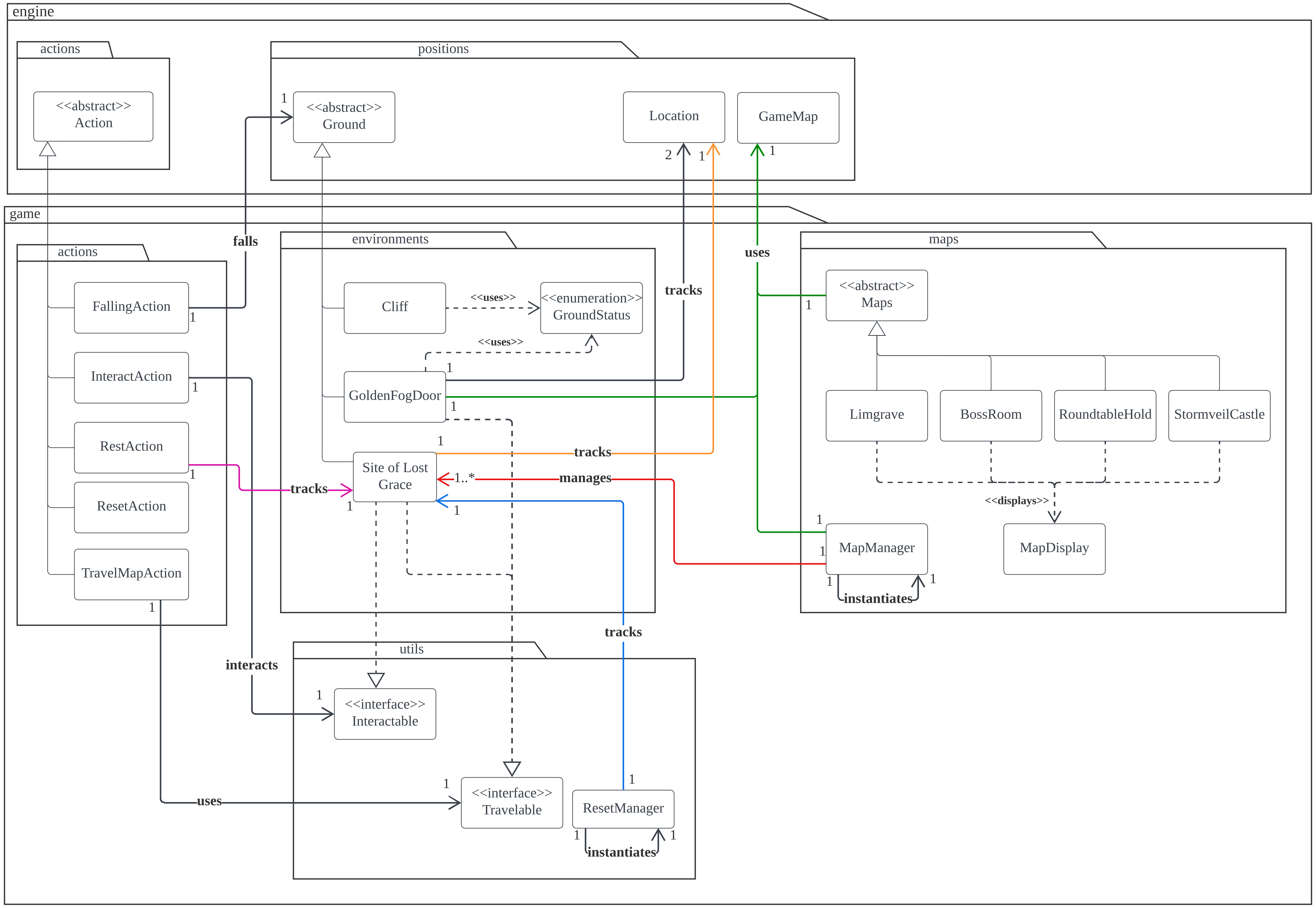
Group: MA_Lab04_Group5
Members: Bryan Wong, Po Han Tay, Wah Yang Tan

https://lucid.app/lucidchart/08951644-0380-497f-b11e-b97da00814c9/edit?invitationId=inv_0435f6cd-bbca-4e36-a5f4-c2adc77241dc

Optional Requirements we did:
- Req 1.B Fast Travel
- Req 3.A Godrick the Grafted
- Req 3.B.3 Site of Lost Grace

Optional Requirements we didn't do:
- Req 2.C Heavy Crossbow
- Req 3.A Grafted Dragon dropping fire
- Req 4.B Astrologer Staff


engine
actions
<<abstract>> Action
positions
<<abstract>> Ground
Location
GameMap
game
actions
FallingAction
InteractAction
RestAction
ResetAction
TravelMapAction
environments
Cliff
<<enumeration>> GroundStatus
GoldenFogDoor
Site of Lost Grace
<<uses>>
<<uses>>
maps
<<abstract>> Maps
Limgrave
BossRoom
RoundtableHold
StormveilCastle
MapManager
MapDisplay
<<displays>>
utils
<<interface>> Interactable
<<interface>> Travelable
ResetManager
falls
tracks
tracks
tracks
tracks
uses
uses
manages
interacts
instantiates
instantiates
1
2
1..*

Design Rationale (Requirement 1)

A. New grounds

1. Cliff

The new **Cliff** class inherits from the **Ground** abstract class, where it has the capability "GroundStatus.FALLABLE" to indicate that **Actors** on the **Cliff** can fall to their deaths. To prevent other **Actors** from entering the **Cliff**, the "canActorEnter" method is overridden so that only **Actors** with the capability "Status.CAN_ENTER_CLIFF" can enter, which the **Player** has. By using these capabilities, it enables us to have easily extendable code such that if additional "fallable" grounds are added or if other **Actors** that can fall off the **Cliff** are added, they are only required to have these capabilities without modifying the base code, thus adhering to the OCP (Open-Close Principle) (Pros) (Future extension). However, the usage of multiple capabilities may make it difficult to keep track of which capabilities an **Actor** has or which ones are needed for a specific action, resulting in additional complexity and confusion in the future (Cons).

To implement the **Player** falling to his death, a **FallingAction** class is used, which inherits from the **Action** abstract class. Here, it checks if the **Ground** the **Actor** is standing on has the capability "Status.FALLABLE". If it does, similarly to the **DeathAction** class, the **Player** is moved to the previous location and then killed. This is to prevent the **Runes** from being dropped on the **Cliff**. The **FallingAction** is returned within the **Player's** "playTurn" method, executing it. This design choice obeys the SRP (Single-Responsibility Principle) as the class only has one responsibility - killing the **Actor** that fell to their death off a "fallable" **Ground**, which improves the maintainability of the code as it does not have numerous other responsibilities to manage (Pros). However, this violates the DRY (Don't Repeat Yourself) principle as besides the **Ground** being the culprit of the killing, the **FallingAction** is virtually indistinguishable from the **DeathAction**, resulting in duplicated code and redundancy (Cons).

2. Golden Fog Door

For the creation of different **GameMaps** in the **World**, a **Maps** abstract class is used which handles the creation of multiple **GameMaps**. The **Limgrave**, **RoundtableHold**, **Stormveil Castle** and **BossRoom** classes all inherit from the **Maps** abstract class. **MapDisplay** will be a class containing static final instance variables of each **Map** string to be used in each **Maps** subclasses to create their **GameMap**. Each **Maps** subclass will use the **FancyGroundFactory**, along with their respective **MapDisplay**, for the creation of their **GameMap**, which is then added to the **World** via the "addGameMap" **World** method. As **Limgrave** is the default spawning location of the **Player**, they are added to the **World** using the "addPlayer" method from the **World** class inside the **Limgrave** subclass constructor. These maps are then created manually in the **Application** class. This design choice follows the OCP as the **Maps** abstract class allows the creation of new **GameMaps** in the future without modifying the base code, making our code easily extendable (Pros) (Future extension). Although in the current state of the game, the **Player** shall always spawn in **Limgrave**, that may not be the case in the future. Manually forcing the **Player** to spawn in

Limgrave, reducing the flexibility and extendability of our code as future changes in the game will result in the modification of existing code, violating the OCP (Cons).

