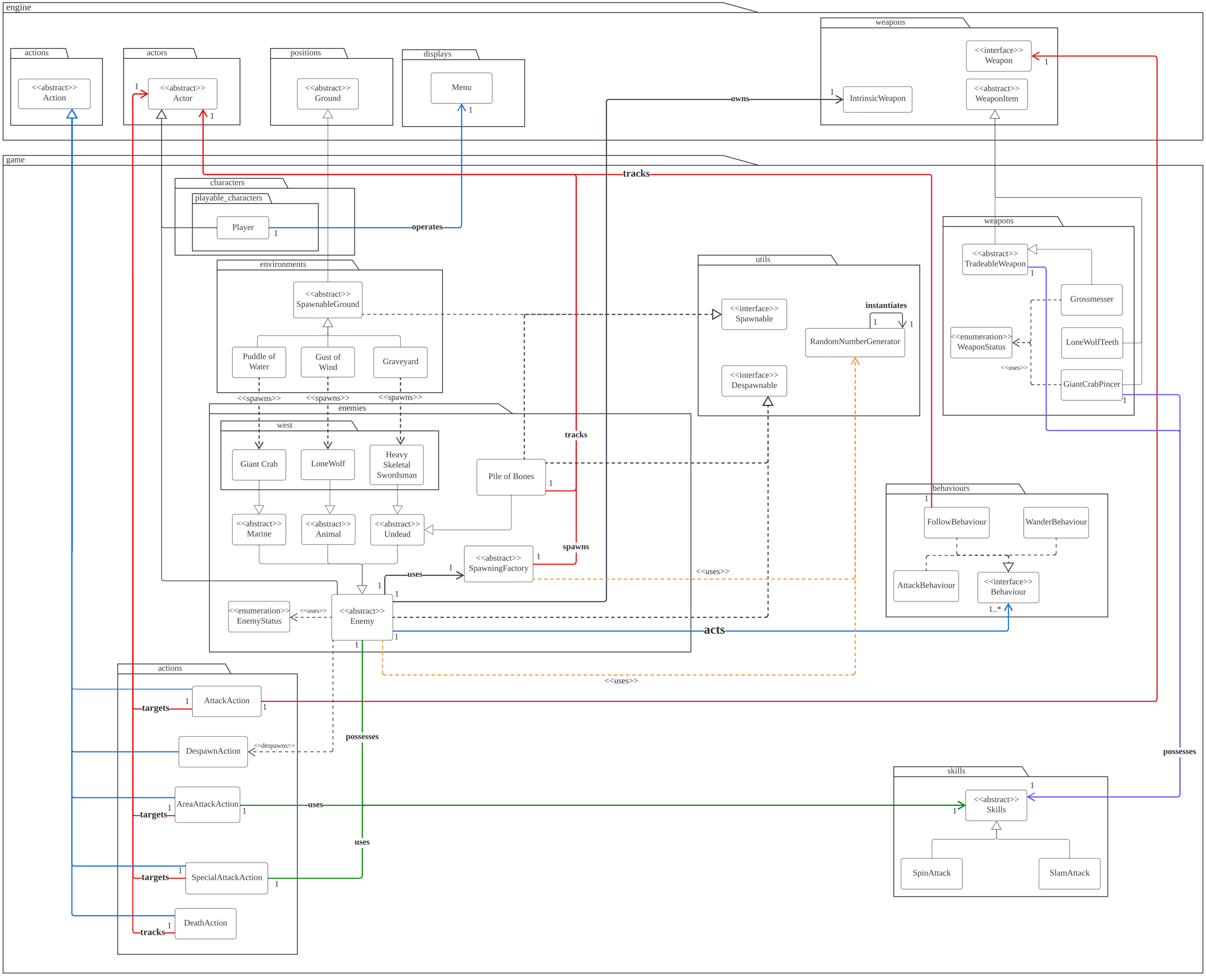
Group: MA_Lab04_Group5

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

Link to Design Rational Google Docs to see version history:
https://docs.google.com/document/d/1-P7d3E8ZiUJNR1AlCHeGUpw6gKKRr1ZkHc8LjOmUpjk/edit?usp=sharing


engine
actions
<<abstract>>
Action
actors
1
<<abstract>>
Actor
1
positions
<<abstract>>
Ground
displays
Menu
1
weapons
<<interface>>
Weapon
1
owns
1
IntrinsicWeapon
<<abstract>>
WeaponItem
game
tracks
characters
playable_characters
Player
1
operates
environments
<<abstract>>
SpawnableGround
Puddle of
Water
Gust of
Wind
Graveyard
<<spawns>>
<<spawns>>
<<spawns>>
enemies
west
Giant Crab
LoneWolf
Heavy
Skeletal
Swordsman
<<abstract>>
Marine
<<abstract>>
Animal
<<abstract>>
Undead
Pile of Bones
1
tracks
spawns
<<abstract>>
SpawningFactory
1
uses
1
1
<<uses>>
<<enumeration>>
EnemyStatus
<<uses>>
<<abstract>>
Enemy
1
1
1
acts
utils
<<interface>>
Spawnable
instantiates
1
1
RandomNumberGenerator
<<interface>>
Despawnable
weapons
<<abstract>>
TradeableWeapon
1
Grossmesser
<<enumeration>>
WeaponStatus
<<uses>>
LoneWolfTeeth
GiantCrabPincer
1
behaviours
1
FollowBehaviour
WanderBehaviour
AttackBehaviour
<<interface>>
Behaviour
1..*
actions
AttackAction
1
1
targets
DespawnAction
<<despawns>>
AreaAttackAction
1
1
targets
SpecialAttackAction
1
1
targets
DeathAction
1
tracks
possesses
uses
uses
<<uses>>
possesses
skills
1
<<abstract>>
Skills
1
SpinAttack
SlamAttack

SCENARIO: Player attacks using IntrinsicWeapon and kills HeavySkeletonSwordsman, turning it into a Pile of Bones.

(All that shown here is ONLY related to Requirement 1 and the scenario given)

:AttackAction
actor:Player
weapon:IntrinsicWeapon
target:HeavySkeletonSwordsman
map:GameMap

execute(actor, map)

Opt
[weapon == null]
getIntrinsicWeapon()
weapon: IntrinsicWeapon

chanceToHit()
hitRate: int

Alt
[!(getRandomInt(1, 100) <= hitRate)]
verb()
verb: String
result: **actor** tried to **verb target** but misses

[Else]
damage()
damage: int
result: **actor verb target** for **damage** damage

hurt(damage)
isConcious()
hitpoints > 0: boolean

Opt
[!target.isConscious()]
<<creates>>
:DeathAction

execute(target, map)
addCapability(Status.DEAD)
getItemInventory()
itemInventory: List<Item>

Loop
[for each item in itemInventory]
<<creates>>
dropActions: ActionList
add(item.getDropAction(target))
drop: DropAction

getWeaponInventory()
weaponInventory: List<WeaponItem>

Loop
[for each weapon in weaponInventory]
add(weapon.getDropAction(target))

Loop
[for each action in dropAction)]
execute(target, map)

hasCapability(EnemyStatus.RESURRECTABLE)
capabilitySet.contains(capability): boolean

Alt
[target.hasCapability(EnemyStatus.RESURRECTABLE]
locationOf(target)
removeActor(target)
increaseMaxHp(points)
<<creates>>
pileOfBones:PileOfBones
addActor(pileOfBones)

[Else]
removeActor(target)

result: String
result: String

Group: MA_Lab04_Group5

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

Design Rationale (Requirement 1)

Environment

The environments (**Graveyard**, **Gust of Wind** and **Puddle of Water**) in this game were created using an abstract class called **SpawnableGround** that implements the **Spawnable** interface, which requires the 3 environments to implement a "spawn" function for spawning their respective enemies. We did this to adhere to the DRY (Don't Repeat Yourself) concept. Since **Graveyard**, **Gust of Wind**, and **Puddle of Water** classes have the same spawning functionality, they will inherit from the **SpawnableGround**, enabling them to share functionality and reduce code duplication, such as 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 (Open Closed Principle)(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** abstract class to add the **Enemy** to the map. This obeys the DIP (Dependency Inversion Principle) as by depending on an abstraction (**SpawningFactory**) 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).

Group: MA_Lab04_Group5

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

Each **Enemy** will have an instance variable of a **SpawningFactory** subclass, relating to the side of the map they will spawn at, together with a spawningChance variable. This is done to manage the different **Enemies** with varying spawning chances. In the future, if any new **Enemy** with a different spawning chance is added, no modification is required for our code to function, proving that it is easily extendable, thus obeying the OCP. (Pros) (Future Extension). However, each **Enemy** will create a new instance of **SpawningFactory**, resulting in code duplication and violating the DRY principle, leading to code bloat and an increase in difficulty in maintenance (Cons).

In addition, the **RandomNumberGenerator** class will be used in the **SpawningFactory** class 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.

Enemies

Regarding the enemies, the **Graveyard** will spawn only **Heavy Skeletal Swordsman**, the **Gust of Wind** will spawn only **Lone Wolf**, and the **Puddle of Water** will spawn only **Giant Crab**. Every enemy will then inherit from their respective types (**Heavy Skeletal Swordsman** inherits from **Undead**, **Lone Wolf** inherits from **Animal**, and **Giant Crab** inherits from **Marine**), which all inherit from the **Enemy** abstract class. This design would also adhere to the DRY concept as these enemies implement a despawning operation and behave similarly, reducing the amount of duplicated code (Pros).

