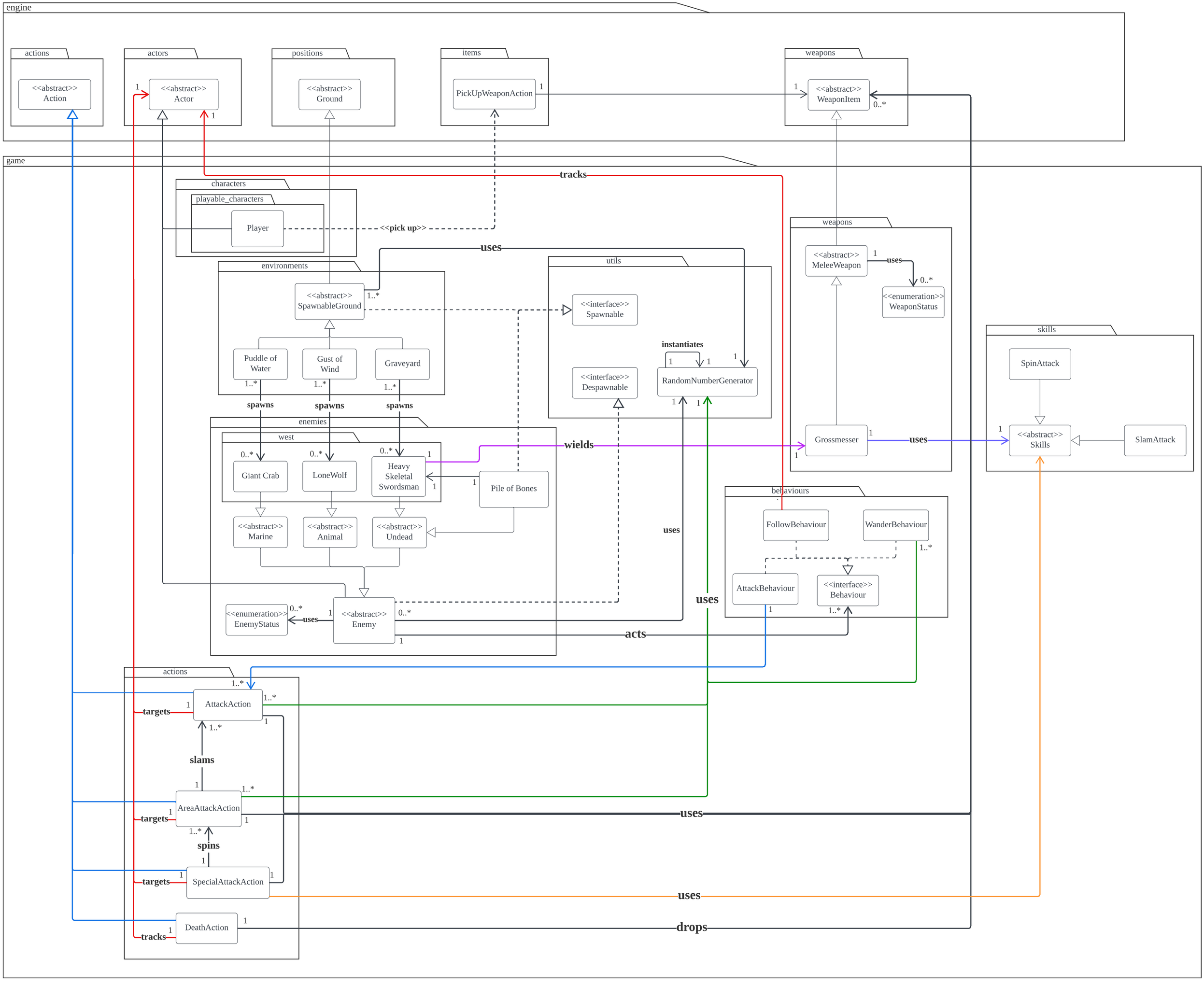
Group: MA_Lab04_Group5

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
items
PickUpWeaponAction
1
weapons
1
<<abstract>>
WeaponItem
0..*
game
tracks
characters
playable_characters
Player
<<pick up>>
uses
environments
<<abstract>>
SpawnableGround
1..*
Puddle of
Water
Gust of
Wind
Graveyard
1..*
1..*
1..*
spawns
spawns
spawns
utils
<<interface>>
Spawnable
<<interface>>
Despawnable
instantiates
1
1
1
RandomNumberGenerator
1
1
weapons
<<abstract>>
MeleeWeapon
1
uses
0..*
<<enumeration>>
WeaponStatus
Grossmesser
1
uses
1
skills
SpinAttack
1
<<abstract>>
Skills
SlamAttack
enemies
west
0..*
0..*
0..*
1
Giant Crab
LoneWolf
Heavy
Skeletal
Swordsman
1
wields
1
Pile of Bones
<<abstract>>
Marine
<<abstract>>
Animal
<<abstract>>
Undead
uses
<<enumeration>>
EnemyStatus
0..*
uses
1
<<abstract>>
Enemy
0..*
1
behaviours
FollowBehaviour
WanderBehaviour
1..*
AttackBehaviour
1
<<interface>>
Behaviour
1..*
uses
acts
actions
1..*
AttackAction
1..*
targets
1
1
1..*
slams
1
AreaAttackAction
1..*
targets
1
1
1..*
spins
1
SpecialAttackAction
targets
1
1
uses
DeathAction
1
tracks
1
drops

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. This design thus adheres to the DRY (Don't Repeat Yourself) concept. The **SpawnableGround** class is then inherited by the **Graveyard**, **Gust of Wind**, and **Puddle of Water** classes, enabling them to share functionality and reduce code duplication. In addition, the **RandomNumberGenerator** class will be used in the **SpawnableGround** class to determine the chances of enemies spawning for each respective environment. Furthermore, this 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 and easily extendable, in which any additional spawning ground can inherit from **SpawnableGround** and implement its own "spawn" function. Also, **RandomNumberGenerator** is a factory class. 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.

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. 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 are added. Not only that, the subclasses have access to all similar functionalities to the parent **Enemy** class.

Similar to **Spawnable**, **Enemy** will implement the **Despawnable** interface that will enforce a “despawn” function to despawn the enemies. 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 types using the **EnemyStatus** enumeration. 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.

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 (Open Closed Principle) as we can add new behaviour in the **Pile of Bones** without modifying the **Heavy Skeletal Swordsman**. After 3 turns, it checks if the **Pile of Bones** is destroyed. If not, the **Heavy Skeletal Swordsman** will be spawned back and replaced with the **Pile of Bones**. Otherwise, it drops the weapon **Grossmesser** which is in the **Pile of Bones** inventory. In addition, the **Pile of Bones** implements the **Despawnable** interface, as once the 3 turns have passed, the **Pile of Bones** will "despawn" before the **Heavy Skeletal Swordsman** respawns. This too adheres to OCP. 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. Furthermore, this provides more flexibility in the future, as adding new classes that require spawning, despawning or both can be implemented without the two interfering with each other as shown in the **Pile of Bones** class.

