Pro: Different traders can allow the player to exchange different things as they keep a record of what they can accept in order to offer the corresponding "ExchangeAction" to the player.
    - Pro: DRY is followed because every trader subclass does not need to repeat the declaration and instantiation of relevant instance variables to keep record of their accepted "Exchangeable" items, as they will automatically inherit one empty ArrayList for these instance variables inside the "Trader" abstract class.
    - Pro: LSP is followed because either weapons or items can be added into this "Exchangeable" ArrayList of the trader, as long as they have implemented the "Exchangeable" interface.
    - Con: May violate SRP as this might not seem to be a responsibility of the trader.
    - Alternative: No "Exchangeable" ArrayList in the "Trader" class, but add capability for example "ACCEPT_REMEMBRANCE" to the trader.
        - Con: Might result in overuse of Enumeration as if a specific trader can accept a lot of things that can be exchanged, then we will be having a lot of elements in the "Status" class, which might affect the code readability and abuse the Java Enumeration.
        - Con: May be a bad design, as one specific trader will have a lot of capabilities.
        - Pro: Avoid association relationship between "Trader" and "Exchangeable" and may seem to follow SRP

8. Added a new interface called “Exchangeable”
    - OOP principles: OCP and ISP
    - Pro: We can make every item that can be exchanged to implement this interface, so that we can easily access the items or weapons that can be exchanged from an “Exchangeable” item or weapon via their own *getWeaponItemsToBeExchanged()* and *getItemsToBeExchanged()* methods, as promised by this interface.
    - Pro: We can have more than one item or other thing in the future that can be exchanged and can offer “ExchangeAction” to the player, if they have this “Exchangeable” item in their inventory.
    - Pro: The “ExchangeAction” can just call and execute the promised method *removeExchangeableFromInventory()* in the “Exchangeable” item, when this particular item has been exchanged with another thing by the player.
    - in any “Exchangeable” instance to perform the exchange functionalities, instead of just directly performing the effect given by “RemembranceOfTheGrafted”.
    - Future extension: If we have another exchangeable item, say “Nostalgia”, once carried by the player, then he can exchange a specific item or weapon such as “Uchigatana” from “FingerReaderEnia”. In this case, this “Nostalgia” will have to implement “Exchangeable” so that this interface can remind the us to add the items or weapons that can be exchanged from “Nostalgia” into their *weaponItemsToBeExchanged* and *itemsToBeExchanged* ArrayLists.

9. Each “Exchangeable” item has an ArrayList of type “Item” and “WeaponItem”
    - OOP principles: SRP and OCP
    - Pro: SRP is followed such that we made every “Exchangeable” item responsible for knowing what are the items that can be exchanged from it.
    - Pro: Reduce complex relationships in the “ExchangeAction” class.
    - Pro: OCP is followed such that we can easily modify the implementation of an “Exchangeable” item for example by adding or removing certain items or weapons that can be exchanged by this specific item in the future.
    - Alternative: In the “ExchangeAction”, it keeps a record of what are the corresponding items that can be exchanged from different “Exchangeable” items.
        - Con: May cause unnecessary and redundant relationships between “ExchangeAction” and “Item” or “WeaponItem” classes.
        - Con: Increase code complexity and reduce code maintainability, as we have to modify this “ExchangeAction” class every time a new “Exchangeable” item is introduced.
    - Future extension: If in the future we decided to add another weapon, say “Uchigatana”, that can be exchanged from the “RemembranceOfTheGrafted”, then we can just simply add one instance of “Uchigatana” weapon item into the corresponding ArrayList instance variable via the constructor of “RemembranceOfTheGrafted”.

10. Created “Remembrance” abstract class that implements the “Exchangeable” interface
    - OOP principles: OCP and DRY
    - Pro: Allows easy extension of the game system such as having more specific “Remembrance” items introduced into the game.
    - Pro: Every “Remembrance” subclass can reuse all the code and share the same functionalities provided by this abstract parent class.
    - Assumption: As of now, our group assumed that every “Remembrance” item can be exchanged for other things by the player. Therefore, we let this “Remembrance” abstract class implement the “Exchangeable” interface.
    - Pro: Prevent code repetition in every specialised “Remembrance” child class, such as the promised methods within the “Exchangeable” interface.
    - Alternative: Without creating this abstract class, but directly let the “RemembranceOfTheGrafted” implements “Exchangeable” interface
        - Con: The game system would be inextensible due to the possibility of having more specific “Remembrance” items in the future.
        - Con: As mentioned above, it will cause repetition of code.
    - Future extension: If in the future there is another “Exchangeable” item called “RemembranceOfNaturalborn” being introduced into the game, then this new item can just simply extend “Remembrance” abstract class to inherit its commonalities such as methods and instance variables. This item does not need to again implement the “Exchangeable” interface as its parent class already did.