This design also follows the LSP (Liskov Substitution Principle) as **Undead**, **Animal** and **Marine** can be used interchangeably with **Enemy**, resulting in making our code easier to extend for the future if more **Enemies** of different types are added (Future extension). For example, new enemies of the **Bird** type will inherit from the **Bird** class, which also inherits from the **Enemy** abstract class. Not only that, the subclasses have access to all similar functionalities to the parent **Enemy** class (Pros). However, this design is only applicable to enemies that are hostile to the **Player**. In the future, there may be an addition of enemies that are passive to the **Player** and this may require us to potentially modify the **Enemy** class, which violates the OCP (Cons). Moreover, this design implements multiple layers of inheritance that may potentially make it more difficult to comprehend and maintain for an outsider (Cons).

Similar to **Spawnable**, **Enemy** will implement the **Despawnable** interface that will enforce a “despawn” function. This “despawn” function is used to check whether the **Enemy** can despawn under the conditions that they aren’t following the **Player** and are within the

despawning chance. If so, it will utilise the **DespawnAction** class to perform the removal of the **Enemy** from the **GameMap**. This design adheres to DIP as the high-level module (**Enemy**) would be forced to depend on the low-level module (**GameMap**) to perform the despawning action if it wasn't for the abstraction layer (**DespawnAction**). This allows for greater flexibility and easier maintenance of the code, as the **Enemy** class won't be modified if there are future changes made to the **GameMap** (Pros). On the other hand, in the case of the future where there are **Enemies** that are unable to despawn, a separate class has to be created. This is because the **Enemy** class implements the **Despawnable** interface and creating a separate class for these **Enemies** will lead to the violation of DRY (Cons). To determine the chances of each enemy's despawning, the **RandomNumberGenerator** class will be utilised.

The reason why each enemy inherits from their respective type class is to prevent each enemy from attacking their own types using the **EnemyStatus** enumeration (Pros). For example, the **Undead** class will have an "EnemyStatus.NOT_HOSTILE_TO_UNDEAD" capability to signify that it can't be attacked by other **Undead** enemies. This is done through their respective "allowableActions" method which is overridden in each type class (**Undead**, **Animal**, **Marine**). The AttackActions will only be created provided the attacker has the capability "EnemyStatus.HOSTILE_TO_ENEMY" and does not have the "EnemyStatus.NOT_HOSTILE_TO_UNDEAD" capability in the example above. Through this, our code can be extended with new **Enemy** types in the future, in which they have to just override the allowableActions and check for their own capabilities for their type (the **Bird** will check if the attacker will not have the "EnemyStatus.NOT_HOSTILE_TO_BIRD" capability), without modifying the super **Enemy** class, which obeys the OCP. (Pros) (Future extension). This is better than the alternative of checking the specific instance of the attacker, which may result in multiple unnecessary checks, instead of using a blanket type to cover all similar **Enemy** types. The cons of this are obvious, as each respective type will require its own "allowableActions" function, resulting in repeated code for each type and therefore violating the DRY principle (Cons).

To fulfil the requirements where the **Heavy Skeletal Swordsman** will turn into a **Pile of Bones** after it is killed, we created a separate class for itself which still inherits from the **Undead** abstract class and implements the **Spawnable** interface. This is because we want to spawn in a **Pile of Bones** that will replace the **Heavy Skeletal Swordsman** when it dies. This design adheres to the OCP as we can add new features in the **Pile of Bones** without affecting the **Heavy Skeletal Swordsman** or **Undead** classes (Pros). We decided to use an instance variable, "turnsToResurrect", signifying the number of turns to resurrect. Any additional resurrectable **Undead** with a varying number of turns to resurrect can easily extend from our existing code without modification, showing that we have heeded the OCP (Pros) (Future extension). However, the **Pile of Bones** is inflexible, in which any non-**Undead** resurrectable enemy to be added in the future will not be able to reuse the **Pile of Bones** class, being forced to implement their own (Cons).

In addition, the **Pile of Bones** implements the **Despawnable** and **Spawnable** interface and utilises **DespawnAction**, as once the 3 turns have passed, the **Pile of Bones** will "despawn" before the **Heavy Skeletal Swordsman** respawns. Both of these two interfaces obey the ISP (Interface Segregation Principle) as it allows classes that only require the spawning functionality to use the **Spawnable** interface without being forced to implement the despawning methods and vice versa (Pros). Furthermore, this provides more flexibility, as adding new classes in the future that require spawning, despawning or both can be implemented without the two interfering with each other as shown in the **Pile of Bones** class, following the OCP concept (Pros) (Future extension). The downsides are that since **Pile of Bones** will have its own individual “despawn” and “spawn” functionality, they have to be overridden from their parent classes, resulting in duplication of code and therefore violating the DRY principle (Cons).

The **Behaviour** interface is implemented by the **FollowBehaviour**, **WanderBehaviour** and **AttackBehaviour**, forcing each subclass to implement its own “getAction” function, as each behaviour has its own functions and features. We follow SRP by creating separate classes for each type of behaviour. This allows each class to have a distinct role, making it simpler to comprehend along with minimising bugs as each subclass is isolated from the other (Pros). However, this could also violate the DRY principle as each behaviour would have to check all potential exit locations of the **Actor** performing the behaviour, leading to duplicated code. In the case of the **WanderBehaviour**, it’s done to determine which exit **Location** to move to. For **FollowBehaviour**, it’s to identify and determine the target to follow and lastly, for **AttackBehaviour**, it’s to create a list of attackable **Actors** on all adjacent tiles. Due to this, multiple repeated loops are used for each behaviour, creating duplicated code (Cons). Moreover, this obeys the DIP (Dependency Inversion Principle) as the **Enemy** relies on the **Behaviour** abstract interface instead of the concrete **WanderBehaviour**, **FollowBehaviour** or **AttackBehaviour.** Because of this, it's insulated from any change in the **Behaviour** subclass, leading to easily modifiable and future-proof code as each **Behaviour** is isolated from changes made in another (Pros). But having multiple different **Behaviours**, each with its own responsibility may result in the creation of complex code which makes it more difficult to maintain and understand for the uninformed (Cons).

The **FollowBehaviour** class employs a behaviour that will allow enemies to follow the **Player** if they are adjacent to them, disabling their despawning via using the “EnemyStatus.FOLLOWING” capability if the **Enemy** is stalking the **Player**. The **WanderBehaviour** will enable the enemies to wander around the map randomly, using **RandomNumberGenerator** to pick a **MoveAction** by random.

The **AttackBehaviour** will reuse the “allowableActions” of the adjacent target to return a list of performable actions to be done against them. In this case, it could be **AttackActions**, **SpecialAttackActions** or **AreaAttackActions**. Through this, we reduce the amount of duplicated code as we reuse inbuilt engine methods, following the DRY principle (Pros). The downside is that for each **Enemy** type, we have to implement and override the “allowableActions” method, increasing the complexity of our code even more (Cons).

Furthermore, the “allowableActions” will utilise the **Utils** static method “getAttackTypes” to determine the performable types of **Attack** (**AttackAction, SpecialAttackAction** and **AreaAttackAction**) the attacker can perform on the target. By using a static method, we can reduce the amount of repeated code, adhering to the DRY principle (Pros). However, usage of a static method may cause side effects, resulting in unpredictable bugs being produced in the **AttackBehaviour** or spreading, resulting in unmaintainable code (Cons).

The **AttackBehaviour** will then randomly choose one of the actions in the ActionList returned by the “allowableActions” method for that actor (which will be a 50% chance for a special or normal attack). Then, it will have a list of Actions for all adjacent **Enemies** with their randomly chosen action within it before randomly choosing one to execute. These **Behaviours** are stored in a hashmap with their values being the priority of execution with **AttackBehaviour** having the highest priority and **WanderBehaviour** being the lowest.

All these **Behaviours** are easily extendable for the future as any future **Enemy** which uses these **Behaviours** can merely add them to the “behaviours” HashMap. Moreover, if future **Enemies** have different behaviours, like a **RunningBehaviour** in which when they are adjacent to the **Player**, they will run away, can just implement the **Behaviours** interface and add said **Behaviour** to that specific **Enemy** without modifying the parent class, thus showing them this design choice obeys the OCP (Pros) (Future extension). However, any new additions of **Behaviours** for existing **Enemies** will require modifying the **Enemy** class to add the **Behaviour**, violating the OCP (Cons).
The following actions (**AttackAction**, **AreaAttackAction**, **SpecialAttackAction** and **DeathAction**) all inherit from the **Action** abstract class in the **engine** package. These classes are created to adhere to the SRP, which makes it easier to manage and modify as each class has their responsibilities, independent of the other, allowing for a more robust and maintainable code (Pros).

The **AttackAction** class, firstly, is used to execute an attack from the perpetrator to the victim in which the “execute” method is overridden from the parent **Action** class. Here, it will get the data attributes of the **WeaponItem** or **IntrinsicWeapon (**“hit rate/accuracy”, “damage” etc) to be used in the attack. The **RandomNumberGenerator** is used to determine whether the attack hit or misses based on hit rate/accuracy.

The **SpecialAttackAction** is similar to the **AttackAction**, but instead of utilising the stats of the **Weapon**, it will use the **Skills** class which is created in the **Weapon** instance. The **Skills** abstract class will contain the data attributes of the skill **(**“hit rate/accuracy”, “damage” etc). This is done to execute **Skill** attacks with different damage or accuracy from the **WeaponItem** used. This follows the OCP method as no modification of the **WeaponItem** class is required to execute the **Skill** attack, reducing the potential bugs created (Pros). On the other hand, the **SpecialAttackAction** code is virtually indistinguishable from the **AttackAction** code with the only minor difference being the usage of **Skills** instead of **Weapons**, leading to unnecessary code repetition, disobeying the DRY principle (Cons).