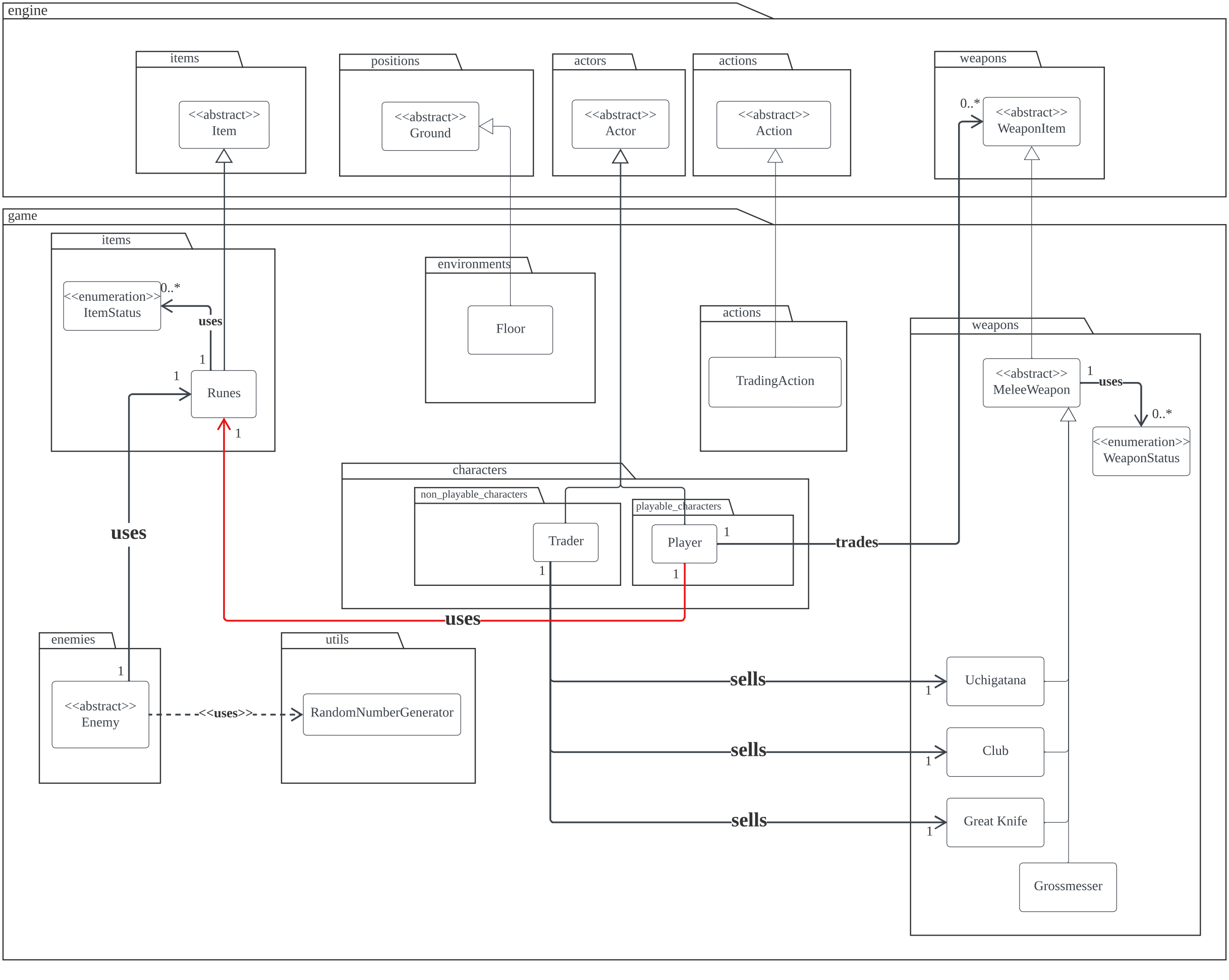
The **Behaviour** interface is implemented by the **FollowBehaviour**, **WanderBehaviour** and **AttackBehaviour**, enabling loose coupling. We follow SRP by creating separate classes for each type of behaviour. This allows each class to have a distinct role, making it simpler to comprehend. Not only that, using a single association line from the **Enemy** class to the **Behaviour** interface removes multiple dependencies for the **Enemy** class on multiple attack actions and behaviours. Moreover, this obeys the DIP (Dependency Inversion Principle) as the **Enemy** relies on the **Behaviour** interface instead of the **WanderBehaviour** or **FollowBehaviour.** Because of this, it's insulated from any change in the **Behaviour** subclass, leading to easily modifiable and future-proof code. For this reason, we don’t always need to fix the **Enemy** class to another attack action or behaviour if they are added (OCP). 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 “EnemyStatus.FOLLOWING” capability if **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. Lastly, the **AttackBehaviour** class will employ a behaviour to the enemies that will attack the **Player** or other types of enemies with the **AttackAction** class.

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. The **AttackAction** class, firstly, is used to execute an attack from the perpetrator to victim whereas the **AreaAttackAction** class will have a list of **AttackActions**, used for slam/spin attacks where multiple **AttackActions** are executed simultaneously. Using the “hasCapability” method, **AreaAttackAction** can use the **EnemyStatus** enumeration “EnemyStatus.CAN_AREA_ATTACK” to check if the enemy can perform an area attack. The **RandomNumberGenerator** class will be used in the **AttackAction** class to determine the attack’s accuracy and will be calculated independently for each **AttackAction** executed. Whenever a **Skill** is used, **SpecialAttackAction** is used. For example, **SpinAttack,** which inherits from **Skills**, will be used by the **Heavy Skeletal Swordsman** wielding the **Grossmesser** via **SpecialAttackAction**. The **DeathAction** class

will check if an enemy can be resurrected with the enum “EnemyStatus.RESURRECTABLE” using the “hasCapability” method.