11. “RemembranceOfTheGrafted” implements “Sellable”
    - OOP principles: OCP
    - Pro: Instead of letting the “Remembrance” abstract class implement this interface, we assumed not every “Remembrance” subclass can be sold, therefore following the OCP as the game system can be extended easily without making much modification to the existing classes.
    - Alternative: “Remembrance” abstract implements “Sellable”
        - Con: If there exists a new “Remembrance” subclass but cannot be sold, then we have to modify implementation of the existing class.
    - Future extension: If in the future there is another “Remembrance” item called “RemembranceOfNaturalborn” being introduced into the game, then this new item cannot be sold to any traders, then we just do not let this class implement the “Sellable” interface, without having to modify any implementation in the “Remembrance” abstract class.

12. “RemembranceOfTheGrafted” has an instance variable of type “SellAction”
    - Similar to Point 2 above, where we mentioned “GoldenRunes” has an instance variable of type “ConsumeAction”.
    - OOP principles: SRP

- To summarise, the main purpose of this implementation is to remove the sell action from being printed out in the I/O console once the "RemembranceOfTheGrafted" has been dropped by the player after being picked up. This can be done by overriding the *getDropActions()* in this class.
- Refer to Point 2 above for more information about the pros and cons.

13. Our group decided to simply add the "RemembranceOfTheGrafted" into the player's item inventory via the player's constructor, since we did not implement the optional part of this assignment, which is "GodrickTheGrafted" boss to be specific.

14. Created new class called "ExchangeAction"
    - OOP principles: OCP and SRP
    - Pro: SRP is followed because every action that can be performed by the player will be assigned to a specific class, in order to separate their special functionalities which will be executed if the player selects to perform a specific action.
    - Pro: We do not let either the FingerReaderEnia" or "Remembrance" classes themselves handle this exchange action's responsibility.
    - Pro: OCP can be achieved by overloading two different constructors for this "ExchangeAction" class, such that an exchange action can accept either an "Item" or "WeaponItem" to be exchanged with a specific "Exchangeable" item.
    - Pro: User can just pass in the "Item" or "WeaponItem" that is to be exchanged from an "Exchangeable" item into the constructor of "ExchangeAction", without having to worry about the parameter type as there are two overloaded constructors present. This resolves the downcasting issue as we did not have to downcast the "WeaponItem" to "Item" in order to match the parameter type of constructor.
    - Alternative: Use only one constructor that accepts "Item" parameter type to be exchanged from an "Exchangeable" type.
        - Con: Will have a downcasting issue as we have to downcast a "WeaponItem" instance to "Item" type in order to pass this item into the constructor of "ExchangeAction".
        - Con: Downcasting issue may also arise when we want to add this "Item" into the player's weapon item inventory if this "Item" is actually a "WeaponItem" type.
    - Future extension: Although as of now we only have "WeaponItem" that can be exchanged from "RemembranceOfTheGrafted", but if in the future this exchangeable item can as well be exchanged with an "Item" say "GoldenPotion", then we can just treat this implementation as normal by passing this item into the "ExchangeAction" class's constructor, since it can accept either an "Item" parameter type or "WeaponItem" parameter type, as discussed above.

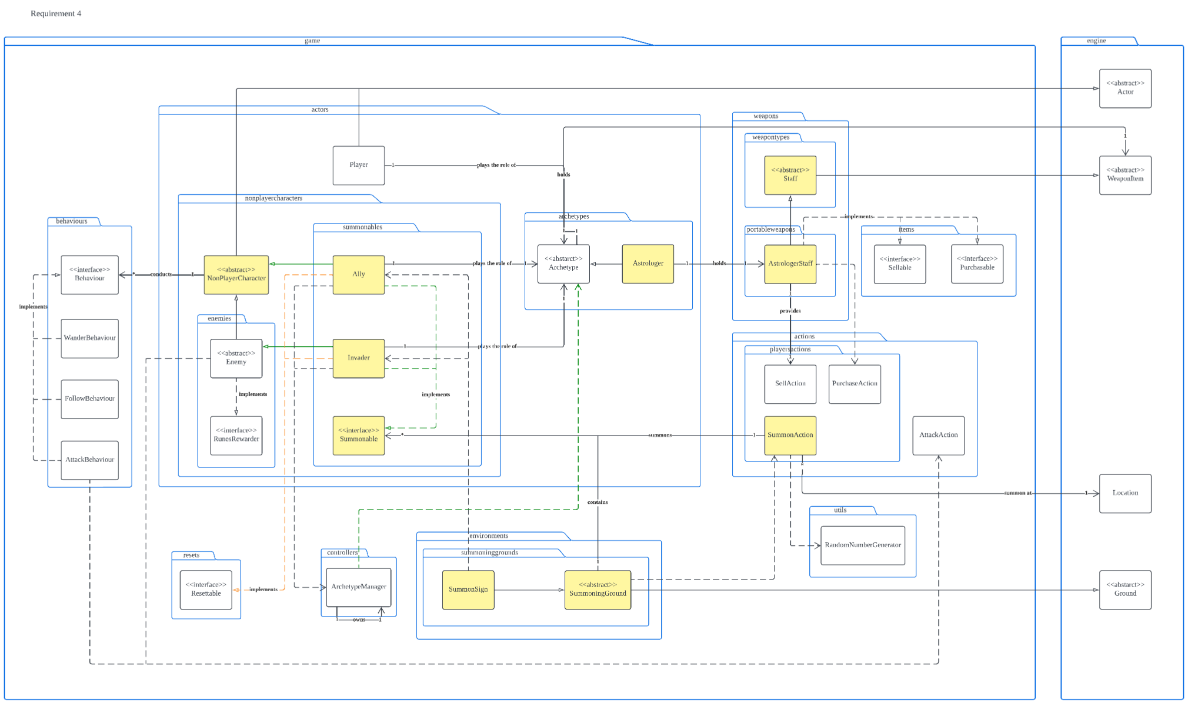
15. “Remembrance” class has an instance variable of ArrayList of type “ExchangeAction”
    - Similar to Point 2 and Point 12 above, where we mentioned “GoldenRunes” has an instance variable of type “ConsumeAction”, and “RemembranceOfTheGrafted” has an instance variable of type “SellAction”
    - OOP principles: SRP
    - To summarise, the main purpose of this implementation is to remove the exchange action from being printed out in the I/O console once the “Remembrance” item has been dropped by the player after being picked up. This can be done by overriding the *getDropActions()* in this class.
    - Refer to Point 2 above for more information about the pros and cons.