Moving on to the Golden Fog Door, it inherits from the **Ground** abstract class and implements the **Travelable** interface. The **GoldenFogDoor** class will set the **GameMap** destination, along with their respective **Location** destination, as instance variables. To indicate the **GoldenFogDoor** can be used to travel to other maps, a capability "GroundStatus.CAN_TRAVEL" is used. As only the **Player** can use and enter these **GoldenFogDoors**, the "canActorEnter" method is overridden to only allow **Actors** with the capability "Status.CAN_FAST_TRAVEL" to enter, which the **Player** has. The usage of enumeration capabilities follows the OCP concept, as any other future **Actors** that can use these **GoldenFogDoors** without modifying the **GoldenFogDoor** class. This shows our code being easily extendable as new **Actors** with fast travel capabilities can be added in the future without any modification of the base code being required (Pros) (Future extension). However, the saving of **Location** and **GameMap** as instance variables for the **GoldenFogDoor** may lead to tight coupling between the two, resulting in unexpected behaviours and errors in the future if one is changed, leading to harder code maintainability (Cons).

The creation and connection of **GoldenFogDoors** to their destination are handled in the **MapManager** singleton class. The "createGoldenFogDoors" method will manage the destination **GameMap**, creating a **GoldenFogDoor** at the specified **Location** by manually setting the **Ground** there to the **GoldenFogDoor**. Then the "connectGoldenFogDoors" method is used to set the **Location** destination of the **GoldenFogDoor** to the specified **Location inputted**. Finally, these methods are called in the **Application** class as the **Locations'** coordinates are hard-coded, thus to reduce the usage of "magic numbers" in the code, they are all handled within the **Application** class. The usage of the **MapManager** singleton enables us to create **GoldenFogDoors** for different **GameMaps** without modifying the **Maps** code and creation process, thus adhering to OCP as any future **GoldenFogDoors** can be easily added to the **GameMap** (Pros) (Future extension). However, the creation of multiple **GoldenFogDoors** in the **Application** class may result in redundant code repetition, leading to the violation of the DRY principle as each separate **GoldenFogDoor** has to be manually created and connected (Cons).

The **Travelable** interface implemented in **GoldenFogDoor** enforces the "travel" method for its subclasses. In this case, the **GoldenFogDoor** "travel" method will move the **Actor** to the **Location** destination in the other **GameMap**. The "travel" method is called inside the **TravelMapAction** class which inherits from the **Action** abstract class. The **TravelMapAction** takes in a **Travelable** instance and is created and returned in the **GoldenFogDoor** "allowableActions" method provided that the **Actor** has the "Status.CAN_FAST_TRAVEL" capability. Through this, it demonstrates that our code follows the OCP as any future addition of fast travelling objects will have to implement their own "travel" method from the **Travelable** parent class, thus allowing our code to be easily extendable without any modification (Pros) (Future extension). However, this will lead to duplication of code for **Travelable** subclasses with similar “travel” functionality, as every subclass of **Travelable** is forced to implement its own “travel” function. For example, if another **Ground** similar to **GoldenFogDoor** is added, they will have to implement their own "travel" method despite similar functionality. This will lead to the violation of the DRY principle, as redundant code is repeated (Cons).

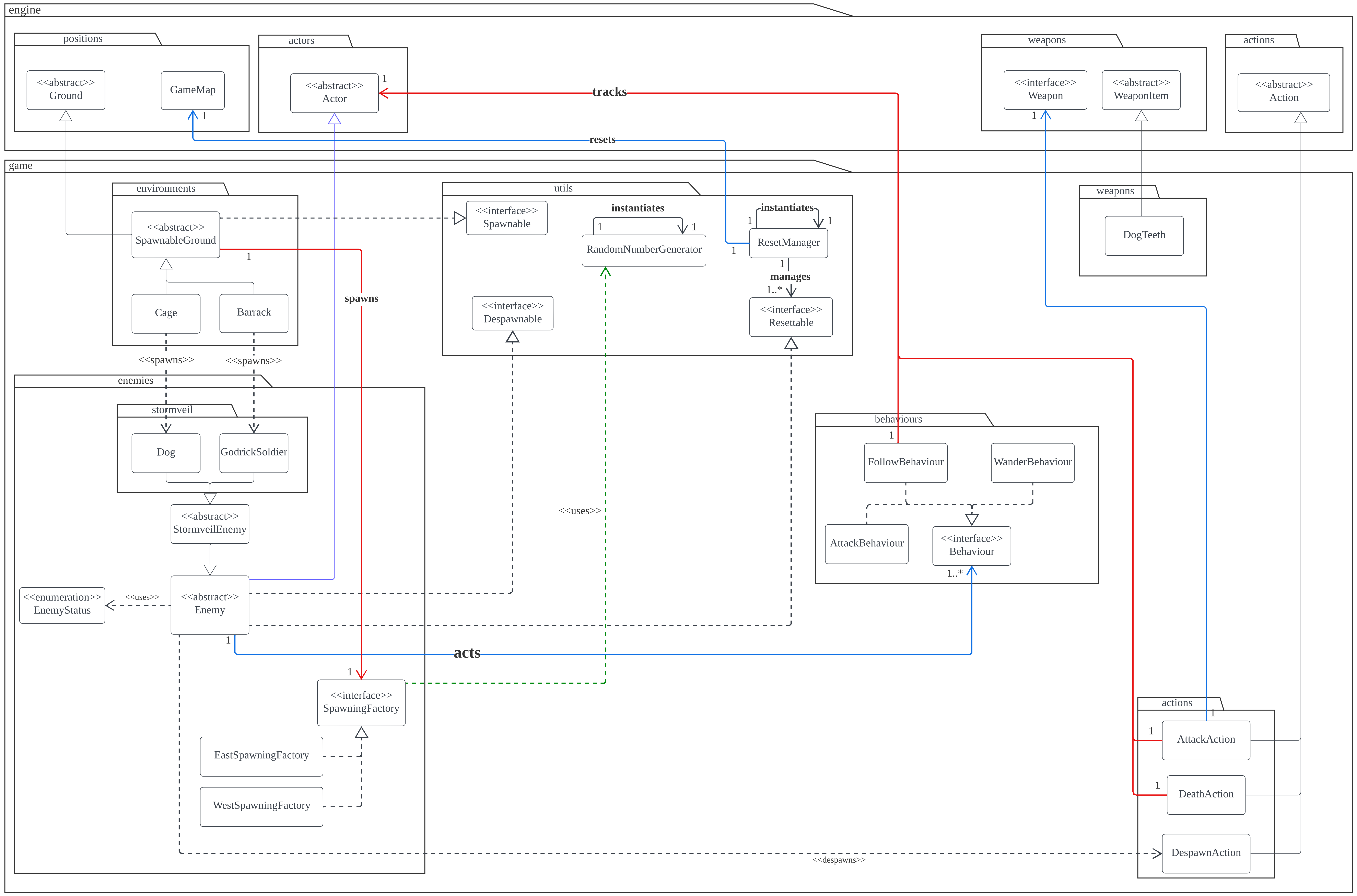
B. Fast Travel

The **SiteOfLostGrace** will be manually added into each **GameMap** via the "createSiteOfLostGrace" in the **MapManager** singleton class, where each **SiteOfLostGrace** will have a unique name for it based on which **GameMap** they are in. Then, the **SiteOfLostGrace** implements the **Interactable** interface, overriding the "interact" method from the parent class. The **SiteOfLostGrace** "interact" method will allow the **Player** to activate this specific **SiteOfLostGrace** via **InteractAction**. The **InteractAction** inherits from the **Action** abstract class, calling the inputted **Interactable** subclass "interact" method within the "execute" method. The **SiteOfLostGrace** "interact" method will add the "GroundStatus.ACTIVATED" capability to itself to indicate that this **SiteOfLostGrace** is activated, adding the **SiteOfLostGrace** instance to the MapManager "activatedSites" ArrayList which holds the List of activated **SiteOfLostGraces**. In the "allowableActions" method, a newly created **InteractAction** is returned if the **SiteOfLostGrace** instance does not have the "GroundStatus.ACTIVATED" capability, preventing any action on the **SiteOfLostGrace** if it’s not activated yet. Through this, our code will be easily extendable as any interactable objects in the future can implement the **Interactable** interface and implement their own "interact" method based on their functionality, therefore following the OCP as no modification of the base code is required to add new interactable objects or **Grounds** (Pros) (Future extension). This is shown later in Requirement 4. However, the usage of the global ArrayList "activatedSites" within the **MapManager** class violates the SRP as the **MapManager** has multiple responsibilities, including managing **GameMaps**, creating **SiteOfLostGrace** and **GoldenFogDoor** objects, tracking activated **SiteOfLostGraces** and many more. This may result in harder-to-maintain code, along with additional tight coupling between the two classes (Cons).