Weapons

The **Grossmesser** class uses the **RandomNumberGenerator** class to determine whether the **Heavy Skeletal Swordsman** will perform a normal attack or a spin attack. **Grossmesser** inherits from the **MeleeWeapon** abstract class that uses the **WeaponStatus** enum class. This enables the **Grossmesser** to be droppable using the enum “WeaponStatus.DROPPABLE” and perform the spin attack using the enum “WeaponStatus.CAN_AREA_ATTACK”. The **Grossmesser** class uses the **Skills** abstract class which is inherited by the **SpinAttack** class for its spin attack. This follows LSP as having a child class of **Skills** we can reuse existing functionality without modifying the superclass.


engine
items
<<abstract>>
Item
positions
<<abstract>>
Ground
actors
<<abstract>>
Actor
actions
<<abstract>>
Action
weapons
0..*
<<abstract>>
WeaponItem
game
items
<<enumeration>>
ItemStatus
0..*
uses
1
1
Runes
1
environments
Floor
actions
TradingAction
weapons
<<abstract>>
MeleeWeapon
1
uses
0..*
<<enumeration>>
WeaponStatus
characters
non_playable_characters
Trader
1
playable_characters
Player
1
1
trades
uses
uses
enemies
1
<<abstract>>
Enemy
<<uses>>
utils
RandomNumberGenerator
sells
1
Uchigatana
sells
1
Club
sells
1
Great Knife
Grossmesser

Group: MA_Lab04_Group5

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

Design Rationale (Requirement 2)

Runes

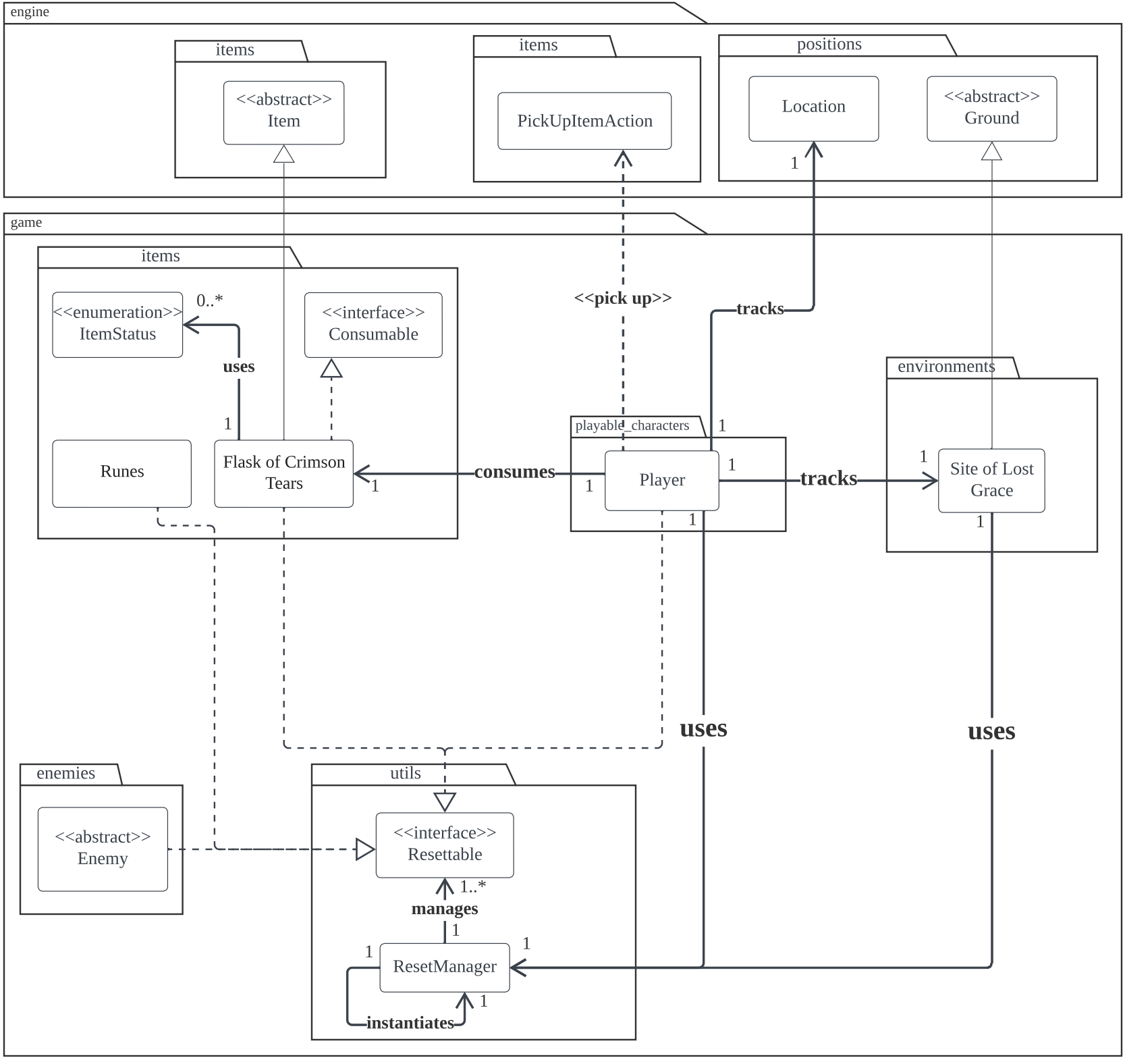
Runes inherit from the Items abstract class and do not have “ItemStatus.TRADEABLE”, thus it is untradeable but has “ItemStatus.DROPPABLE”, as the **Player** will drop them when killed. Each **Enemy** will use **RandomNumberGenarator** to generate a random amount of runes in a set range that will be given to the **Player** when killed. 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. Likewise, additional **ItemStatus** capabilities can be added, making it easily extendable. **Runes** have a “runeCount” variable, keeping track of how many runes the **Player** has.