16. The instance of “ExchangeAction” will be added into the “Remembrance” item’s *allowableActions* ActionList via the *tick()* method
    - The item will check if the player’s surroundings has an actor (trader in this case) that can provide exchange service, via the *PROVIDE_EXCHANGE_SERVICE* capability as mentioned earlier.
    - OOP principles: SRP
    - Pro: Made this functionalities and responsibility of adding exchange action into the player’s selectable actions to be under the care of “Remembrance” item itself, but not the “FingerReaderEnia”.
    - Pro: Avoid downcasting issue as if this was to be done inside the “FingerReaderEnia” class, then we may need downcasting (downcast the “Item” to “Exchangeable” type) in order to pass this parameter type into the “ExchangeAction” class’s constructor.
    - Con: May violate DRY as we have to code the similar implementation in every “Exchangeable” item’s *tick()* method, but it solves downcasting issue and follows SRP
    - Alternative: Let “FingerReaderEnia” to handle the functionality of adding “ExchangeAction” into the player’s selectable actions.
        - Con: May violate SRP and encounter some downcasting issues as mentioned above.
    - Future extension: If we have another exchangeable item, say “Nostalgia”, then we will have to implement the similar code in this new class’s *tick()* method, without modifying implementation in other classes.

17. Created a new class called “SurroundingChecker” with a static method surroundingHasActorWithCapability(Location currentLocation, Status capability)
    - OOP principles: DRY
    - Pro: Since we always need to check if the player’s surroundings has a trader with different capabilities via the *tick()* method of different “Item” and “WeaponItem” classes, therefore we decided to add a new class with a static method that can be accessed in everywhere of the game system and therefore can be reused.

- Pro: We just have to pass in the player’s current location and the capability of the actor to check for, then this method will return a boolean indicating if there exists such an actor in the surroundings of the current location.
- Pro: Reduce a few lines of similar codes in the *tick()* method, making the code shorter and readable.
- Future extension: If a new trader with new capability is introduced into the game system, say “TraderMania” that has capability of *PROVIDE_WEAPON_UPGRADE_SERVICE*, then we can just pass this capability of parameter type Status into the static method *surroundingHasActorWithCapability()* to check for the player’s surroundings.

<u>Game Requirement 4</u>



1. Created a new class "Astrologer" as the new archetype. "Astrologer" class extends "Archetype" abstract class
   - Decision: Our group decided to add a new weapon item called "AstrologerStaff" to this new archetype "Astrologer"
   - OOP principles: DRY and OCP
   - Pro: Avoids repetition of code since all the specialized archetype classes share common attributes and methods
   - Pro: If there are any common changes to be made to the general implementation of behaviour of these combat archetypes, we could just modify the abstract/parent class without having to modify all those specialized archetype classes.
   - Future extension: If there are more archetypes introduced in the future, our system can be extended easily - create a concrete class of the new archetype and extend it from the "Archetype" parent class.

2. Created new abstract class "Staff" and new weapon item "AstrologerStaff". "AstrologerStaff" implements both "Purchasable" and "Sellable" interfaces.
   - Decision: A weapon "AstrologerStaff" is created and extends the abstract class "Staff", this "Staff" abstract class defines a new type of weapon items.
   - Decision: Similar to the previous discussion in Assignment 1, we designed to make the "Staff" to be an abstract class to prevent a direct instance of "Staff" from being created since this is not the real weapon that an actor should hold, but just a type (class) that groups all possible weapons in the game based on their features and similarities, which is "AstrologerStaff" in this case.