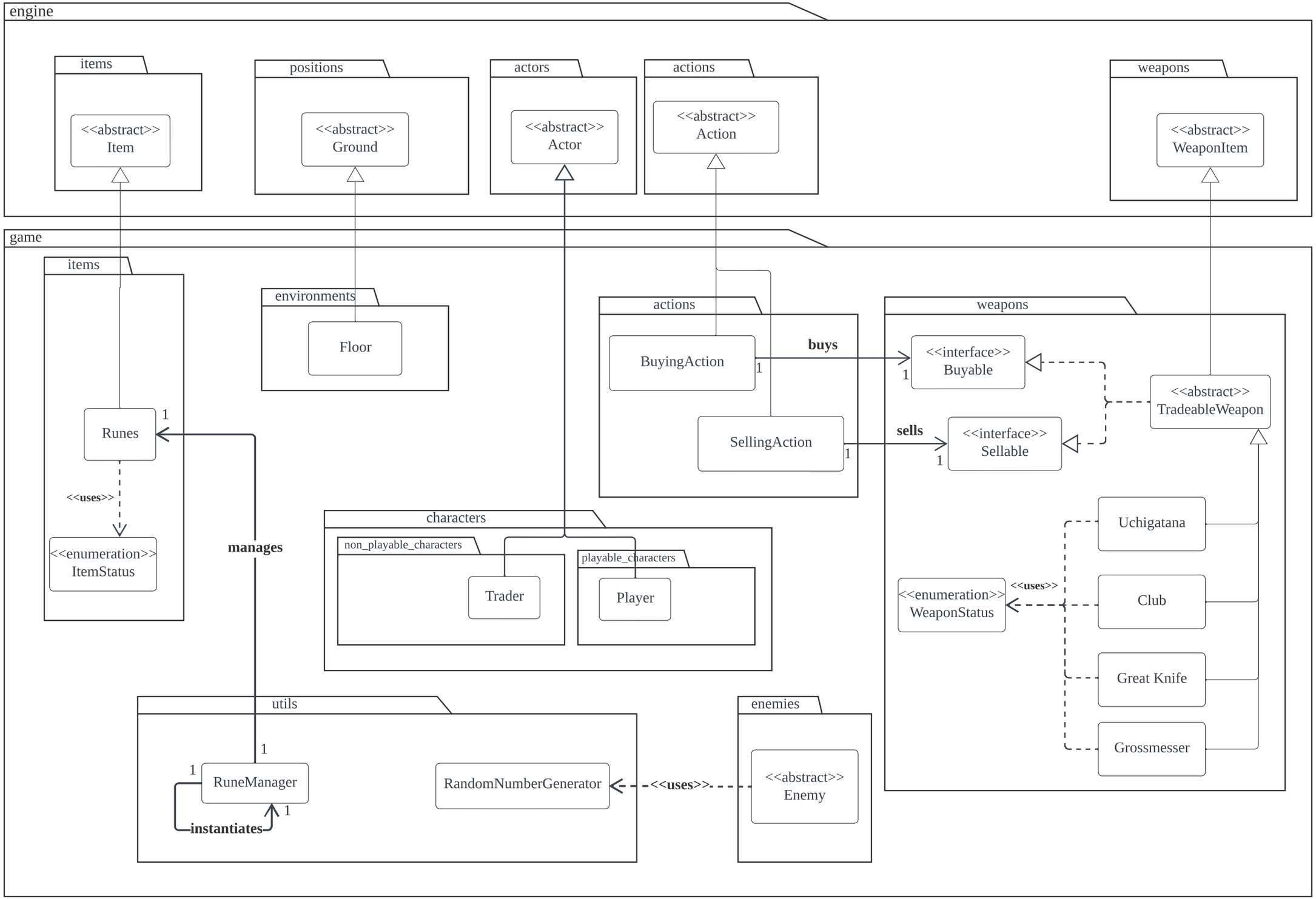
Group: MA_Lab04_Group5

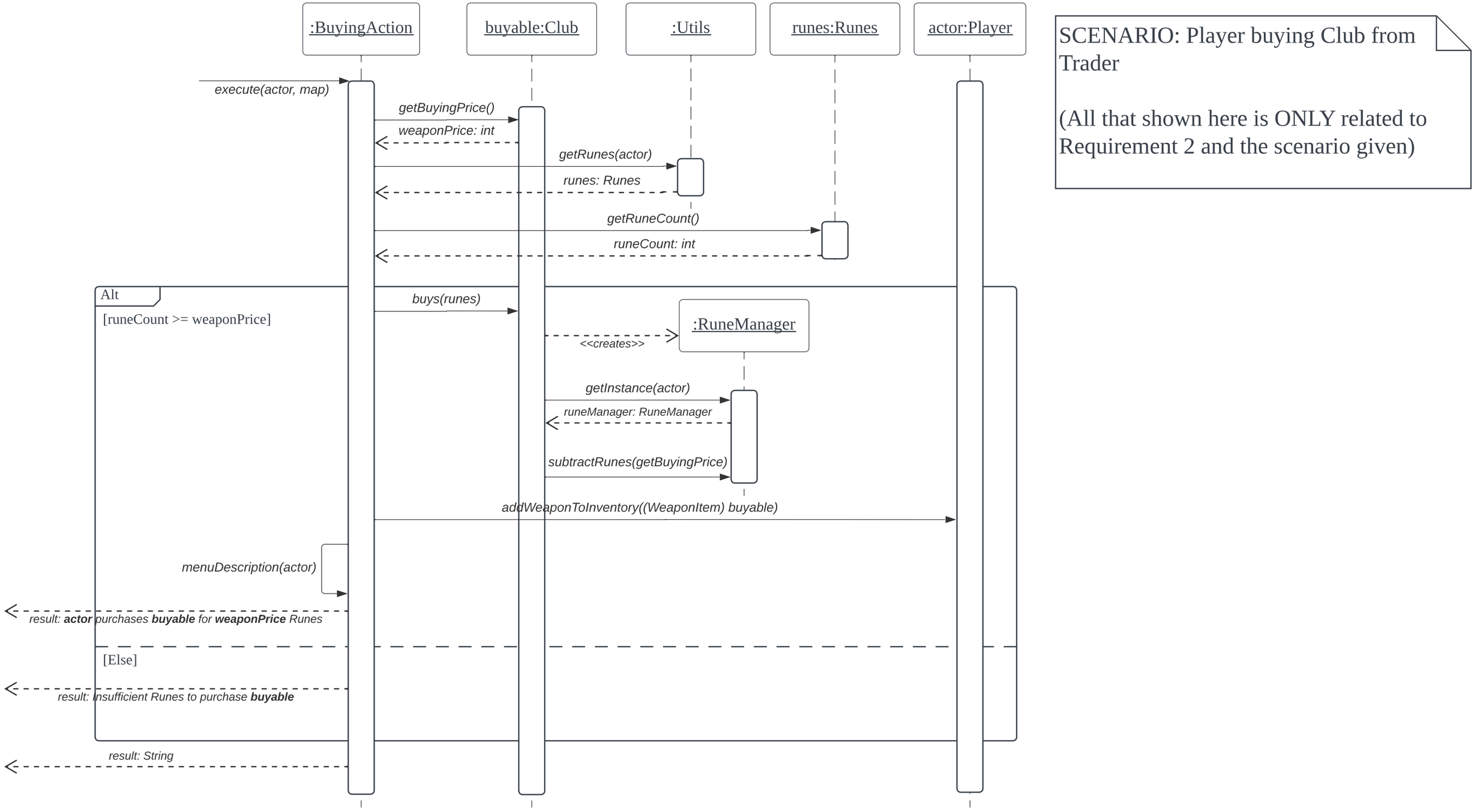
Members: Bryan Wong, Po Han Tay, Wah Yang Tan

The **AreaAttackAction** class will have a list of **SpecialAttackActions**, used for slam/spin attacks where multiple **SpecialAttackActions** are executed simultaneously on all adjacent actors. The **RandomNumberGenerator** class will be used in the **SpecialAttackAction** class to determine the attack's accuracy and will be calculated independently for each **SpecialAttackAction** executed. For example, the **Grossmesser** can execute an **AreaAttackAction** using the **SpinAttack** attributes set. Although at the current state of the game, **SpinAttack** will have the same damage/accuracy as **Grossmesser**, that may not be the case in the future. Mayhaps an additional **AreaAttackAction** or **SpecialAttackAction Skill** could be added in the future, where it may have different damage or accuracy than the **Weapon** used to execute it. Through this, our code is effortlessly extendable since **SpecialAttackAction** (and **AreaAttackAction** by proxy) uses the **Skills** data attributes for the special attack instead of relying on the **Weapon**, thus following the OCP as no modification of the **WeaponItem** is necessary to perform this (Pros) (Future extension). However, this may result in the creation of unwarranted **Skills** classes for **Skills** that only use the **Weapon** attributes instead of having different attributes, increasing code duplication and complexity and therefore violating the DRY (Cons).