Trader

The **Trader** will spawn on the **Floor**. The **Floor** class utilises the “canActorEnter” function that only allows **Actors** with the “Status.PLAYER” to enter, preventing **Enemies** from waltzing into the building. **TradingAction** is used to facilitate buying and selling between the **Player** and **Trader**. **Uchigatana**, **Club** and **Great Knife** will be in the **Trader**'s inventory, ready to be bought by the **Player**. **Grossmesser** can be sold to the **Trader**. All these weapons will have the “WeaponStatus.TRADEABLE” capability, allowing them to be traded for runes. This provides additional functionality in the future if untradeable weapons are added, thus it follows the OCP as extensions can be effortlessly made without accidentally causing side effects to other code.

Weapons

Weapons have a “buyingPrice” and “sellingPrice” variable which determines their price during a **TradingAction**. The **Player** “runeCount” will be updated based on the respective **TradingAction** performed. For example, if they bought the **Club**, their “runeCount” in the **Rune** class will subtract by 600.


engine
items
<<abstract>>
Item
items
PickUpItemAction
positions
Location
<<abstract>>
Ground
1
game
items
<<enumeration>>
ItemStatus
0..*
uses
1
<<interface>>
Consumable
Runes
Flask of Crimson
Tears
1
<<pick up>>
tracks
consumes
playable_characters
1
1
Player
1
1
tracks
environments
1
Site of Lost
Grace
1
uses
uses
enemies
<<abstract>>
Enemy
utils
<<interface>>
Resettable
1..*
manages
1
1
ResetManager
1
1
instantiates

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", 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. 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. It will not have the "ItemStatus.DROPPABLE" capability, preventing the **Player** from dropping it, which also follows the concept of OCP. Not only that, it also implements the **Resettable** interface to reset the max uses which adheres to ISP and SRP. 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.