Furthermore, the use of **InteractAction** using an instance of **Interactable** instead of specifically **SiteOfLostGrace** follows the LSP (Liskov Substitution Principle) as it can be used interchangeably with other **Interactable** subclasses without affecting the behaviour of the program, thus making our code more maintainable (Pros). However, every single subclass of **Interactable** is forced to implement its own “interact” function, even though they may share similar functionality or code. This will lead to redundant code duplication, therefore infringing on the DRY principle.
When the **Player** rests on the **SiteOfLostGrace**, the instance of the **SiteOfLostGrace** rested is saved in the **ResetManager** via **RestAction**. When the **Player** dies and their "reset" method is called, the **Player** will be moved to the saved **SiteOfLostGrace Location**, thus respawning on the last **SiteOfLostGrace** rested. To implement the Fast Travel feature, the **SiteOfLostGrace** will implement the **Travelable** interface, similar to the **GoldenFogDoor**, except **SiteOfLostGrace** will travel to other activated **SiteOfLostGraces** instead. This

proves our code follows the OCP, as the **TravelMapAction** and other base codes are not modified and the **TravelMapAction** is easily extended for the **SiteOfLostGrace** fast travelling feature (Pros). In the "allowableActions" method, the "activatedSites" ArrayList is looped through, where every activated **SiteOfLostGrace** (except the current one) will have a **TravelMapAction** towards it created and added to the actions list only if the current **SiteOfLostGrace** is already activated.

The usage of multiple interfaces (**Interactable** and **Traveable**) follows the ISP (Interface-Segregation Principle) as we managed to avoid the creation of large interfaces that forces its subclasses to implement unnecessary functions (Pros). In the future, any additional item that can be **Interactable** and **Traveable** can also implement both interfaces, thus allowing the code to be easily extended without modifying any existing code, thus following the OCP (Pros) (Future extension). However, the reliance between **Player** respawning and **SiteOfLostGrace** may introduce tight coupling between the two, making it inflexible to new game functionalities in the future. If new respawning behaviour not related to **SiteOfLostGrace** is added in the future, it may result in bugs, making the code less maintainable (Cons).


engine
positions
<<abstract>> Ground
GameMap
actors
<<abstract>> Actor
weapons
<<interface>> Weapon
<<abstract>> WeaponItem
actions
<<abstract>> Action
tracks
resets
game
environments
<<abstract>> SpawnableGround
Cage
Barrack
utils
<<interface>> Spawnable
instantiates
RandomNumberGenerator
instantiates
ResetManager
manages
<<interface>> Despawnable
<<interface>> Resettable
weapons
DogTeeth
spawns
<<spawns>>
<<spawns>>
enemies
stormveil
Dog
GodrickSoldier
<<abstract>> StormveilEnemy
<<enumeration>> EnemyStatus
<<uses>>
<<abstract>> Enemy
<<uses>>
acts
behaviours
FollowBehaviour
WanderBehaviour
AttackBehaviour
<<interface>> Behaviour
<<interface>> SpawningFactory
EastSpawningFactory
WestSpawningFactory
actions
AttackAction
DeathAction
DespawnAction
<<despawns>>
1
1..*

<u>Design Rationale (Requirement 2)</u>

A. Grounds

Similar to Assignment 2, the environments (**Cage** and **Barrack**) in this game were created using an abstract class called **SpawnableGround** that implements the **Spawnable** interface, which requires the 2 environments to implement a "spawn" function for spawning their respective enemies. We did this to adhere to the DRY (Don't Repeat Yourself) principle. Since **Cage** and **Barrack's** classes have the same spawning functionality, they will inherit from the **SpawnableGround**, enabling them to share functionality and reduce code duplication, such as the said "spawn" function (DRY) (Pros).

However, the **SpawnableGround** class has some disadvantages. One is that it's rigid and is only for **Ground** types. In the future, if an **Item** that can spawn **Enemies** is added, it will not be able to inherit from **SpawnableGround** and reuse its "spawn" function (Cons). To tackle this debilitating issue, as stated before, we have decided to create a **Spawnable** interface. Using this, any future "spawnable" things, such as the **Item** example mentioned above (Future extension), will be able to implement that interface and its own "spawn" function, without having to modify any existing code thus following the OCP (Pros). The downside of the **Spawnable** interface is that every single one of its subclasses has to implement its own "spawn" function, even if they share the same spawning functionality (except **SpawnableGround** subclasses), resulting in duplicated code which violates DRY (Cons).

Furthermore, **SpawnableGround** follows the SRP (Single Responsibility Principle) as the **SpawnableGround** abstract class and its children classes all handle ground that spawns **Enemies** only. Thus they only have one responsibility - spawning. Due to this, our code is more maintainable, as any changes to the spawning functionality will not induce side effects in other areas, reducing the risk of unintentional bugs (Pros). On the other hand, **SpawnableGround** having a single responsibility results in limited functionality for additional features. Because of this, to handle new **Ground**-related features, such as traps or hazards for the **Player**, we may have to create separate and specific classes just for those features (Cons).

The "spawn" function in the **SpawnableGround** relies on the **SpawningFactory** interface class to add the **Enemy** to the map. This obeys the DIP (Dependency Inversion Principle) as by depending on a high-level abstraction (**SpawningFactory** interface) instead of a concrete class, the **SpawnableGround** is decoupled and unaware of the specific details on how the **Enemy** is added to the **GameMap**. This makes the **SpawnableGround** more maintainable, as the **SpawnableGround** is isolated from changes in the **GameMap** (Pros). There are some disadvantages to this, however. One such is an increase in complexity. Adding this abstraction layer can make the code harder to understand and more complex, especially if one is unfamiliar with how the code functions (Cons).
The **SpawnableGround** "tick" method will create an instance of **WestSpawningFactory** or **EastSpawningFactory** based on which half of the **GameMap** is currently located by checking the **Location**. For example, the **GameMap** "xRange" method is used to get the