- Decision: “AstrologerStaff” implements both “Purchasable” and “Sellable” interfaces as it can be purchased from traders and also sold to traders.
- OOP principles: OCP and SRP
- Pro: By creating the “Staff” abstract class, common attributes and methods can be defined once and reused by the “AstrologerStaff” weapon item class, avoiding code repetition.
- Pro: This implementation follows OCP principle as “AstrologerStaff” class is created simply by extending the “Staff” abstract class without modifying any existing code. All the special functionalities of a Staff’s type weapon items can be implemented in the specialised classes themselves (such as “AstrologerStaff” in this case) without modifying the existing abstract class “Staff”.
- Future extension: If there is a new weapon type added into the game, a new weapon type abstract class can be created and let the specific weapon classes extend this new abstract class defining their type.
- Future extension: If a new “Staff” weapon is added into the game in the future (for example “SamuraiStaff”), it can be implemented by creating a new class and extending this new class from the existing “Staff” abstract class without modifying it.

3. Created a new abstract class “NonPlayerCharacter” extending “Actor” class, and this new class is extended by every non-player character (except the “Player”)
    - OOP principles: OCP and DRY
    - Pro: Made the game system more extensible and maintainable, since it promises that every NPC will have *behaviours*
    - Pro: The NPCs who currently have specific behaviours can still maintain their *behaviours* HashMap instance variable, but those NPCs without any specific behaviours as of now will have an empty *behaviours* HashMap, without affecting the current game system or functionalities.
    - Pro: Reduce code repetition to manually code a new *behaviours* HashMap instance variable in an NPC actor class.
    - Con: May result in multi-level abstraction
    - Future extension: Since we have extended “Trader” from “NonPlayerCharacter”, if in the future a “Trader” can have its specific behaviours to perform in the game, we can simply add instances of different behaviours into the “Trader” class’s *behaviours* instance variable.

4. Added capability *FRIENDLY_TO_PLAYER* to the new actor “Ally”
    - OOP principles: OCP
    - Pro: Created a difference between “Player” and “Ally” since they are both friendly actors, but with some differences as “Ally” is an NPC but “Player” is not.
    - Pro: OCP is followed as we do not need to modify the implementation of any existing classes. We just have to do something in the “Ally”’s

allowableActions() method to check if the surrounding actor is of the same type. If it is not of the same type (not having the capability of *FRIENDLY_TO_PLAYER*), then we add attack behaviour to “Ally”
- Future extension: If there exists a new friendly actor in the future, say “Elly”, then this “Elly” can also reuse the same capability *FRIENDLY_TO_PLAYER*, so that it will not attack other friendly actors.

5. The new actor “Invader” extends “Enemy” abstract class
    - OOP principles: OCP, DRY and prevent excessive use of literals
    - Pro: Final static variables are used to keep a record of the values of specific attributes for “Invader” in order to prevent excessive use of literals, as well as making the code more readable and intuitive.
    - Pro: “Invader” and “Enemy” share the similar characteristics and attributes (instance variables), for example, they have follow behaviour, following DRY principle as “Invader” do not have to repeat the similar code.
    - Pro: “Invader” does not have to implement the “RunesRewarder” interface since “Enemy” has already done it. So, “Invader” will automatically be a member of “RunesRewarder”, following DRY principle.
    - Pro: OCP is followed because if there are additional features or functionalities required by “Invader”, we can simply override or code extra implementation in their own methods.
    - Future extension: In the future, if “Invader” can have different behaviour such as “SummonBehaviour”, then we can add this behaviour into “Invader” *behaviours* in the *allowableActions()* method inside the “Invader” class.
    - Alternative 1: Created a new abstract class “Guest” to be extended by both “Ally” and “Invader”
        - OOP principles: OCP
        - Pro: Promise the functionality of choosing archetype at random in the constructor of this new class.
        - Con: May seem a bit redundant and unnecessary as it breaks DRY
        - Con: “Invader” has to repeat the same code block as in the “Enemy” abstract class since they have the similar characteristics.
    - Alternative 2: Added a new interface “ArchetypeOwner” to be implemented by both “Ally” and “Invader”.
        - OOP principles: ISP
        - Pro: Promise the method for choosing archetype at random in the constructor of this new class.
        - Con: May seem a bit redundant and unnecessary as it breaks DRY, while abusing the use of interface.
        - Con: Both “Ally” and “Invader” have to override the same method and code the same implementation, breaking DRY.
    - Decision: We decided not to go with this two alternatives as we assumed if we do remember to let a new actor class extend this “Guest” class or implement this “ArchetypeOwner” interface, then it will be the same as we remember to call “ArchetypeManager” to perform this functionality in the “Ally” and

"Invader" classes themselves. So, we do not actually need an abstract class or interface to promise this.