Site of Lost Grace

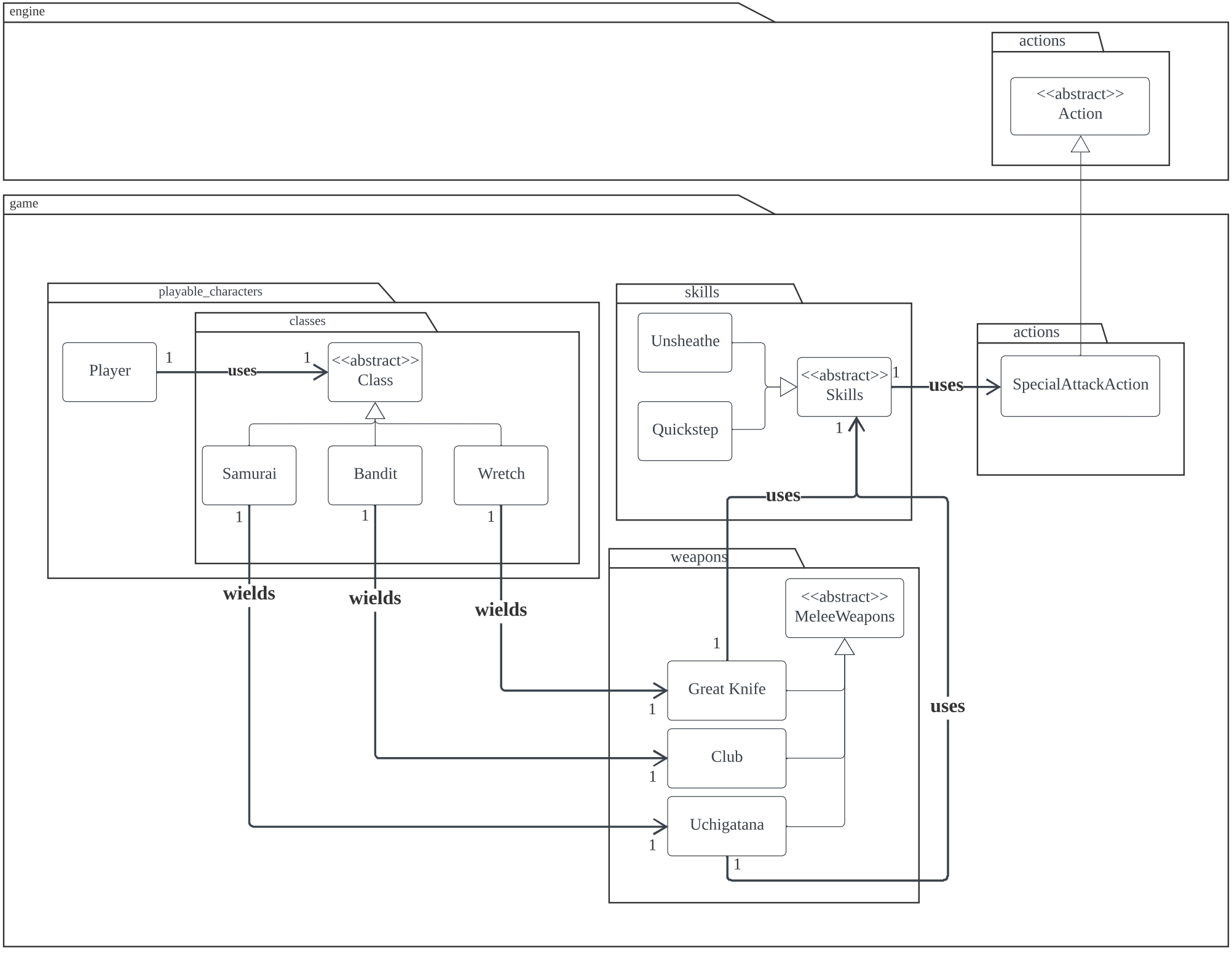
The **Site of Lost Grace** inherits from **Ground** and when the **Player** rests on it, the game resets. The **Player** will have an association relationship with the **Site of Lost Grace**, saving the specific **Site of Lost Grace** after resting. Once the **Player** is killed, they will respawn back at the saved **Site of Lost Grace**. 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.

Game Reset

The factory 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. Objects that reset when the **Player** rests implement the **Resettable** interface, forcing all subclasses to implement the “reset” function and follow the DRY concept. This would be executed in the **ResetManager,** resetting the object**,** for example, the **Flask of Crimson Tears** would reset its “uses” variable back to 2 using the “reset” method. Moreover, the SRP is used as the **ResetManager** has only one responsibility, resetting the **Resettable**. **ResetManager** is also called when the **Player** is killed.