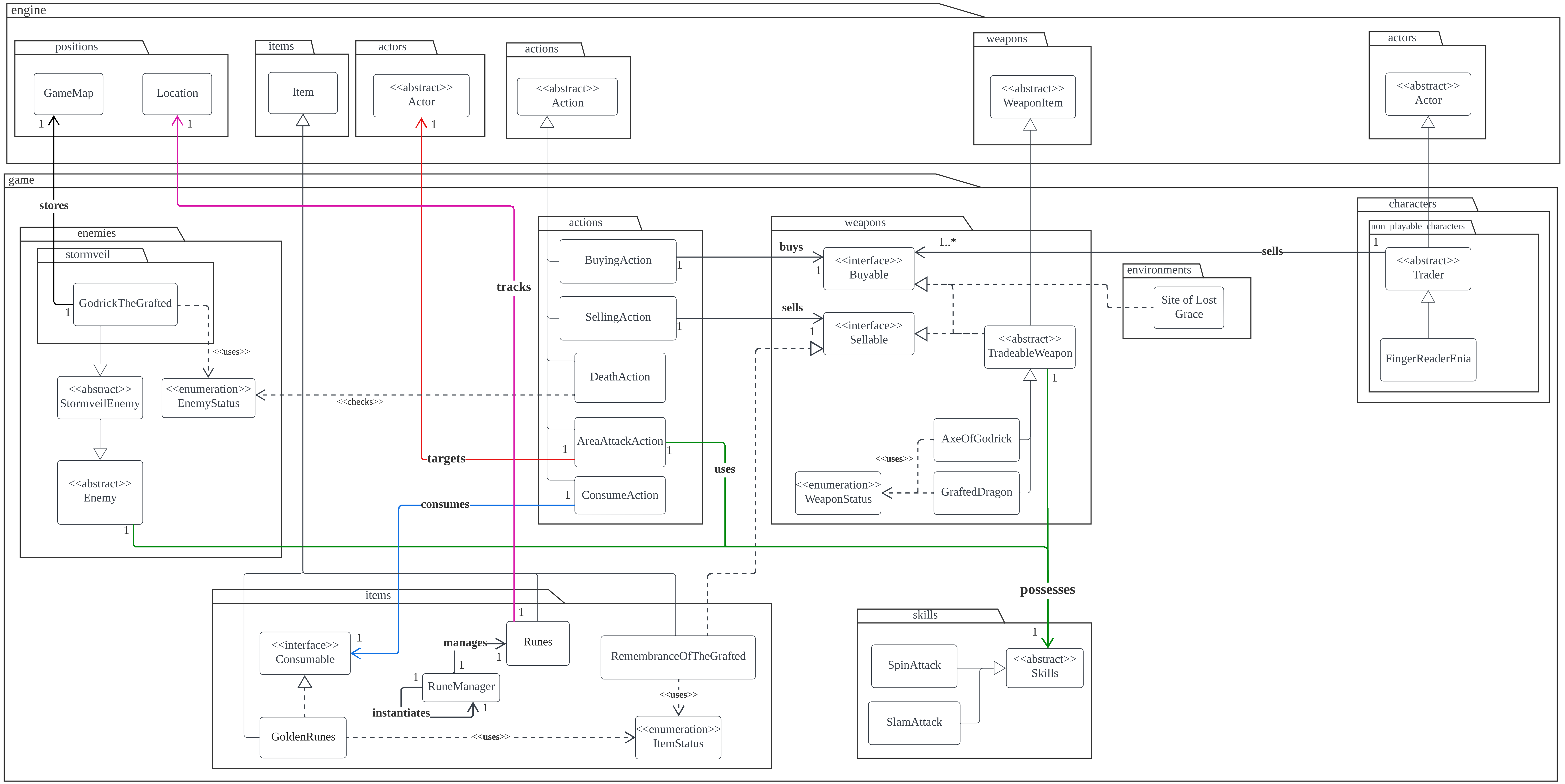
maximum x-coordinate of the **GameMap**. This value is divided by 2 to split the map into half and then compared with the current x-coordinate of the **SpawnableGround**. If the **SpawnableGround** x-coordinate is on the West side of the **GameMap**, a **WestSpawningFactory** will be created. Otherwise, an **EastSpawningFactory** instance will be created instead. Then, the "spawn" function mentioned earlier will be called.
The **SpawningFactory** enforces multiple methods its subclasses must have. For example, they will have methods like "createUndead" which both **WestSpawningFactory** and **EastSpawningFactory** must implement. **WestSpawningFactory** will create and add a **HeavySkeletalSwordsman** to the **GameMap**, whereas **EastSpawningFactory** will create and add a **SkeletalBandit** to the **GameMap**. All other **Enemies** in Assignment 2 will have the same design too. In the case of **Cage** and **Barrack**, the "createDog" and "createSoldier" methods will be enforced. However, since there are no specific spawns on the West or East side of the **GameMap** for these two **SpawnableGrounds**, both the **WestSpawningFactory** and **EastSpawningFactory** will create and add the same **Enemy**. For example, the **Cage**, the **WestSpawningFactory** and **EastSpawningFactory** "createDog" method will both create and add a **Dog** to the **GameMap.** Similarly for the **Barrack**, the "createSoldier" method will both create and add a **GodrickSoldier** to the **GameMap**. This usage of interfaces obeys OCP as new **SpawningFactory** subclasses can implement the interface, implementing all spawning methods. For example, if a **NorthSpawningFactory** is added in the future, all it needs is to just implement the **SpawningFactory** interface without any modification required. Therefore showing that this design choice is easily extendable for future usage (Pros) (Future extension). However, as the "create" methods are done in the **SpawningFactory** interface, any addition to new **Enemies** in the future will require the modification of the **SpawningFactory** interface, violating the OCP as the code is not easily extendable for new **Enemies** (Cons).
In addition, the **RandomNumberGenerator** class will be used in the **SpawningFactory's** "spawnEnemy" function to determine the chances of enemies spawning for each respective environment. Also, **RandomNumberGenerator** is a singleton, allowing any other functions that require random number generation to use this, reducing further code duplication (DRY) (Pros). As a result, it's not recreated every time the class is invoked to generate random numbers, reducing the verbosity of creating parameterized type instances (Pros). As **SpawningFactory** depends on **RandomNumberGenerator**, it creates a tight coupling between the two classes. As a result, the DIP (Dependency Inversion Principle) is violated, which results in a less maintainable code, as the **SpawningFactory** is not isolated (Cons).

B. Enemies

Similar to Assignment 2, the **Barrack** will spawn only **GodrickSoldier** and the **Cage** will spawn only **Dog**. These two classes will inherit from the **StormveilEnemy** abstract class which inherits from the **Enemy** abstract class. As in Assignment 2, the **Enemy** already implements the **Despawnable** interface and the "despawn" method, meaning **GodrickSoldier** and **Dog** will inherit the "despawn" method for them to despawn from the map every turn using the **DespawnAction**. Similarly, the **Enemy** already implements the **Resettable** interface, hence the two newly added **Enemies** will also inherit the "reset" method without any modification required. This proves our code from Assignment 2 is easily extendable, as two new **Enemies** can be added without modifying the base code, therefore adhering to the OCP (Pros). However, the multi-level inheritance will introduce rigid hierarchies which will limit flexibility and potentially violate OCP for future **Enemies** types if they do not conform to the existing classes (Cons).

To prevent the two **Enemies** from attacking each other, the **StormveilEnemy** class has the capability "EnemyStatus.NOT_HOSTILE_TO_STORMVEIL_ENEMY", preventing the creation of **AttackAction** between the two just like in Assignment 2. The **GodrickSoldier** and **Dog** will have the same behaviours as all other **Enemies** in Assignment 2: **AttackBehaviour**, **FollowBehaviour** and **WanderBehaviour**. For the dropping of **Runes**, the two **Enemies**, similar to Assignment 2 **Enemies**, will have a "runeMin" and "runeMax" variables to randomly generate a "runeCount" for the **Runes** object in their inventory.