Weapons

Each **Enemy** will have their own respective weapons with the **Heavy Skeletal Swordsman** using the **Grossmesser**, the **Lone Wolf** using the **LoneWolfTeeth**, and the **Giant Crab** using the **GiantCrabPincer**. The **LoneWolfTeeth** and **GiantCrabPincer** both inherit from **WeaponItem** abstract class whereas the **Grossmesser** class inherits from the **TradeableWeapon** abstract class, which also inherits from **WeaponItem,** for trading actions (Requirement 2). This is naturally done so that each weapon will have its own "getSkill" method inherited from the **engine**, which **GiantCrabPincer** and **Grossmesser** uses to perform an **AreaAttackAction**. This design follows OCP as it allows for the addition of new weapons without modifying existing code. For future extensibility example, a newly created **BirdClaw** class which can perform an Area Attack can inherit from **WeaponItem**, and override the inbuilt "getSkill" method, which provides easier maintenance in the codebase (Pros) (Future extension). However, every new **Enemy** will have their own specialised **WeaponItem** instead of using the inbuilt **IntrinsicWeapon** as a result, leading to duplicate classes and codes, violating the DRY principle (Cons). Not only that, both **GiantCrabPincer** and **Grossmesser** use the **WeaponStatus** enum class as it enables both weapons to perform an Area Attack using the enum "WeaponStatus.CAN_AREA_ATTACK". This enum is then used in the "getSkill" method to determine if an **AreaAttackAction** can be performed. To prevent the **Enemy** like **LoneWolf** from dropping its **WeaponItem**, the "togglePortability" method is used to turn the "portable" variable false, thus preventing them from being dropped when killed. As **WeaponItem** by default has "portable" set to True, **Grossmesser** can be dropped by the **Heavy Skeletal Swordsman** after their **Pile of Bones** is killed.


engine
items
<<abstract>>
Item
positions
<<abstract>>
Ground
actors
<<abstract>>
Actor
actions
<<abstract>>
Action
weapons
<<abstract>>
WeaponItem
game
items
Runes
1
<<uses>>
<<enumeration>>
ItemStatus
manages
environments
Floor
actions
BuyingAction
1
buys
SellingAction
1
sells
1
weapons
<<interface>>
Buyable
1
<<interface>>
Sellable
1
<<abstract>>
TradeableWeapon
Uchigatana
Club
Great Knife
Grossmesser
<<enumeration>>
WeaponStatus
<<uses>>
characters
non_playable_characters
Trader
playable_characters
Player
utils
1
1
RuneManager
1
instantiates
RandomNumberGenerator
<<uses>>
enemies
<<abstract>>
Enemy



SCENARIO: Player buying Club from Trader

(All that shown here is ONLY related to Requirement 2 and the scenario given)

Group: MA_Lab04_Group5

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

Design Rationale (Requirement 2)

Runes

Runes inherit from the **Items** abstract class and do not have “ItemStatus.TRADEABLE”, thus they are untradeable. The **ItemStatus** enumeration will allow a multitude of items to have different capabilities, without them interfering with or relying on each other. This demonstrates OCP as any future items added can use the enum if it has the same capability (Pros) (Future extension). Likewise, additional **ItemStatus** capabilities can be added, making it easily extendable. On the other hand, checking for the multiple **ItemStatus** capabilities may result in bloated and obtuse code, making it hard to maintain and debug (Cons).

Runes will have a “runeCount” variable, keeping track of how many runes the **Actor** has. Each Enemy will have a **Rune** object in their ItemInventory, along with a “runeMin” and “runeMax”, and then use **RandomNumberGenarator** to generate a random amount of runes between “runeMin” and “runeMax” that will be given to the **Player** when killed which will be stored in their “runeCount”. To manage the **Runes**, a **RuneManager** singleton will be used. The **RuneManager** will accept a **Runes** object or an **Actor** instance, which will then search for the **Runes** in said **Actor**’s inventory. This is done to avoid downcasting when trying to modify the **Runes**, along with having the management of **Runes** in a separate class to provide encapsulation and abstraction. This adheres to OCP, as any method that can modify the **Runes** (“addRunes”, “subtractRunes”) will be able to be added to the **RuneManager** class without affecting the **Actor** or **Runes** class (Pros). When the **DeathAction** is executed and the target has the capability “EnemyStatus.CAN_DROP_RUNES” and the attacker has the capability “Status.CAN_GAIN_RUNES”, then the **RuneManager** “gainRunes” function will be called, which calls the “addRunes" function to increase the “runeCount” of the **Player** based on the “runeCount” of the **Enemy** in their respective **Runes** object. However, this may make the code less maintainable as DeathAction relies on **RuneManage**r, and any changes in **RuneManager** may induce bugs in **DeathAction** (Cons).

Trader

The **Trader** will spawn on the **Floor**. The **Floor** class utilises the “canActorEnter” function that only allows **Actors** with the “Status.CAN_ENTER_FLOOR” to enter, preventing **Enemies** from waltzing into the building. Moreover, the usage of an enumeration will allow any future **Actor** that can enter the **Floor** to do so without modifying the **Floor** class (OCP); they will only require adding the “Status.CAN_ENTER_FLOOR” capability. Through this, we have shown our code is easily extendable for future uses (Pros) (Future extension). On the other hand, having so many different enumeration to handle may lead to poor, unmaintainable code and numerous redundant checks to see if the **Actor** has that capability, thus reducing the readability of the code (Cons).

Weapons

Each weapon (**Uchigatana**, **Club**, **Great Knife** and **Grossmesser**) inherits from the **TradeableWeapon** abstract class, which follows SRP as the **TradeableWeapon** class will

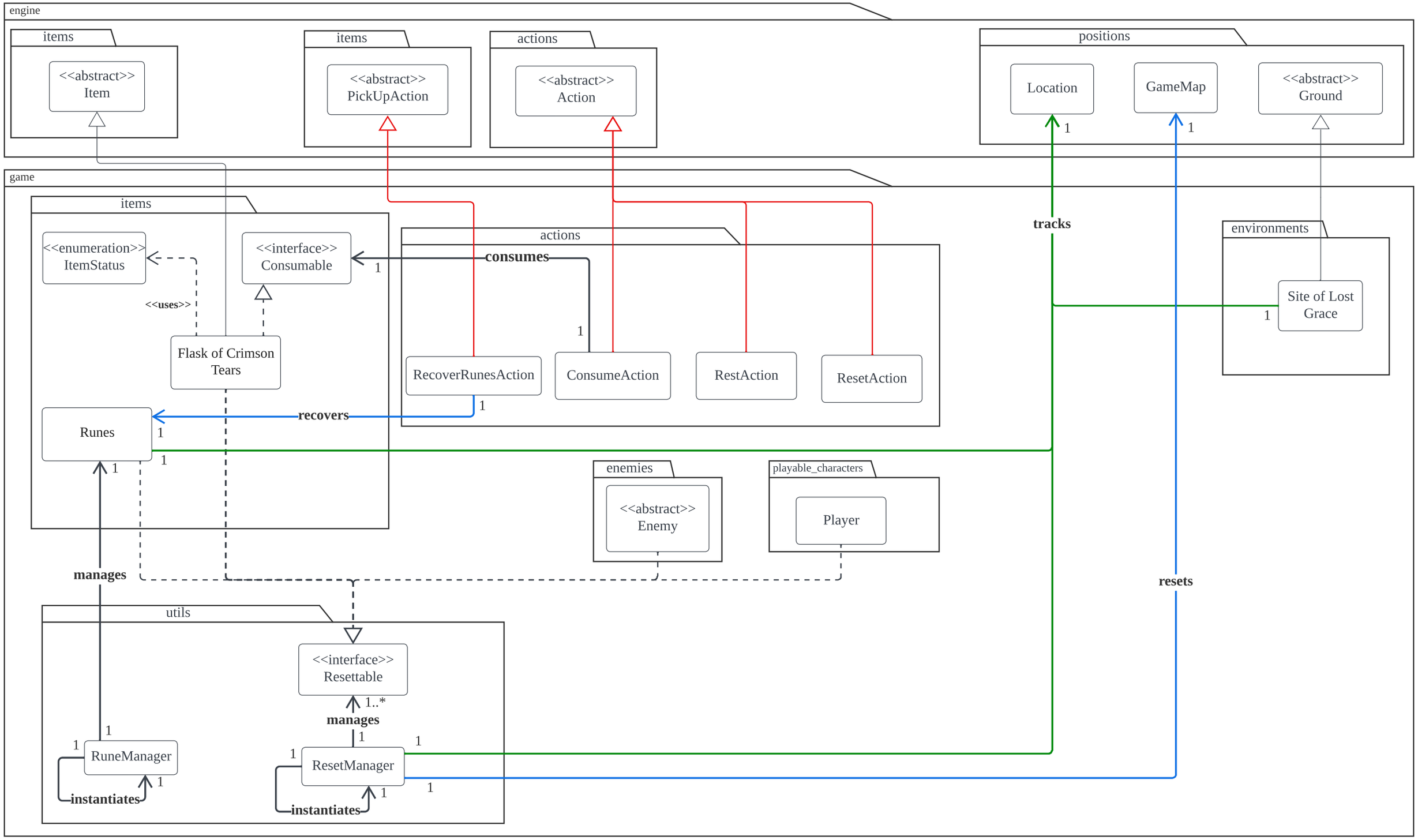
only manage tradable **Weapons**, improving the maintainability of the code (Pros). Having these weapons inherit from an abstract class will reduce code duplication due to shared features and methods for the subclasses, which follows the DRY principle (Pros). Following that, **TradeableWeapon** implements the **Buyable** interface. This is done to enforce a "buys()" when buying, along with the "getBuyingPrice" and "setBuyingPrice" methods. They will also implement the **Sellable** interface which enforces a "sells()" method when selling, together with "getSellingPrice" and "setSellingPrice" methods. They will have a "buyingPrice" and "sellingPrice" variable which determines their price during a **BuyingAction** and **SellingAction** respectively. By having two separate interfaces, **Buyable** and **Sellabe**, this obeys the ISP as it separates the responsibilities of buying and selling, allowing the classes that are **Buyable** to only implement the buying method without being forced to implement the other (and vice versa) (Pros). However, to reduce repeated "buy" and "sell" methods for every single **WeaponItem**, we decided to implement both in **TradeableWeapon** as all **WeaponItems** can be sold to the **Trader**. For the **Trader**, only **WeaponItems** in their inventory can be bought, limiting them to only **Uchigatana**, **Club** and **Great Knife**. However, because of this, the subclasses of **TradeableWeapon**, like **Grossmesser,** that cannot be bought from the **Trader** will inherit the "buy" method along with it being a **Buyable**, resulting in it having methods it does not require (Cons). The tradeoff here is that there will be less repeated code, as instead of each **WeaponItem** implementing the **Buyable** or **Sellable** interface along with their methods, they are all inherited from **TradeableWeapon**. We also decided to have a "WeaponStatus.TRADEABLE" to indicate if the **WeaponItem** is tradeable or not, so that if in the future, an untradeable **WeaponItem** is added, we can choose not to inherit from **TradeableWeapon**, thus following OCP as we do not need to modify the **Trader** or the **Buying/SellingActions** to take in account of this (Pros) (Future extension).