6. Coded another method in "ArchetypeManager" to choose archetype at random
    - OOP principles: DRY and SRP
    - Pro: Reuse existing method inside the "ArchetypeManager" to choose archetype at random, by just passing in a random number without letting the player input a number into the existing method.
    - Pro: High cohesion and low coupling - no unnecessary dependency relationship between every existing "Archetype" subclass with "Ally" or "Invader", but just dependency relationship between "Archetype" abstract class with "Ally" or "Invader" via polymorphism.
    - Pro: SRP is followed as we let "ArchetypeManager" handle its responsibilities of choosing and assigning archetype to an actor.
    - Future extension: In the future, there is a new actor who can have archetype, we can simply call "ArchetypeManager" to do its job for choosing and assigning archetype.

7. "Ally" and "Invader" implement "Resettable"
    - This is simply because both "Ally" and "Invader" will be removed from the game maps once the player died
    - OOP principles: OCP
    - Pro: As discussed earlier in Assignment 1, we do not let "Enemy" implement "Resettable" because it is not compulsory that every enemy can be reset once the player dies. So, in this case, we let every specialised actor who can be reset implement this interface in order to prevent modification to other existing classes if there exists one specific new enemy who cannot be reset.

8. "SummonSign" extends "SummoningGround" abstract class
    - OOP principles: OCP and DRY
    - Pro: Easy extension and maintainability of the game system
    - Pro: DRY is followed as we coded every commonality of a summonable ground inside the abstract class, therefore its subclasses can inherit all its coded attributes and functionalities, without code repetition.
    - Pro: OCP is followed as every specialised summoning ground can perform its special functionalities by overriding the methods in abstract class. Modification just needs to be done in the specific subclasses if required.
    - Future extension: In the future, if we have another new ground called "SummonSite" being introduced into the game, then we can simply just let this new class extend the "SummoningGround" abstract class, and code its special functionalities inside this new class.

9. Our group assumed that “SummonSign” can only be entered by the “Player”, otherwise if other actors other than the player can enter this ground, then the player will not be able to perform summon action once the ground is occupied by another actor.
    - This can be done by overriding the *canActorEnter()* method in the “SummonSign” class and checking actor’s capability *HOSTILE_TO_ENEMY*
    - Alternative: “SummoningGround” cannot be entered by other actor instead of just “SummonSign”
        - Pro: Avoid code repetition such that every specialised summon ground does not have to code the same implementation in the method *canActorEnter()*
        - Con: May violate OCP as we assumed not every summon ground can only be entered by the player
    - Future extension: If there exists another new summon ground called “SummonSite” in the future, where this ground can be entered by enemies and it allows enemies to summon a guest, then we do not have to modify the previous implementation done in “SummonSign” class.

10. Created new class for “SummonAction”
    - OOP principles: SRP
    - Pro: SRP is followed because every action that can be performed by the player will be assigned to a specific class, in order to separate their special functionalities which will be executed if the player selects to perform a specific action.
    - Pro: We do not let the “SummoningGround” handle this summon action’s responsibility.
    - Future extension: In the future, we can just call this “SummonAction” whenever the player is allowed to summon an actor once a certain condition is met, for example when the player is standing next to a new ground, say “SummonSite”.

11. Added a new interface “Summonable”
    - OOP principles: ISP
    - Pro: This interface has its responsibility of promising some required methods for every actor that can be summoned.
    - Pro: The promised method may help to remind us to always include the attribute of summon chance for the summonable actors.
    - Pro: The promised method can also help to reduce certain downcasting issues in the “SummonAction” class, especially when we want to check for a valid location in the surroundings of the summon ground to summon an actor. This is because we have to use the *canActorEnter()* and *addActor()* methods provided by the engine code. However these two methods can only accept “Actor” parameter type instead of “Summonable” parameter type. Therefore,

the promised methods in the “Summonable” member classes can help to solve this issue since every “Summonable” member class is also an “Actor”.