Runes

The **Player** uses the **Location** class to store the current position before any **MoveAction** is conducted. The **Runes** will then be dropped at that stored **Location** if the **Player** is killed. Not only that, the **Player** utilises a **PickUpItemAction** method when standing next to the dropped **Runes**, allowing them to pick them up. Within the **Player** class, there is a count variable representing the number of deaths. This is used to keep track of whether the **Player** has already died twice without picking up their dropped **Runes**. If so, the **Runes** will be removed from the map since it implemented the **Resettable** interface. In these situations, or when the **Player** manages to pick up their dropped **Runes**, said death counter will be reset back to 0.


engine
actions
<<abstract>>
Action
game
playable_characters
classes
Player
1
uses
1
<<abstract>>
Class
Samurai
Bandit
Wretch
1
1
1
wields
wields
wields
skills
Unsheathe
Quickstep
<<abstract>>
Skills
1
1
uses
actions
SpecialAttackAction
uses
weapons
<<abstract>>
MeleeWeapons
1
Great Knife
1
Club
1
Uchigatana
1
1
uses

Group: MA_Lab04_Group5

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

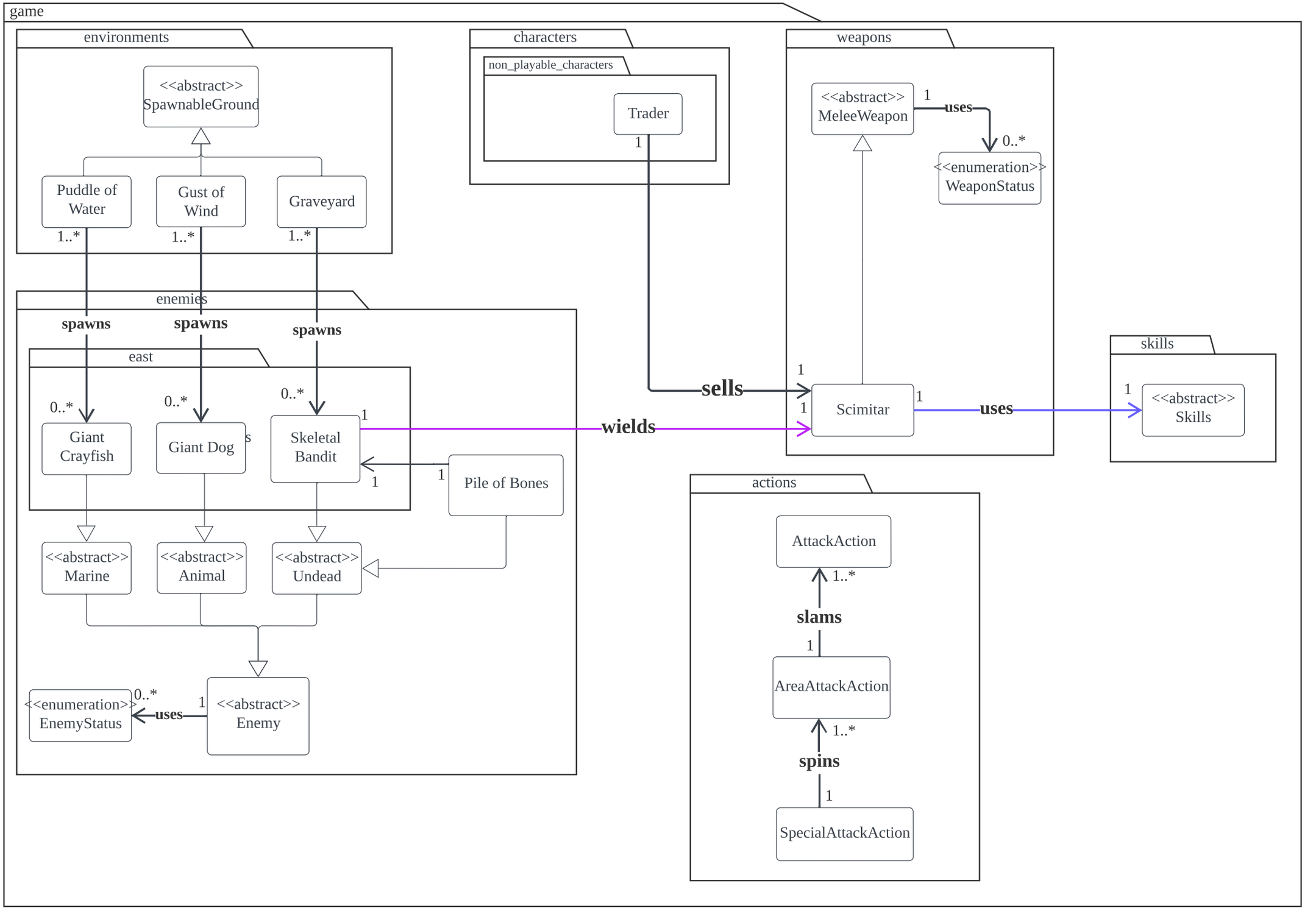
Design Rationale (Requirement 4)