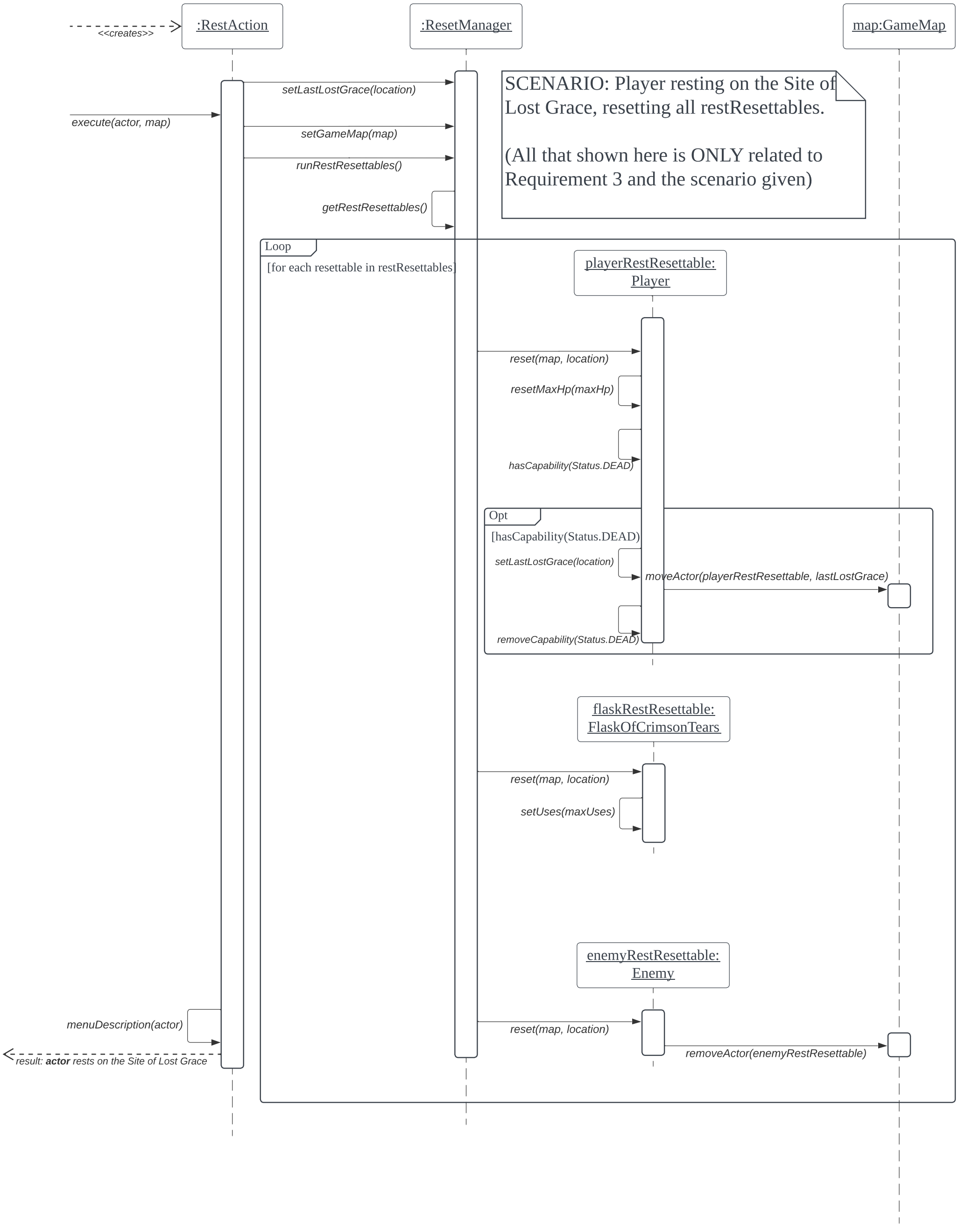
Trading

To manage Trading, **BuyingAction** and **SellingAction** are used, both of which inherit from the **Actions** abstract class. **BuyingAction** uses a **Buyable** and **SellingAction** uses a **Sellable**. This obeys the LSP principle as the **WeaponItem** can instead be substituted with their respective interface, thus avoiding downcasting to call methods like "getBuyingPrice" etc. In addition, having the actions use those interfaces will provide functionality if in the future non-**Weapons** (like a tradeable **Item**) are added. Hardcoding the input to be a **WeaponItem** may result in a modification in the future if different **Items** can be bought or sold. Hence, our methods of using the **Buyable** or **Sellable** interface as the input type will result in our code being future-proof and easily extendable, following the OCP (Pros) (Future extension). However, this may result in duplicated code as every subclass of the **Buyable** interface and **Sellable** interface will be forced to implement their own "buy" or "sell" methods, despite sharing the same feature/function. Because of this, the DRY principle is violated (Cons).

The **Player** "runeCount" will be updated based on the respective **BuyingAction** or **SellingAction** performed. For example, if they bought the **Club**, their "runeCount" in the **Rune** class will subtract by 600. In the **BuyingAction** "execute" method, it will call the

“buys” method from the **Buyable**, in this case being the **Club**. The “buy” method will utilise **RuneManager** to manipulate the “runeCount” of the **Runes**. The “subtractRunes” method is called, with the input being the “buyingPrice” of the **Club**. Using this, downcasting can be avoided to modify the **Runes**, which obeys the OCP principle since no modification is required for future tradeable **Items** (Pros) (Future extension). However, the “buy” and “sell” methods will be dependent on the **RuneManager** to manage the **Runes**, resulting in bugs or side effects leaking through and creating unmaintainable code (Cons).


engine
items
<<abstract>>
Item
items
<<abstract>>
PickUpAction
actions
<<abstract>>
Action
positions
Location
GameMap
<<abstract>>
Ground
1
1
game
items
<<enumeration>>
ItemStatus
<<interface>>
Consumable
<<uses>>
Flask of Crimson
Tears
Runes
1
1
1
1
recovers
actions
consumes
1
RecoverRunesAction
ConsumeAction
RestAction
ResetAction
1
tracks
environments
Site of Lost
Grace
1
enemies
<<abstract>>
Enemy
playable_characters
Player
manages
resets
utils
<<interface>>
Resettable
1..*
manages
1
1
1
RuneManager
1
1
instantiates
ResetManager
1
1
1
instantiates


:RestAction
:ResetManager
map:GameMap
<<creates>>
SCENARIO: Player resting on the Site of Lost Grace, resetting all restResettables.
(All that shown here is ONLY related to Requirement 3 and the scenario given)
execute(actor, map)
setLastLostGrace(location)
setGameMap(map)
runRestResettables()
getRestResettables()
Loop
[for each resettable in restResettables]
playerRestResettable: Player
reset(map, location)
resetMaxHp(maxHp)
hasCapability(Status.DEAD)
Opt
[hasCapability(Status.DEAD)
setLastLostGrace(location)
moveActor(playerRestResettable, lastLostGrace)
removeCapability(Status.DEAD)
flaskRestResettable: FlaskOfCrimsonTears
reset(map, location)
setUses(maxUses)
enemyRestResettable: Enemy
reset(map, location)
removeActor(enemyRestResettable)
menuDescription(actor)
result: actor rests on the Site of Lost Grace

Group: MA_Lab04_Group5

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

Design Rationale (Requirement 3)

Flask of Crimson Tears

Flask of Crimson Tears inherits from **Item** abstract class and is added to the **Player** inventory at the game start. **Flask of Crimson Tears** uses a variable "uses", representing their number of uses, which is default set to 2. The **Flask of Crimson Tears** also implements the **Consumable** interface, which requires the **Flask of Crimson Tears** to implement a "consume()" function that heals the **Player** after consumption. Besides that, there will be a **ConsumeAction** class inheriting from the **Action** abstract class that allows the Player to execute the "consume()" function on the **Item**. This design obeys OCP as there could be potential **Items** added in the future that could also be consumed, thus being able to implement this function without modifying any existing code, therefore making the code "open" for more extensions (Pros) (Future extension). However, this will lead to duplication of code for **Items** with similar “consume()” functionality, as every subclass of **Consumable** is forced to implement its own “consume()” function. One example of this could be an apple, when consumed, will also heal the **Player's** health just like the **Flask of Crimson Tears**. As a result, this violates the DRY principle (Cons). The **ConsumeAction** will only be available in the **Flask of Crimson Tears** “allowableActions” method provided that the number of uses is more than 0. Once all **Flask of Crimson Tears** uses are used up, the **ConsumeAction** will no longer be available.