For their Weapons, the **Dog** has their own specialised **WeaponItem** called **DogTeeth** which has 101 damage and 93% attack accuracy. As we decided not to implement the optional Ranged Weapons like Heavy Crossbow, we gave the **GodrickSoldier** the **Club** from Assignment 2. Through this, we have proven our code can be easily extended for new **Enemies** and types without any modification required, showing our design follows OCP (Pros). However, the **Enemies** may have repeated similar functionality, resulting in additional unnecessary code duplication, violating the DRY principle (Cons).


engine
positions
GameMap
Location
items
Item
actors
<<abstract>> Actor
actions
<<abstract>> Action
weapons
<<abstract>> WeaponItem
actors
<<abstract>> Actor
game
stores
enemies
stormveil
GodrickTheGrafted
<<uses>>
<<abstract>> StormveilEnemy
<<enumeration>> EnemyStatus
<<abstract>> Enemy
tracks
actions
BuyingAction
SellingAction
DeathAction
<<checks>>
AreaAttackAction
targets
ConsumeAction
consumes
uses
weapons
buys
<<interface>> Buyable
sells
<<interface>> Sellable
<<abstract>> TradeableWeapon
AxeOfGodrick
<<uses>>
<<enumeration>> WeaponStatus
GraftedDragon
environments
Site of Lost Grace
characters
non_playable_characters
sells
1..*
<<abstract>> Trader
FingerReaderEnia
possesses
items
<<interface>> Consumable
manages
Runes
RuneManager
instantiates
RemembranceOfTheGrafted
<<uses>>
<<enumeration>> ItemStatus
GoldenRunes
<<uses>>
skills
SpinAttack
<<abstract>> Skills
SlamAttack

<u>**Design Rationale (Requirement 3)**</u>

A. Godrick The Grafted

GodrickTheGrafted inherits from **StormveilEnemy**, and like all other **Enemy** subclasses, has an instance variable depicting their despawning chance. As **GodrickTheGrafted** cannot despawn, the “despawnChance” variable is set to 0 so they will never despawn. This proves our code obeys the OCP as no modification of the parent classes or original code is needed to add in this new **Enemy**. As a result, our code is easily extendable for any future **Enemies** added (Pros) (Future extension). To prevent **GodrickTheGrafted** from being removed from the map whenever a game reset occurs, the “reset” method inherited from **Enemy** is overridden to reset **GodrickTheGrafted** health back to the maximum, then manually moving them back to its starting position in the Boss Room. **GodrickTheGrafted** will also have the same behaviours as all other **Enemies** in Assignment 2: **AttackBehaviour**, **FollowBehaviour** and **WanderBehaviour**. **GodrickTheGrafted** saves a **GameMap** instance variable which describes which **GameMap** they are in, which in this case it's the **BossRoom**. During the "reset" call, it uses the said saved **GameMap** to move back to its starting **Location**. This creates tight coupling between the two, resulting in difficulties maintaining the code in the future as changes in the **GameMap** could have a direct impact on **GodrickTheGrafted** functionality (Cons).

On creation, an **AxeOfGodrick** is added to its weapon's inventory, which is their weapon during the first phase. A capability "EnemyStatus.FIRST_PHASE" is used to indicate which phase **GodrickTheGrafted** is currently in. The **AxeOfGodrick**, similar to other **Weapons** in Assignment 2, has a **Skills** instance variable which represents which skills it has. In this case, it will be a **SpinAttack** skill which contains all required data attributes (damage, accuracy etc) to be used during an attack. Then, the "getSkill" method from the engine is overridden to return an **AreaAttackAction**. The **AttackBehaviour** will automatically randomly choose an **AttackAction** or **AreaAttackAction**, with a chance of 50% each, using the **AxeOfGodrick** without any modification required as the "getSkill" method is used, thus proving our code is easily extendable and adhering to OCP (Pros). However, this design choice may violate the DRY principle as every single **Weapon** which has similar special skills will require the overriding of the "getSkill" method despite similar features, thus resulting in redundant repeated code (Cons). **AxeOfGodrick** will inherit from **TradeableWeapon**, thus implementing the **Sellable** interface via the parent class. Through this, **AxeOfGodrick** can also be sold to the **Trader**.

Within the "playTurn" method, it checks for the current health of **GodrickTheGrafted**, and whether it's <= 50% of its maximum health. If so and **GodrickTheGrafted** has the capability "EnemyStatus.FIRST_PHASE", which indicates that it's still at the first phase, trigger the second phase by removing the **AxeOfGodrick** from the inventory and replacing it with a **GraftedDragon**, removing the "EnemyStatus.FIRST_PHASE" capability to signify its no longer in that phase. The **GraftedDragon** is similar to the **AxeOfGodrick**, in that the "getSkill" method is overridden to return an **AreaAttackAction**. The difference, however, is that they have a **SlamAttack** skill instead. To prevent the dropping of the **Weapons** when

GodrickTheGrafted is slain, the "togglePortability" method is used to turn the **Weapon** "portable" instance variable to false. We decided not to do the optional requirement of making the **GraftedDragon** drop flames on its surroundings. After the phase check, the parent "playTurn" (super.playTurn) method is called to execute the normal behaviour it inherits from the **Enemy** class. Through this, we reduced the need for duplicated code as we reuse the parent methods instead of rewriting them again, thus following the DRY principle (Pros). Similar to the **AxeOfGodrick**, the **GraftedDragon** will also inherit from **TradeableWeapon**, inheriting the "sell" method enabling itself to be sold. **AxeOfGodrick** and **GraftedDragon** will not be in the buyable HashMap the **Trader** has, preventing it to be bought by the **Player**. While this design reduces code duplication, it violates the ISP principle as **TradeableWeapon** will both implement **Buyable** and **Sellable**, resulting in **AxeOfGodrick** and **GraftedDragon** implementing the "buy" method despite it not being purchasable by the **Player** (Cons).

During creation, a **RemembranceOfTheGrafted**, which inherits from the **Item** abstract class, is added to their item inventory, allowing it to be dropped once **GodrickTheGrafted** is felled. **GodrickTheGrafted** will have a capability "Status.DEMIGOD", which during the **DeathAction**, will create a **SiteOfLostGrace** on their corpse once they are killed. This design choice will obey the OCP as any future demigods added with similar features when killed, can reuse this by having the "Status.DEMIGOD" capability (Pros) (Future extension). However, this may introduce rigid and inflexible functionality, as any future demigod which has different **DeathAction** functionality will be unable to reuse this feature, resulting in a violation of OCP if modification is required (Cons).

The "runeMin" and "runeMax" will be set to 20000 to allow the **Player** to receive exactly 20000 **Runes** when **GodrickTheGrafted** is killed by them. As **RemembranceOfTheGrafted** can be sold, they will implement the **Sellable** interface, enforcing their own "sells" method. In the "tick" method which is called when the **Item** is carried, it checks for adjacent **Actors** with the capability "Status.TRADEABLE". If any is found, a **SellingAction** is added to the "allowableActions" List, allowing the **RemembranceOfTheGrafted** to be sold. Through this, we have sufficiently demonstrated how our design choice follows ISP as **RemembranceOfTheGrafted** only needs to implement the interfaces it requires (such as **Sellable**), avoiding the need to implement "dud" methods (Pros). However, this will result in needless code duplication for all **Items** that have similar selling functionality, as every **Sellable Item** are required to implement its own "sell" method, resulting in the violation of the DRY principle (Cons). To implement the exchanging feature between **AxeOfGodrick**, **GraftedDragon** and **RemembranceOfTheGrafted**, both the **AxeOfGodrick** and **GraftedDragon** "buy" methods are overridden in which they will accept an **Item** with the capability "Status.EXCHANGEABLE" (which **RemembranceOfTheGrafted** has) instead of costing **Runes**, allowing the **Player** to exchange a **RemembranceOfTheGrafted** in their inventory for the **AxeOfGodrick** or **GraftedDragon**. If the **Player** does not have any **RemembranceOfTheGrafted**, the exchange will not occur.

B. More Selling and Purchasing

1. Golden Runes

GoldenRunes inherits from the **Items** abstract class, implementing the **Consumable** interface to signify it can be consumed. During creation, the **RandomNumberGenerator** singleton is used to randomly generate a "goldenRuneCount" between 200 to 10000. Through this, we have managed to avoid redundancy via composition and reusing methods, adhering to the DRY principle (Pros). On the other hand, this will result in a dependency between **GoldenRunes** and **RandomNumberGenerator**, leading to a tight coupling between the two. As a result, the maintainability of the code decreases (Cons).