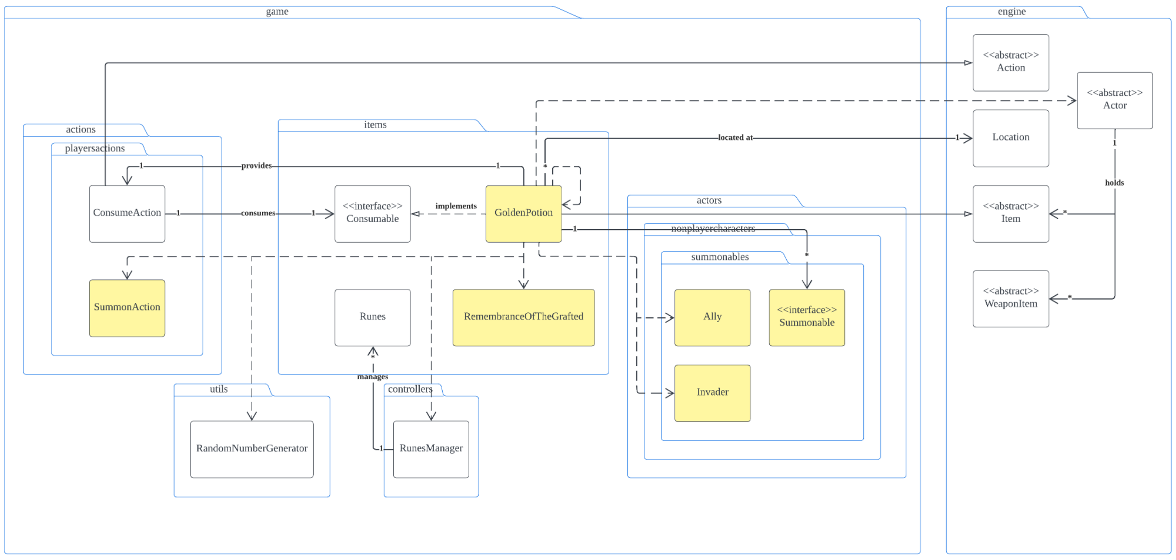
- Con: May result in certain repetitive code, somewhat breaking DRY, due to the same implementation in the methods promised by this “Summonable” interface in every “Summonable” member class.
- Con of not using this interface:
    - Have to always remember to include the required instance variables (for example the final static variable of SUMMON_CHANCE in this case).
    - Make the game somewhat inextensible especially if there exists a new actor that can be summoned, as we have to modify the existing implementation in the “SummonAction” to calculate the chance of summoning certain actors.
    - May cause downcasting issue as discussed above.
- Future extension: If there exists a new actor that can be summoned, say “Elly”, then we can let “Elly” implement “Summonable” interface to promise that the declaration of its final static variable of *SUMMON_CHANCE* as well as certain promised methods for solving downcasting issues as mentioned above.

12. The “SummoningGround” abstract class has a List of type “Summonable”
    - OOP principles: OCP, LSP and DRY
    - Pro: OCP is followed as different subclasses of “SummoningGround” can know what “Summonable” actors to be summoned by chance, by simply adding new instance of their accepted summonable actors into the *summonables* List initialised in the “SummoningGround” parent abstract class.
    - Pro: Prevent code repetition in every specialised “SummoningGround” child class, as they will automatically inherit all the coded attributes and methods inside the “SummoningGround” parent abstract class.
    - Pro: Made the game to be easily extensible, as no further modifications are required to be made in existing classes.
    - Pro: LSP is followed as this association relationship may help to reduce multiple unnecessary dependency relationship between “SummoningGround” with any other specialised “Summonable” actors such as “Ally” and “Invader” in this case.
    - Future extension: In the future if “SummonSign” can summon another actor, say “Elly”, then we can simply add an instance of “Elly” into this *summonables* List of type “Summonable” via the constructor of “SummonSign” class.

13. “SummonAction” also has a List of type “Summonable” which is passed by the “SummoningGround”
    - OOP principles: LSP and OCP

- Pro: Made the calculation of summoning actors by chance easy to be done while making the game system more maintainable and extensible.
- Pro: Calculation for summoning actors by chance can be done easily based on their respective summon chance. We do not have to specifically care about what are the “Summonable” specialised subclasses when doing the calculation.
- Pro: All we need to do is just use a simple mathematical concept to handle the different summon chance of different summonable actors within the *summonables* List of type “Summonable” which is passed by a specific “SummoningGround”.
- Con: This association relationship may seem unnecessary.
- Con of not having this association: We will have to hard code in order to calculate the summon chance of these two existing “Ally” and “Invader” only, making the game system not very extensible and maintainable. One example would be if there exists another “Summonable” actor called “Elly” who can be summoned by the “SummonSign”, then we will have to manually modify the hardcode of this calculation (for summoning actors by chance).
- Alternative: Keep a “SummoningGround” instance variable in the “SummonAction” so that “SummonAction” can as well get the same *summonables* List from this “SummoningGround”
    - Con of this alternative: The resulting UML diagram would then have a loop which seems illogical.
- Future extension: Even if there is an additional actor being added into the *summonables* List in the future (instead of only “Invader” and “Ally” as of now), our code implementation, specifically the calculation part, will still be able to handle it without any further modifications required.

<u>**Game Requirement 5**</u>



Creative idea: Introduced a “GoldenPotion” into the game, which allows the player to pick it up and consume it whenever he/she wants. Once “GoldenPotion” is consumed, one out of different effects may be caused to the player which consist of both buffs and debuffs, depending on the luck of the player. However, once the game is reset, all the instances of “GoldenPotion” that have not been picked up by the player will not be removed from the game maps (“GoldenPotion” is not resettable) and these buffs or debuffs will be removed from the player. Meaning, the game will be continuing as usual and the player will now be free from any buffs or debuffs caused by this “GoldenPotion”.