Not only that, but it also implements the **Resettable** interface to reset the max uses. Since the **Flask of Crimson Tears** implements both interfaces, we managed to avoid the creation of large interfaces that may force its subclasses to implement unnecessary functions, which follows the ISP (Pros). In the future, any additional item that can be **Consumable** and **Resettable** can also be implemented in both interfaces, thus allowing the code to be easily extended without modifying any existing code, thus following the OCP (Pros) (Future extension).

Site of Lost Grace

The **Site of Lost Grace** inherits from **Ground** and when the **Player** rests on it, the game resets. Each **Site of Lost Grace** will have an instance variable, saving its current **Location**, which is set once in the “tick” method if the **Location** has not been set yet. The **ResetManager** will then save the **Location** of the last **Site of Lost Grace** the **Player** rested on. Once the **Player** is killed, they will respawn back at the saved **Site of Lost Grace** using the saved **Location**. Although the current implementation has only one **Site of Lost Grace**, this will improve the scalability of the code as if more **Sites of Lost Grace** are added, the functionality of respawning at the last **Site of Lost Grace** will not be impeded, following the concept of OCP as no modification is needed when more **Site of Lost Grace** are added (Pros) (Future extension). However, this may result in unmaintainable code, as the **ResetManager** “reset” method may rely on the **Location** class, resulting in unpredictable bugs and side effects (Cons). The **Site of Lost Grace** will execute the **RestAction**, which inherits from the **Action** abstract class when the **Player** rests on it. The **RestAction** will allow any **Actor** with the capability “Status.CAN_REST” to rest on the **Site of Lost Grace**, meaning any future

Actor with this capability will also be able to rest on the **Site of Lost Grace** without modifying the **RestAction**, therefore following the OCP (Pros) (Future extension).

Game Reset

The singleton class **ResetManager** creates only one instance of itself using the public static factory method. This is done to prevent the recreation of **ResetManager** every single time it's called because **ResetManager** will hold an array of **Resettables**. Recreating the **ResetManager** will result in multiple arrays of **Resettables** when only one is required. **ResetManager** has only the responsibility of managing **Resettables**, which heeds the SRP, allowing for easier, maintainable code (Pros). However, **ResetManager** will be tightly coupled with **RestAction** or **ResetAction**, resulting in potential side effects and bugs as a result if one is changed (Cons).

Objects that reset when the **Player** rests implement the **Resettable** interface (**Player**, **Flask Of Crimson Tears**, **Runes** and **Enemies)**, forcing all subclasses to implement a “reset” function. Since every individual **Resettable** in the future (such as an **Item** which disappears once the game resets), may have a different reset feature, this design will allow us to have each **Resettable** implement its own reset functionality without any modification to its parent class or other code, thus obeying the OCP (Pros) (Future extension). However, some **Resettables** may have similar “reset” functionality, resulting in repeated code as all **Resettables** are forced to implement their own “reset” function, resulting in a violation of the DRY principle (Cons).
Player, **Flask Of Crimson Tears** and **Enemies** will use the **ResetManager** “registerRestResettable”, adding these objects to the restResttable array stored within the **ResetManager** singleton. When the **Player** rests on the **Site of Lost Grace**, via **RestAction**, the “runRestResettables” method in **RestManager** is called. The method will loop through the “restResettables” **Resettable** array list, calling the “reset” method for each **Resettable** in the list. In this case, the **Player's** health will be reset, the **Enemy** will be removed from the map, and the **Flask of Crimson Tears** will reset its uses back to the max uses.

The **Player**, **Flask Of Crimson Tears** and **Enemies** and **Runes** will be added to the “deadResettables” array list via the **ResetManager** “registerDeadResettables” method. This design choice is chosen due to certain **Resettables** only having their “reset” method executed under different circumstances. **Runes** can only be “reset” when the **Player** dies, not when the **Player** rests. Furthermore, having two separate Arrays will allow future **Resettables,** which resets under different circumstances, to “reset” without modifying the **RestAction**, thus adhering to the OCP (Pros) (Future extension). The downside of having two separate arrays is the additional complexity created. Since “deadResettables” and “restResettables” and their respective methods are virtually the same, this will result in duplicated code such as the “register” methods that both have which will allow **Resettables** to be added to their respective **Resettables** array list. Due to this, the DRY is violated (Cons).

The **ResetAction** will call the “runDeadResettables” method which will loop through and execute all the “deadResettables”. This **ResetAction** is only executed within the **DeathAction** if the dead **Actor** has the capability “CAN_RESET_GAME”. Through this, any future dead **Actor** that can also reset the game can do so without modifying the **DeathAction** or **ResetAction**, following OCP (Pros) (Future extension). However, this led to the creation of obtuse code, with multiple checks to take into account different **Actors** within the **DeathAction**, reducing the maintainability of the code (Cons).

Runes

The **Runes** object uses both the **engine** “tick” methods to store the current **Location** of the **Runes** every turn, when it’s being carried and when it’s on the **Ground**. When the **Actor** with the capability “Status.CAN_GAIN_RUNES”, which the **Player** has, dies, the **Player** is moved to that stored **Location.** This is so that when the **DropAction** for the **Runes** is executed, it will drop at the **Player’s** previous location and not its current location. The reason why we decided to implement it this way is to reuse the **engine DropAction** method where the **Items** are dropped at the **Player's** current **Location**. The alternative is to create an entirely new **RunesDropAction**, which is merely just for dropping **Runes** when the **Player** dies. This is additional unneeded complexity, hence our implementing this design instead of the alternative obeys the DRY principle as we reuse existing **Actions** within the **engine** without creating new ones just for a single purpose (Pros). However, this method of moving the **Player** manually is bug-prone, as they rely on the **RuneManager** to return the stored **Location** of the **Runes**, resulting in a tight coupling between the **DeathAction** and **RuneManager** (Cons).

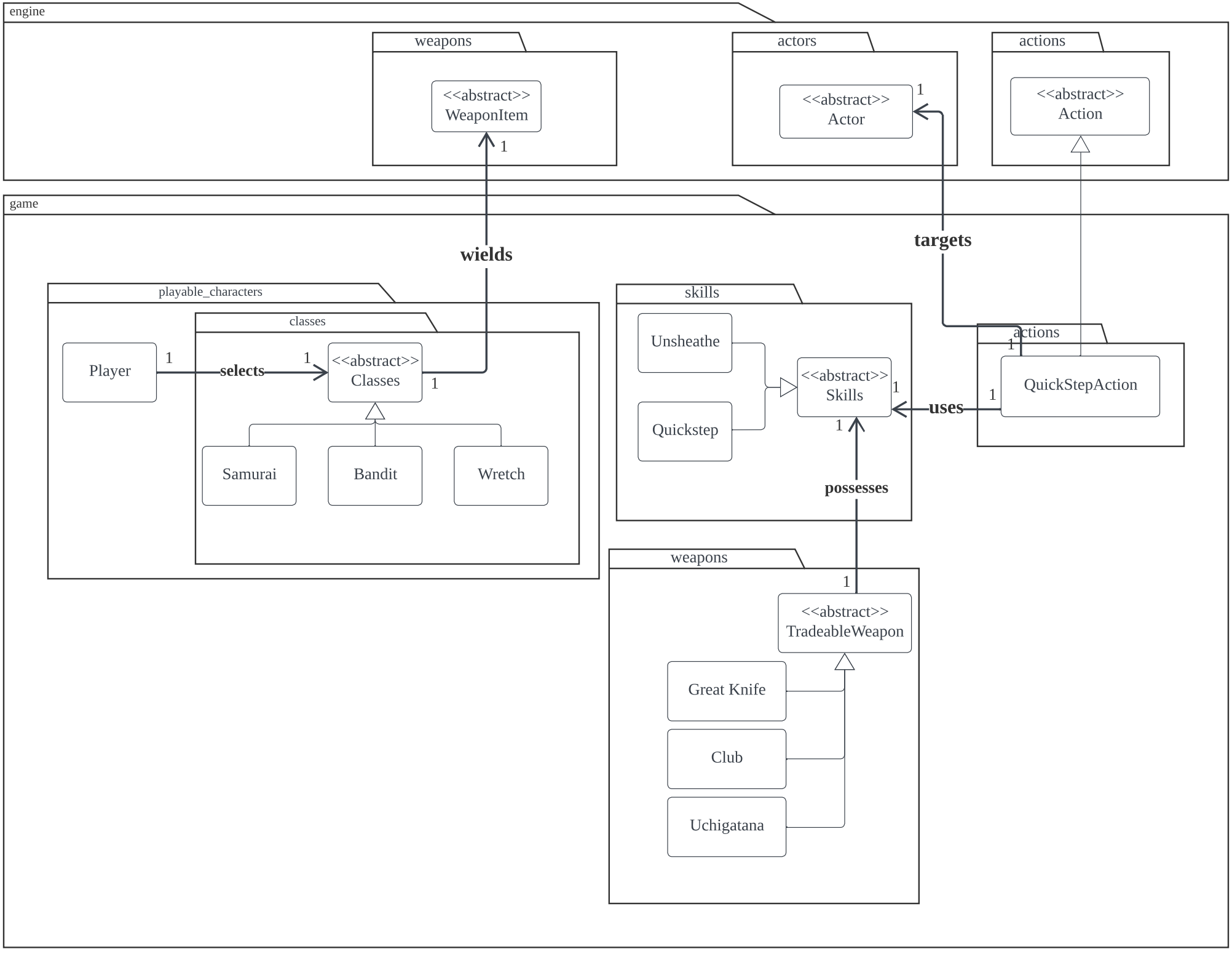
The **Runes** will drop if the **Player** dies, and the **Runes** “reset” method will increment the variable “numberOfDeaths” each time it is called. Since **Runes'** “reset” method is only called when the **Player** dies (**ResetAction**), this is used as a counter to count the number of deaths. So if the **Player** dies without picking up dropped **Runes**, the stored current **Location** of the **Runes** will be used to remove the **Runes** from the map, to be lost forever.