Since **GoldenRunes** can be consumed by the **Player**, it implements the **Consumable** interface, implementing its own "consume" method which will increase the **Player Runes** "runeCount" based on the randomly generated "goldenRuneCount" earlier. Without modifying the base code, we have shown our code is easily extendable as new **Consumables** are added without affecting the behaviour of the base code, thus demonstrating that the OCP is followed (Pros). Every turn, the "tick" method is used to add the **ConsumeAction** into the **GoldenRunes** "allowableActions" List provided it's empty. Once the **ConsumeAction** is executed, it is removed from the List. However, this method will result in rigid code as it is under the assumption that **GoldenRunes** will only have one **Action** in the "allowableActions" List. If any more **Actions** are added in the future, this feature will break, resulting in the violation of OCP as modification will be required to account for this (Cons).

To randomly scatter the **GoldenRunes** across **Limgrave** and **StormveilCastle**, the **MapManager** is used where the "generateGoldenRunes" method will randomly generate 3 to 5 **GoldenRunes** to add to the **GameMap**. **Grounds** with the capability "GroundStatus.CAN_GENERATE_GOLDEN_RUNES" (in this case will be **Floor** and **Dirt**) are the only **Ground** allowed to have **GoldenRunes** generated on them. This is done to prevent **GoldenRunes** from spawning in a **Wall**, **Cliff** or other unreachable **Grounds**. Then, the **GameMap** "getXRange" and "getYRange" inbuilt engine methods are utilised to get the maximum and minimum x and y coordinates. The **RandomNumberGenerator** "getRandomInt" method is then used to generate a random x and y coordinate between the two ranges given. Next, the **GoldenRunes** will be added to that randomly generated **Location** based on the x and y coordinates provided that there aren't any **Items** already at that **Location.** Through this, any future **GameMaps** which can generate **GoldenRunes** that have varying ranges of x and y coordinates will be able to utilise this "generateGoldenRunes" method without any modification required, therefore following the OCP (Pros) (Future extension). However, the "generateGoldenRunes" method has to be manually called in the **Application** for each **GameMap** that requires it, resulting in duplicated repetitious code, leading to the violation of the DRY principle making code less maintainable (Cons).

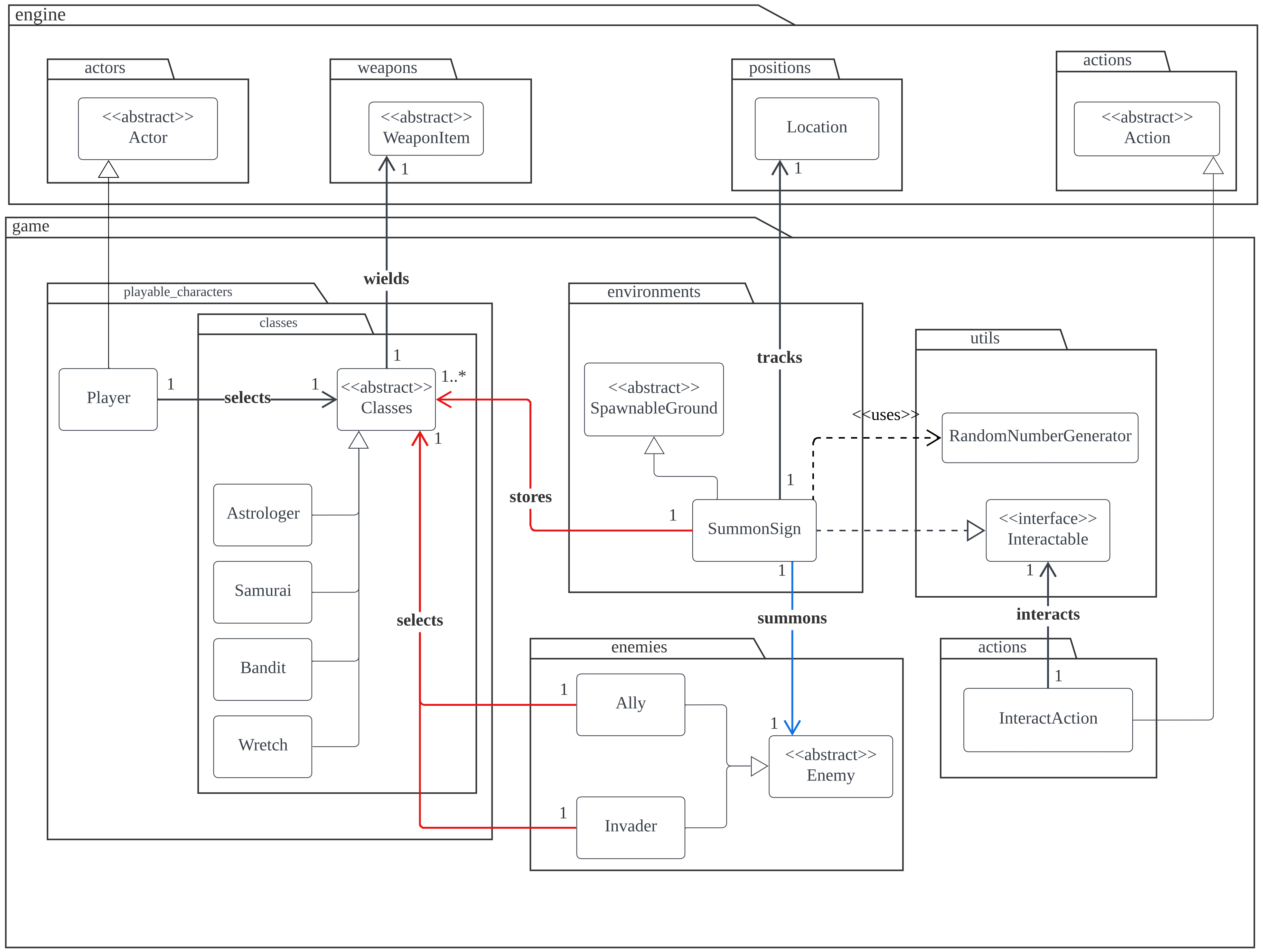
2. Finger Reader Enia

The **Trader class** has been made an abstract class, with **MerchantKale** and **FingerReaderEnia** inheriting from it. This is done to reduce code duplicity as both **Traders** allow the **Player** to sell their **Weapons** to them, adhering to the DRY principle (Pros). However, the usage of inheritance will lead to inflexibility as if new **Traders** are added in the future that does not allow **Player** to sell their **Weapons** to them, modification of the **Trader** abstract class is required, resulting in the violation of OCP (Cons). Both **Traders** will also have a "buyable" HashMap, containing all the **Buyables** purchasable by the **Player**. When an **Actor** with the capability "Status.TRADEABLE" is adjacent, which the **Player** has, **BuyingActions** are created for all **Buyable** instances within the "buyable" HashMap inside the "allowableActions" method. This is done in future scenarios of new **Buyable** items that are not **WeaponItems.** If in the future, a **Buyable Item** is added, no modification of the **Trader** abstract class is required to implement this functionality, only adding the **Buyable** to the HashMap, thus showing our code is easily extendable and following OCP (Pros) (Future extension).

FingerReaderEnia is manually added to the **RoundtableHold GameMap** via the **Application** class. As **AxeOfGodrick** and **GraftedDragon** can be "bought"/exchanged from **FingerReaderEnia**, both of them are added to the "buyable" HashMap in the constructor of **FingerReaderEnia**. The exchanging function is already explained earlier in **A. Godrick The Grafted**.