Buffs:

- Increase the player's maximum hit point
- Double the player's runes amount
- Double the attack damage from this point onwards
- Increase the number of usage for Flask of Crimson Tears by one
- Add RemembranceOfTheGrafted into actor's inventory
- Add another GoldenPotion at Player's surrounding
- Summon Ally in the player’s surroundings

Debuffs:

- Summon Invader in the player’s surroundings
- Poison the player and causes the decrease of hit point
- Set player's Runes to 0 (clear the player’s Runes)
- Remove AREA_ATTACK and SPECIAL_ATTACK capabilities of the weapons that are currently present in his weapon inventory
- Decrease the number of usage for Flask of Crimson Tears by one
- Clear the player's weapon inventory

OOP principles of this entire implementation: SRP and OCP

Pro of this entire implementation:

- Follow SRP principle as only the “GoldenPotion” class will be responsible for the random effect caused to the player once consumed.
- Follow OCP principle as this code can be easily extended by removing existing effects or adding new possible effects to the *switch case* block inside the *consumedBy()* method.

Con of this entire implementation:

- May cause the “GoldenPotion” class to become lengthy and somewhat unreadable.

Future extension: In the future if we decide to remove unnecessary effects or add more creative effects caused to the player, for example getting a “Bomb” item, then we can simply extend the game by modifying the *switch case* block inside the *consumedBy()* method in this “GoldenPotion” class, without changing the implementation in other classes.

1. Created new class “GoldenPotion” extending “Item” abstract class
    - OOP principles: OCP and DRY
    - Pro: “GoldenPotion” can inherit all the implemented functionalities and instance variables from the “Item” parent abstract class.
    - Pro: Reduce code repetition and the “Item” abstract class is open for modification as if the “GoldenPotion” wants to override certain coded methods provided by the “Item” class.
    - Future extension: If there exists a new item in the future, for instance “Bomb”, then this new item can again extend “Item” abstract class to inherit its instance variables and coded methods.

2. “GoldenPotion” implements “Consumable” interface
    - OOP principles: OCP and ISP
    - Pro: As discussed in Assignment 2, the “Consumable” interface follows ISP as it promises its own method(s) that is/are to be implemented in every class that implements this interface, which is, the items that can be consumed. This separates the responsibility and functionality of “Consumable” from the other interfaces.
    - Pro: OCP is followed as when “GoldenRunes” is consumed by the player, then a different effect will take place when its *consumedBy()* promised method is executed, and this effect is different from those we encountered previously.
    - Future extension: *(Reiterating the point from Assignment 2)* If we have another consumable item, say “PowerPotion”, once drank by player, then he

can be shielded and protected from any attacks in three consecutive rounds in the game. In this case, this “PowerPotion” will have to implement “Consumable” and code the implementation of “shielded” inside its own *consumedBy()* method, which is different from the effects of other consumable items.

3. “GoldenPotion” has association relationship with “ConsumeAction” class
    - OOP principles: SRP
    - Pro: Each “GoldenRunes” instance is responsible for adding and removing the instance of “ConsumeAction” from their *allowableActions* ActionList, by overriding the *getDropAction()* in “GoldenRunes” class.
    - Pro: If an instance of “GoldenRunes” is added into the player’s item inventory, then in every round of the game, once the *tick()* method is executed, an instance of “ConsumeAction” (instance variable) will be added into this item’s *allowableActions* ActionList, to be printed out at the I/O console.
    - Pro: When this item is dropped by the player without being consumed, then in the “GoldenRunes”’s *getDropAction()* we can remove this instance of “ConsumeAction” (instance variable) from this “GoldenRunes” instance’s *allowableAction* ActionList, so that it would not be printed out at the I/O console.
    - Con: This association relationship between “GoldenRunes” and “ConsumeAction” may seem redundant and unnecessary, but this is the best and only workable alternative that we can think of as of now, in order to remove the “ConsumeAction” from the I/O console once it is dropped by the player without being consumed.
    - Alternative: Without keeping an instance variable of “ConsumeAction”, directly add a new instance of “ConsumeAction” into the “GoldenRunes” instance’s *allowableActions* ActionList if this item is in the player’s item inventory via the *tick()* method.
        - Pro: No association relationship between “GoldenRunes” and “ConsumeAction”
        - Con: Cannot remove the “ConsumeAction” from this item’s *allowableActions* if the player drops this item without consuming it. So, the I/O console may still print out this option for the player to choose although “GoldenRunes” has already been dropped.

4. “GoldenPotion” has association relationship with “Location” class
    - Pro: Save the current location of the player so that certain effects involving “SummonAction” and addition of “GoldenPotion” in the player’s surroundings can be carried out.
    - Future extension: In the future, if we decide to add a new effect, for example turning a random ground in the player’s surroundings into “Cliff” ground type, then this instance variable of type “Location” may come handy and useful.