As **PickUpItemAction** in the **engine** class is fixed, only being able to add the dropped **Item** to the **Actor’s** inventory, a new **Action** is recovered to facilitate the addition of **Runes** to the **Player.** Here, we decided to create a **RecoverRunesAction** that inherits from **PickUpAction**, which will utilise the **RuneManager** “gainRunes” method to increment the “runeCount” of the **Runes** in the **Player’s** inventory based on the “runeCount” of the **Runes** on the **Ground**. Once the **Runes** have been recovered, it is removed from the **Ground** by calling the “super.execute” method. The **Runes** “getPickUpActon” is overridden and replaced to return an instance of the **RecoverRunesAction** instead, only allowing **Actors** with the capability “Status.CAN_GAIN_RUNES” to pick them up. This method follows the SRP as they only have one responsibility which is to enable the **Player** to recover the lost **Runes**. As a result, the code is more modular and easier to maintain (Pros). However, the creation of a new **Action** specifically for recovering **Runes** may violate the OCP. This is because it is closed for extension, as the **Action** is specifically only for recovering **Runes** and nothing else (Cons).

Group: MA_Lab04_Group5

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

Any future **Items** in the future that provide a different effect when recovered will require the creation of a new and separate **Action** for them, leading to an increase in complexity and violating the DRY principle (Cons).


engine
weapons
<<abstract>>
WeaponItem
1
actors
<<abstract>>
Actor
1
actions
<<abstract>>
Action
game
wields
targets
playable_characters
classes
Player
1
selects
1
<<abstract>>
Classes
1
Samurai
Bandit
Wretch
skills
Unsheathe
Quickstep
<<abstract>>
Skills
1
uses
1
actions
1
QuickStepAction
1
1
possesses
weapons
1
<<abstract>>
TradeableWeapon
Great Knife
Club
Uchigatana

SCENARIO: Player performs Unsheathes and kills LoneWolf, gaining between 55 and 1470 Runes.

(All that shown here is ONLY related to Requirement 4 and the scenario given)

:SpecialAttackAction
actor:Player
skills:Skills
target:LoneWolf
map:GameMap
execute(actor, map)
getAccuracy()
accuracy: int
Alt
[!(getRandomInt(1, 100) <= hitRate)]
getName()
name: String
result: actor name but misses target
[Else]
getDamage()
damage: int
result: actor names and hits target for damage damage
hurt(damage)
isConcious()
hitpoints > 0: boolean
Opt
[!target.isConscious()]
<<creates>>
:DeathAction
execute(target, map)
addCapability(Status.DEAD)
hasCapability(Status.CAN_GAIN_RUNES)
capabilitySet.contains(capability): boolean
hasCapability(EnemyStatus.CAN_DROP_RUNES)
capabilitySet.contains(capability): boolean
hasCapability(EnemyStatus.RESURRECTABLE)
capabilitySet.contains(capability): boolean
Opt
[actor.hasCapability(Status.CAN_GAIN_RUNES) && (target.hasCapability(EnemyStatus.CAN_DROP_RUNES)) && !(target.hasCapability(EnemyStatus.RESURRECTABLE))]
:RuneManager
getInstance(actor)
runeManager: RuneManager
gainRunes(actor, target)
getItemInventory()
itemInventory: List<Item>
Loop
[for each item in itemInventory]
<<creates>>
dropActions: ActionList
add(item.getDropAction(target))
drop: DropAction
getWeaponInventory()
weaponInventory: List<WeaponItem>
Loop
[for each weapon in weaponInventory]
add(weapon.getDropAction(target))
Loop
[for each action in dropAction)]
execute(target, map)
hasCapability(EnemyStatus.RESURRECTABLE)
capabilitySet.contains(capability): boolean
Alt
[target.hasCapability(EnemyStatus.RESURRECTABLE]
locationOf(target)
removeActor(target)
increaseMaxHp(points)
addActor(pileOfBones)
[Else]
removeActor(target)
result: target was killed by actor
result: String

Group: MA_Lab04_Group5

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

Design Rationale (Requirement 4)

Classes

Due to sharing common attributes, like a starting weapon or hitpoints, the **Samurai**, **Bandit** and **Wretch** classes all inherit from the **Classes** abstract class. Through this, we avoid duplicating code redundantly, following the DRY principle (Pros). However, the usage of inheritance may lead to a rigid hierarchy that makes it harder to extend in the future. One such example is if in the future a new **Class**, which does not share the same attributes with the current classes, is added, it may not fit into the existing hierarchy, resulting in it violating the OCP (Cons).

The **Player** will have an instance variable that sets the chosen **Class** and their hitpoints and staring weapon will be determined by the **Classes** chosen. This obeys the LSP, as all instances of **Classes** can be substituted with its subclasses, without inducing any side effects in the code. Through this, the code will be more flexible and easier to extend as any newly created **Classes** can inherit from the **Classes** abstract class, allowing shared functionality (Pros) (Future extension).

Weapons

Similarly to the above, we apply the DRY concept where the **Uchigatana**, **Club**, and **Great Knife** classes inherit from the **TradableWeapon** abstract class. By using this approach, the amount of redundant, repeated code is minimised, making code maintenance simpler (Pros). However, new **Weapons** that are not Tradeable will not be able to inherit from the **TradableWeapon** due to its rigid nature, resulting in a violation of OCP (Cons).

Both **Uchigatana** and **Great Knife** will have an association with the **Skills** abstract class, where they will have their respective **Skills** (**Unsheate** and **Quickstep** classes). This is used in the **SpecialAttackAction** to execute the attack via the **Skills** attribute. In the case of **Unsheate**, the damage will be twice the damage of **Uchigatana**, along with the accuracy being 60%. Through this, any new **Skills** in the future that can be added that only change the **WeaponItem** damage/accuracy can reuse, following the OCP as the classes are easily extendable without any modification required (Pros) (Future extension).

Quickstep is slightly different, as it has additional functionality (evading after an attack). To account for this, a separate **Action** is created - **QuickStepAction**. The **QuickStepAction** will first execute a **SpecialAttackAction** to perform a **Skill** attack. This is additional proof that our code is extendable as **SpecialAttackAction** can be reused in **QuickStepAction**, following the DRY principle as duplicated code is reduced (Pros). Then in **QuickStepAction**, **MoveActions** are created to all adjacent possible **Exits** and added to a list. One **MoveAction** is chosen by random and is executed after the attack (evade). However, technically this violates the SRP as the **QuickStepAction** has two responsibilities, performing a **Skill** attack, and then evading afterwards, resulting in increased complexity which makes the code harder to maintain (Cons).