3. Site of Lost Grace

SiteOfLostGrace will implement the **Buyable** interface as the **Player** can "buy" upgrades for their health. **SiteOfLostGrace** implements multiple interfaces (**Interactable**, **Travelable** and **Buyable**), revealing that our code follows the ISP and OCP as **SiteOfLostGrace** can implement methods they need, without implementing unnecessary methods, in addition to no modification required to implement this functionality. Any future objects that have similar features can also implement the same interfaces, showing our code is easily extendable (Pros) (Future extension). The "buy" method is overridden to allow the **Player** to upgrade their health by 48 each time, with an increment in the "buyingPrice" by 100 each time an upgrade is made. As the "buyingPrice" is shared between all **SiteOfLostGraces**, the "buyingPrice" is stored in the **ResetManager** singleton. The "getBuyingPrice" and "setBuyingPrince" implemented from the **Buyable** interface will be overridden to set and return the "buyingPrince" inside the **ResetManager** singleton. Through this, all different **SiteOfLostGraces** will share the same "buyingPrice". However, the dependency between **SiteOfLostGrace** and **ResetManager** will introduce tight coupling between the two classes, potentially violating the SRP and reducing code maintainability (Cons).


engine
actors
<<abstract>> Actor
weapons
<<abstract>> WeaponItem
positions
Location
actions
<<abstract>> Action
game
playable_characters
classes
Player
selects
<<abstract>> Classes
1..*
wields
Astrologer
Samurai
Bandit
Wretch
selects
stores
environments
<<abstract>> SpawnableGround
tracks
SummonSign
<<uses>>
utils
RandomNumberGenerator
<<interface>> Interactable
summons
interacts
enemies
Ally
Invader
<<abstract>> Enemy
actions
InteractAction
1

Design Rationale (Requirement 4)

A. New Role (Astrologer)

Similar to Assignment 2, the code follows the OCP by allowing the addition of a new class, **Astrologer**, which inherits from the **Classes** abstract class. The **Astrologer** will have a starting HP of 396. This design decision proves the code's extendability and flexibility, as it does not require modifications to the parent class or the original code. By adhering to OCP, the code can easily accommodate the inclusion of new **Classes** in the future (Pros) (Future extension). Since **Classes** require each child class to wield a **WeaponItem**, we gave the **Astrologer** class the **Great Knife** from Assignment 2, as we decided not to implement the optional Ranged Weapons like Astrologer Staff. However, the creation of separate **Classes** for every class in the game, even though they may have similar functionality or features, may result in the DRY principle being violated as the code will be repeated constantly (Cons).

B. Allies/Invaders

1. Summon Sign

In line with Assignment 2, the environment (**SummonSign**) inherits from a **SpawnableGround** abstract class as they can spawn **Allies** or **Invaders**. This abstract class implements the **Spawnable** interface, which enforces the implementation of a "spawn" function responsible for spawning an **Ally** or **Invader**. This approach was chosen to ensure adherence to the DRY principle, minimising code duplication. This also promotes code reusability and modularity, as future environments can implement the "spawn" function according to its specific requirements without duplicating common functionality (Pros) (Future extension).

Within the **SummonSign** class, there is an ArrayList instance variable that stores **Classes** objects. It allows the **Ally** or **Invader** to individually select their class from the ArrayList. Moreover, this design proves that we adhered to LSP as it ensures that any class inheriting from **Classes** can be stored in the ArrayList. This leads to the interchangeable usage of the subclasses and newly introduced subclasses due to the extensibility of **Classes** (Pros) (Future extension). However as the addition of new **Classes** is done via the **SummonSign** constructor, if any new **Classes** are added in the future, modification of the **SummonSign** is required, which results in the violation of the OCP as the code is not easily extendable (Cons).

Not only that, **SummonSign** implements the **Interactable** interface, overriding the “interact” function from the parent class. This “interact” method will allow the **Player** to summon the **Ally** or **Invader** via **InteractAction**. It proves that this design obeys OCP, due to its extensibility without any modification as any **Interactable** object in the future can just implement the interface and the “interact” method (Pros) (Future extension). **InteractAction** is a subclass of the **Action** abstract class, invoking the **SummonSign** “interact” method within its “execute” method. The **SummonSign** “interact” method will use the **RandomNumberGenerator** to choose a class from the ArrayList of **Classes** objects and which entity (**Ally** or **Invader**) to be summoned with both being 50% chance. Once the entity is chosen, it will assign the randomly chosen class to the entity, setting their hitpoints and starting weapon based on the respective class chosen. Each **SummonSign** will have an instance variable, saving its current **Location**. This will be used to spawn the **Ally** or **Invader** at a random adjacent location that they can normally spawn using the “canActorEnter” method. Meaning, **Grounds** that they can’t enter will be **Grounds** that they cannot spawn in too. This is done to prevent **Allies/Invaders** from spawning in **Walls**, **Cliffs** etc. Furthermore, **Actors** are prevented from entering the **SummonSign** so that they cannot block the **Player** from summoning. The disadvantage of such a design choice is the limitations in the choosing of which **Enemy** to spawn. As **Ally** and **Invader** are both hardcoded to have a 50% chance to spawn, future summonable **Enemies** added will require modification of the **SummonSign** to summon them, resulting in the violation of OCP as the code will not be extendable (Cons).

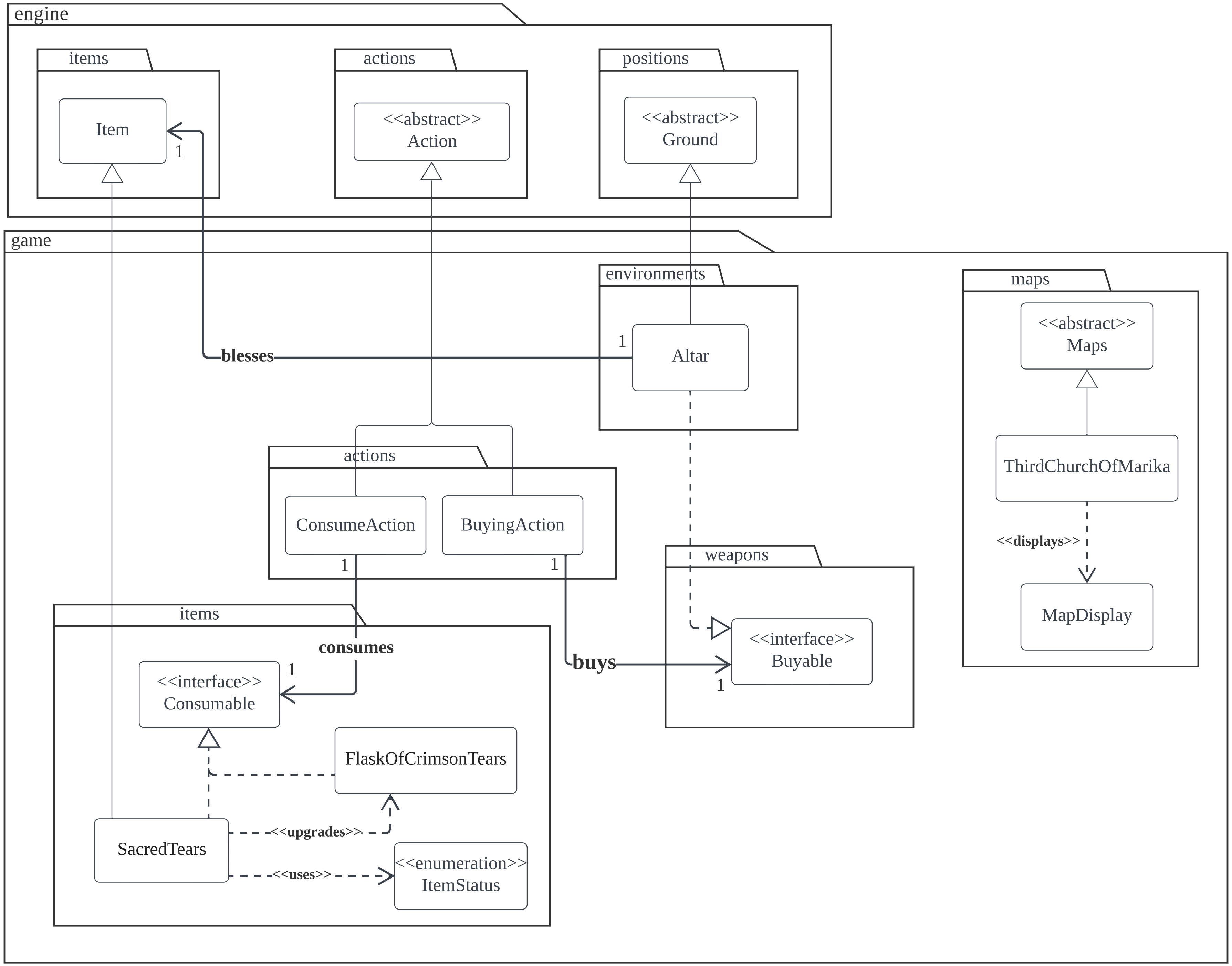
SummonSign, like all **SpawnableGrounds**, will have the **SpawningFactory** instance. Similar to **Cage** and **Barrack**, in that there are no differences in **Enemies** spawned based on the side of the **GameMap**, the **SpawningFactory** will have a "createSummonable" function which takes in an **Enemy** instance to add to the **GameMap**. The **SummonSign** "spawn" function will be called in the "interact" function, passing in the randomly selected **Enemy** to be summoned (**Ally** or **Invader**) to the "createSummonable" method which then adds it to the map based on the random **Location** given.