5. “GoldenPotion” has association relationship with “Summonable” interface and dependency relationship with “SummonAction” class
    - OOP principles: OCP, LSP and SRP
    - Decision: As for now, for the purpose of simplicity, our group decided to let “GoldenPotion” to only be able to summon either “Invader” or “Ally”, but our design does make it to be extensible in the future. Meaning, our design allows us to simply modify the *summonables* List of type “Summonable” in the “GoldenPotion” class by adding more or removing certain actors to be summoned by this “GoldenPotion” class.
    - Pro: OCP is followed as “GoldenPotion” can know what “Summonable” actors to be summoned by chance, by simply adding new instances of their accepted summonable actors into the *summonables* List initialised in the “GoldenPotion” class.
    - Pro: LSP is followed as this association relationship may help to reduce multiple unnecessary dependency relationships between “GoldenPotion” with any other specialised “Summonable” actors which are “Ally” and “Invader” in this case.
    - Pro: The dependency relationship between “GoldenPotion” and “SummonAction” follows SRP as we let “SummonAction” handle the functionalities of summon action whenever we want to summon an actor, preventing “GoldenPotion” itself from handling this responsibility.
    - Future extension: In the future if “GoldenPotion” can summon another actor, say “Elly”, then we can simply add an instance of “Elly” into this *summonables* List of type “Summonable” via the constructor of “GoldenPotion” class.

6. “GoldenPotion” has dependency relationship with “RunesManager” class
    - OOP principles: SRP
    - Pro: Allow “RunesManager” to carry out its responsibilities of increasing or decreasing the player’s runes amount.
    - Future extension: Any functionality related to player’s runes, we can just call “RunesManager” to handle it, for example deducting the player’s runes by a random amount.

7. “GoldenPotion” has dependency relationship with “Actor” class
    - Pro: Allow us to loop through the player’s weapon inventory and remove all of them (one of the debuffs)
    - Pro: Allow us to loop through the player’s weapon inventory and remove specific capabilities (*AREA_ATTACK* and *SPECIAL_ATTACK*) from the player’s currently holding weapons. These two capabilities had been used initially in Assignment 2 in which they were added to the specific weapon items that allows the actors to perform “AreaAttackAction” and certain special attacks such as “UnsheatheAction”.

- Pro: Allow us to increase and decrease the player’s hit points via the *resetMaxHp()* method provided in the engine code (one of the buffs and debuffs).
- Future extension: In the future, if we decide to add a new possible buff by adding the capability of *AREA_ATTACK* to every weapon item that is currently carried by the player.
- Future extension: In the future, if we decide to add a new possible buff by further increasing the player’s hit points to a bigger number, say 5000, then we can just reuse the *resetMaxHp()* method to perform this functionality.

8. “GoldenPotion” has dependency relationship with “RemembranceOfTheGrafted” class
    - This is simply because one of our buffs is to add a new instance of “RemembranceOfTheGrafted” directly into the player’s item inventory, even though the player has not yet defeated the boss “GodrickTheGrafted”.

9. “GoldenPotion” has dependency relationship with “GoldenPotion” class itself
    - This is simply because one of our buffs is to add a new instance of “GoldenPotion” directly onto a random exit around the player’s currently standing location, so that the player can come closer to the “GoldenPotion” and pick it up if the player wants to try their luck.

10. Increment and decrement of the number of usage of “FlaskOfCrimsonTears” can be done by adding new capabilities to the player, for example *INCREASE_CRIMSON_TEARS* and *DECREASE_CRIMSON_TEARS*.
    - Pro: In the *tick()* method inside the “FlaskOfCrimsonTears” class, we can check if the actor holding it (which is the player in this case), is having these two capabilities. If the boolean returns true, then we can perform the functionality that is intended to either increase or decrease the temporary maximum usage instance variable of this class.
    - Pro: Since we have another final static variable to keep the record of the maximum usage of Flask of Crimson Tears as specified in Assignment 2, once the game is reset and the *reset()* method of the “FlaskOfCrimsonTears” is executed, then its maximum usage will then be reset to the final static variable.
    - Future extension: In the future, if we decide to add a new possible buff by further increasing the player’s number of usage of “FlaskOfCrimsonTears” to a bigger number, say 5, then we can perform the similar functionality by adding another new capability to the player.

11. Doubling the player's attack can be done by adding the capability of *DOUBLE_ATTACK_DAMAGE* to the player.
    - OOP principles: OCP and DRY
    - Pro: This capability is reused here as initially in Assignment 2 this capability was added only to specific attack action that can perform double attack, which is "UnsheatheAction". *(As mentioned in Assignment 2)* "We just have to create a method to double the damage in the "AttackAction" class before it is executed". Therefore, now we just have to check if this actor who wants to perform an attack has the capability of *DOUBLE_ATTACK_DAMAGE*. If the boolean returns true, then we just have to simply call this particular method to double the damage, without adding any more additional implementation other than this.
    - Future extension: If a new enemy say "Monster" is introduced into the game in the future, where this new enemy can double its damage once certain condition is met, then we can just add this *DOUBLE_ATTACK_DAMAGE* capability to this enemy without modifying any further to the implementations of the other existing classes.

12. Addition of certain buffs or debuffs to the player once consumed can be done by adding corresponding capabilities to the player as mentioned above.

13. Removal of certain buffs or debuffs from the player once the game is reset can be done by removing the corresponding capabilities via the player's *playTurn()* method. After checking if the player has certain capabilities caused by consuming "GoldenPotion", then we will be removing these capabilities obtained from the "GoldenPotion" from the player if the boolean returns true.