engine

weapons

<<abstract>>
WeaponItem

game

environments

<<abstract>>
SpawnableGround

Puddle of Water

Gust of Wind

Graveyard

<<spawns>>

<<spawns>>

<<spawns>>

enemies

east

Giant Crayfish

Giant Dog

Skeletal Bandit

1

EastSpawningFactory

west

Heavy Skeletal Swordsman

1

LoneWolf

Giant Crab

WestSpawningFactory

1

<<abstract>>
Marine

<<abstract>>
Animal

<<abstract>>
Undead

Pile of Bones

<<enumeration>>
EnemyStatus

0..*

uses

1

<<abstract>>
Enemy

GeneralSpawningFactory

<<abstract>>
SpawningFactory

weapons

<<interface>>
Buyable

<<interface>>
Sellable

<<abstract>>
TradeableWeapon

1

possesses

Scimitar

<<uses>>

<<enumeration>>
WeaponStatus

GiantCrayfishPincer

1

possesses

GiantDogHead

1

possesses

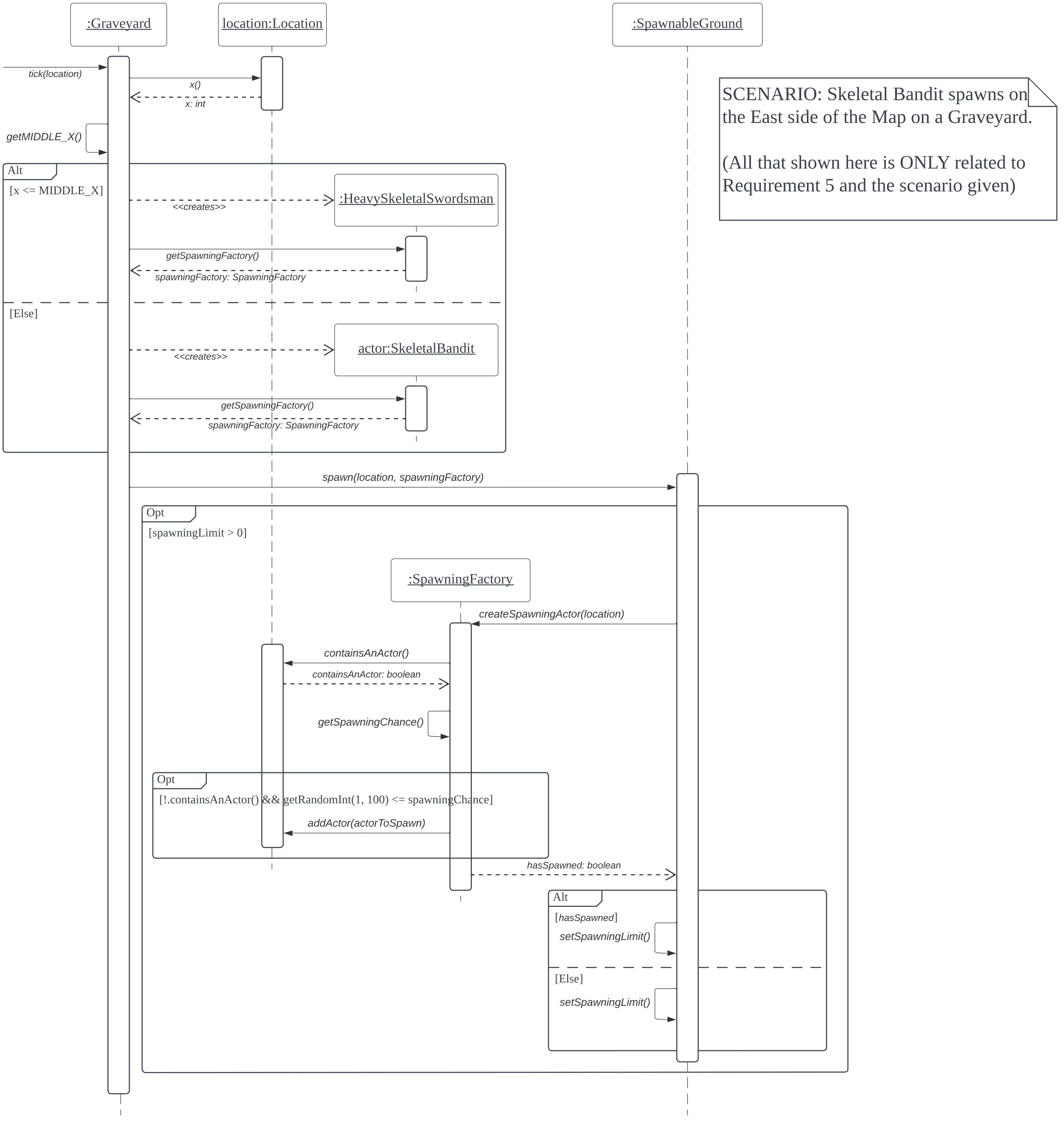
skills

1

<<abstract>>
Skills

SpinAttack

SlamAttack


:Graveyard
location:Location
:SpawnableGround
tick(location)
x()
x: int
getMIDDLE_X()
Alt
[x <= MIDDLE_X]
<<creates>>
:HeavySkeletalSwordsman
getSpawningFactory()
spawningFactory: SpawningFactory
[Else]
<<creates>>
actor:SkeletalBandit
getSpawningFactory()
spawningFactory: SpawningFactory
spawn(location, spawningFactory)
Opt
[spawningLimit > 0]
:SpawningFactory
createSpawningActor(location)
containsAnActor()
containsAnActor: boolean
getSpawningChance()
Opt
[!.containsAnActor() && getRandomInt(1, 100) <= spawningChance]
addActor(actorToSpawn)
hasSpawned: boolean
Alt
[hasSpawned]
setSpawningLimit()
[Else]
setSpawningLimit()
SCENARIO: Skeletal Bandit spawns on the East side of the Map on a Graveyard.
(All that shown here is ONLY related to Requirement 5 and the scenario given)

Group: MA_Lab04_Group5

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

Design Rationale (Requirement 5)

Enemies

To distinguish the different enemies that spawn specifically on the west side of the east, we separated the enemies (**Heavy Skeletal Swordsman**, **Lone Wolf**, and **Giant Crab)** into a **west** package, with the rest being in an **east** package.

Similar to Requirement 1, the 3 enemies **Giant Crayfish**, **Giant Dog** and **Skeletal Bandit** are spawned by their respective **SpawnableGround** (**Puddle of Water**, **Gust of Wind** and **Graveyard)** which implements the **Spawnable** interface. The **Enemies** that spawn on the east side of the map will have an **EastSpawningFactory** instance (**Giant Crayfish**, **Giant Dog** and **Skeletal Bandit**) with the **Enemies** on the west having a **WestSpawningFactory** instance (**LoneWolf**, **Giant Crab** and **Heavy Skeletal Swordsman**). This is because the west and east **Enemies** each have different spawning chances, proving our code is extendable and thus adheres to OCP (Pros). Finally, **PileOfBones** has a **GeneralSpawningFactory** instance instead, as they are not limited to where they spawn.

The “spawn” method will check for the **Location** x-coordinate and spawn the appropriate west or east enemy, creating it and passing it to the **SpawningFactory** "createSpawningActor" method. As the "spawn" takes in a **SpawningFactory** abstract class, both the **EastSpawningFactory** and **WestSpawningFactory**, which inherits from **SpawningFactory**, can be passed into it with no issue, showing that our code follows the LSP as **SpawningFactory** can be substituted with its subclasses without any modification required for any additional extension (Pros). On the other hand, this method means the "spawn" method will only spawn the respective **Enemies** on a certain side of the map. For example, **Graveyard** will only spawn **Skeletal Bandit** on the East side of the map. This may potentially violate OCP as in the future if a new **Undead Enemy** is added which can spawn on the East side of the map, it may require modifications to implement this additional functionality (Cons).

Not only that, these enemies inherit from their respective types (**Skeletal Bandit** inherits from **Undead**, **Giant Dog** inherits from **Animal**, and **Giant Crayfish** inherits from **Marine**) which all will inherit from the **Enemy** abstract class. This will prevent **Enemies** from attacking their own types as **Skeletal Bandit** for example will have the capability “Status.NOT_HOSTILE_TO_UNDEAD” just like **Heavy Skeletal Swordsman**. The **Enemy** itself will implement the **Despawnble** interface, enforcing a "despawn" method for its subclasses. Here **Skeletal Bandit**, **Giant Dog** and **Giant Crayfish** will create a new **DespawnAction** provided that the returned value of the "despawn" method is true. Through this, we have demonstrated that our code follows OCP, as it’s easily extendable by adding new enemies without impeding our earlier code (Pros). On the other hand, if future non-**Despawnable Enemies** are added, they will not be able to inherit from our rigid **Enemy** abstract class, forcing us to create new classes to cover them. This results in a more complex or unnecessary code, violating the DRY principle (Cons).

Other than that, these enemies employ similar design aspects such as behaviours (**FollowBehaviour**, **WanderBehaviour** and **AttackBehaviour**), couldn't attack their type, actions (**AttackAction**, **AreaAttackAction**, **SpecialAttackAction** and **DeathAction**), skills (**SpinAttack** and **SlamAttack**), dropping **Runes** upon death, spawning and despawning method as the enemies on the west side of the map. These design aspects would follow their respective design principles laid out in previous requirements accordingly as well.

Weapons

Similarly, the **Grossmesser** class and the other weapons, the **Scimitar** class inherit from the **TradeableWeapon** abstract class. Through this **Scimitar** will implement the **Buyable** and **Sellable** interfaces by proxy. The **Scimitar** will be added to the **Trader** inventory to allow it to be purchasable by the **Player**. **Scimitar** will inherit the "buy", "getBuyingPrice", "setBuyingPrice", "sell", "getSellingPrice" and "getBuyingPrice" from the **TradeableWeapon**, in addition to having the "WeaponStatus.TRADEABLE", allowing it to be tradeable. **Scimitar** will be in the inventory of the **Skeletal Bandit**. Through this, we have successfully demonstrated OCP in our code as its open for extension but closed for modification as shown above (Pros). The other **Enemies** added (**GiantDog** and **GiantCrayfish**), will have specialised weapons for them (**GiantDogHead** and **GiantCrayfishPincers**).

As all these 3 weapons added can perform Area Attacks, they will override the "getSkill" method, returning a newly created **AreaAttackAction**. **Scimitar** will have a **SpinAttack Skills** instance, with **GiantDogHead** and **GiantCrayfishPincers** having a **SlamAttack Skills** instance. Although **GiantDog** and **GiantCrayfish SlamAttack** will have the same damage/accuracy as their normal attacks, this may not always be the case in the future. As **AreaAttackAction** utilises **SpecialAttackAction**, which uses **Skills** data attributes to determine their damage/accuracy, any future **SlamAttack** which has its own damage/accuracy can be easily added without modification to the **AreaAttackAction**, which follows OCP (Pros) (Future extension). However, both **SpinAttack** and **SlamAttack,** as of the current state of the game, have similar functionality (performing an **AreaAttackAction** with their **WeaponItem**) with the only difference being the verb used. This means that there is redundant, repeated code, which violates the DRY principle, resulting in more complex code (Cons).