2. Ally/ Invader

Ally and **Invader** will both inherit from the **Enemy** abstract class, with their "despawningChance" variable being 0 as both of them cannot despawn. To prevent **Allies** from attacking each other, they have the capability "Status.NOT_HOSTILE_TO_ALLY", which will be used in the "allowableActions" method to prevent the creation of **AttackAction** between the two. Similarly, **Invader** will have the "Status.NOT_HOSTILE_TO_INVADER" capability to do the same. Since **Allies** cannot attack the **Player**, the "Status.HOSTILE_TO_PLAYER" capability is removed from the **Ally** to prevent them from attacking the **Player**. Through this, we have confirmed our code is easily extendable, thus following the OCP as no modification of the base code is required to change the attacking behaviour for the **Ally** (Pros).

As **Allies** cannot follow the **Player**, they only have **AttackBehaviour** and **WanderBehaviour** in their "behaviours" HashMap. On the other hand, **Invader**, like all other **Enemies**, will have **AttackBehaviour**, **FollowBehaviour** and **WanderBehaviour**. As **Allies** and **Invaders** cannot be removed from the **GameMap** if the **Player** rests on the **SiteOfLostGrace**, they are both removed from the “restResettables” ArrayList stored inside the **ResetManager** singleton. As **Enemies** implement **Resettable**, the **Ally** and **Invader** subclasses will inherit the "reset" methods. However, since they are no longer in the "restResettables" ArrayList (but still in the "resetResettables" ArrayList), they can only be removed from the **GameMap** if the **Player** dies. Again, this demonstrates the ease of extendability of our code, further proving that it follows OCP as zero modifications are required to implement these new features (Pros). However, the manipulation of the

ResetManager Resettables ArrayList may be a potential violation of the SRP as the **Ally/Invader** have too many responsibilities, resulting in less maintainable code (Cons).



engine
items
Item
1
actions
<<abstract>> Action
positions
<<abstract>> Ground
game
environments
1
Altar
blesses
maps
<<abstract>> Maps
ThirdChurchOfMarika
<<displays>>
MapDisplay
actions
ConsumeAction
BuyingAction
1
1
weapons
<<interface>> Buyable
buys
1
items
consumes
1
<<interface>> Consumable
FlaskOfCrimsonTears
SacredTears
<<upgrades>>
<<uses>>
<<enumeration>> ItemStatus

Design Rationale (Requirement 5)

A. New Map

A new **GameMap** called the **ThirdChurchOfMarika** is added to the **World**. Similar to other **GameMaps**, the **ThirdChurchOfMarika** class will inherit from the **Maps** abstract class, using the **FancyGroundFactory** and **MapDisplay** to construct the **GameMap** instance for itself. Then, the **GameMap** is manually created in the **Application** class. As shown in Requirement 1, these new **GameMaps** follow the OCP as no modification of the **Maps** class is required to add new **GameMaps** to the **World**, meaning our code is easily extendable for new **GameMaps** (Pros). However, as every **GameMap** is added manually in the **Application** class, redundant duplicated code will be created as every single **GameMap** requires the instantiation of their respective **Maps** subclass (Cons).

The **ThirdChurchOfMarika** will contain a new **Ground** subclass - **Altar**. The **Altar** will have different parameters within its constructor: "offeringPrice", "offeringIncrement", "offeringLimit" and "divineBlessing". These instance variables are used to determine what **Item** is given from the **Altar**, for how much, how expensive they will get after each offering and how many offerings are allowed. This design choice is chosen in future scenarios where more **Altars** are added into the game but with different **Items** or prices, thus showing that our code is easily extendable as no modification is required for the **Altar** class for these new features (OCP) (Pros) (Future extension). However as the **Altar** constructor requires parameters, the **Ground** has to be manually set using the "setGround" method in the **GameMap** inside the **Application** class, which violates the DRY principle due to repeated duplicative code (Cons).

The **Altar** class will implement the interface **Buyable**, enforcing a "buy" method to allow the **Player** to make offerings of **Runes** in exchange for an **Item**, in this case being **SacredTears**. Every time an offering is made, the "offeringPrice" is increased based on the "offeringIncrement" variable. If the **Player** does not have enough **Runes**, no offering is made as the "Gods are displeased with their stringent offering". The "allowableActions" method inside the **Altar** class will create and return a new **BuyingAction** if the **Player** is adjacent to it, allowing the **Player** to offer **Runes**. This shows our code adheres to the OCP because this **Ground** can implement the **Buyable** interface along with using the **BuyingAction** without additional modifications, showing our code is easily extendable for future **Buyables** (Pros).

B. New Consumable

SacredTears class inherits from the **Items** abstract class, implementing the **Consumable** interface, overriding the “consume” method from its parent class. The **SacredTears** is used to upgrade the **FlaskOfCrimsonTear's** potency, increasing the amount of health restored on used for it with diminishing returns. The **SacredTears** have an instance variable "healthToIncrease" which represents the increase in health restored per use. The **FlaskOfCrimsonTear** will have an instance variable "potency" describing the potency of itself, along with another instance variable "healthToHeal" which represents the health restored on use.

The **SacredTears** "consume" method will upgrade the **FlaskOfCrimsonTears** potency. For example, the **FlaskOfCrimsonTears** initial health restored is 250. Once **SacredTears** is consumed, the "healthToHeal" is now 345 and the "potency" will be 1. The next time another **SacredTears** is consumed, the "healthToHeal" will be increased by 85 to 430, and the "potency" will be incremented by one to 2, in which each successive use will decrement the "healthToIncrease" by 10 until the minimum health upgrade (which in this case will be 15) is reached. ie:

1) First upgrade will be a +95 health increase (345 total health restored)
2) Second upgrade will be a +85 health increase (430 total health restored) ...
8) Eight upgrade will be a +25 health increase (730 total health restored)
9) Ninth upgrade will be a +15 health increase (745 total health restored)
10) Tenth upgrade will be a +15 health increase (760 total health restored) 11) Eleventh upgrade will be a +15 health increase (775 total health restored)
12) Twelfth upgrade will be a +15 health increase (790 total health restored)

The **Altar** "offeringLimit" will be 12, where no more **SacredTears** are given to the **Player** once 12 offerings are made. The "tick" method will add a **ConsumeAction** to the "allowableActions" List, allowing the **Player** to consume the **SacredTears**, upgrading their **FlaskOfCrimsonTears** in the process via the "consume" method overridden from the **Consumable** interface. Through this, we have once again demonstrated the flexibility and extendability of our code design as new **Consumables** are added with no modification being made, thus adhering to the OCP (Pros). However, this approach introduced tight coupling between the **SacredTears** and the **FlaskOfCrimsonTears**. This means that if in the future, **SacredTears** can upgrade different **Items**, modifications to the **SacredTears** are required to do so, resulting in the violation of the OCP (Cons).