Classes

Due to sharing common attributes, like a starting weapon or hitpoints, the **Samurai**, **Bandit** and **Wretch** classes all inherit from the **Class** abstract class. Through this, we avoid duplicating code redundantly, following the DRY principle. The **Player** will have an instance variable that sets the chosen **Class**, drastically reducing the dependencies required. Furthermore, this obeys the LSP, as all instances of **Class** 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. Likewise, the **Class** class implements the SRP as it only has one responsibility - managing the **Player** classes.

Weapons

Similarly to the above, we apply the DRY concept where the **Uchigatana**, **Club**, and **Great Knife** classes inherit from the **MeleeWeapon** abstract class. By using this approach, the amount of redundant, repeated code is minimised, making code maintenance simpler. Since both **Uchigatana** and **Great Knife** have their respective skills (**Unsheathe** and **Quickstep**), they will inherit from the **Skills** abstract class. This is done due to both **Unsheathe** and **Quickstep** inheriting from **Skills**, which obeys the OCP as new skills that could be potentially added later will again be inherited from **Skills**, minimising the addition of unnecessary relationships. The **Skills** class will contain all necessary data attributes about the skills, like the attack accuracy or damage. In addition, **SpecialAttackAction** will utilise the **Skills** class to execute a skill, adhering to the LSP as both **Skills** and its subclasses can be used interchangeably.


game
environments
<<abstract>>
SpawnableGround
Puddle of
Water
Gust of
Wind
Graveyard
1..*
1..*
1..*
characters
non_playable_characters
Trader
1
weapons
<<abstract>>
MeleeWeapon
1
uses
0..*
<<enumeration>>
WeaponStatus
enemies
spawns
spawns
spawns
east
0..*
0..*
0..*
Giant
Crayfish
Giant Dog
Skeletal
Bandit
1
1
1
Pile of Bones
sells
1
1
Scimitar
1
uses
skills
1
<<abstract>>
Skills
wields
<<abstract>>
Marine
<<abstract>>
Animal
<<abstract>>
Undead
<<enumeration>>
EnemyStatus
0..*
uses
1
<<abstract>>
Enemy
actions
AttackAction
1..*
slams
1
AreaAttackAction
1..*
spins
1
SpecialAttackAction

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 "spawn" method will check for the **Location** x-coordinate and spawn the appropriate west or east enemy. 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**). The environments and types still adhere to the DRY concept and LSP as they all remain inherited from the **Enemy** abstract class. Through this we have demonstrated that we implemented these principles successfully, proving our code is easily extendable by adding new enemies without impeding our earlier code.

Another similar aspect to Requirement 1, the **Skeletal Bandit** will also turn into a **Pile of Bones** after it dies. Therefore, the **Pile of Bones** will replace the **Skeletal Bandit** when it dies. Similarly, it follows OCP as new behaviour can be added into the **Pile of Bones** without modifying the **Skeletal Bandit**. Once 3 turns have passed, if the **Pile of Bones** hasn't been destroyed yet, the **Skeletal Bandit** will respawn back and replace the **Pile of Bones**. If killed, it drops the weapon **Scimitar** from the **Pile of Bones** inventory.

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 **MeleeWeapon** abstract class. Since we know the **MeleeWeapon** abstract class uses the **WeaponStatus** enum class, the **Scimitar** will also be made droppable using the "WeaponStatus.DROPPABLE", perform a spin attack using the enum "WeaponStatus.CAN_AREA_ATTACK", and tradeable using the enum "WeaponStatus.TRADEABLE". This offers the code to be open for extension but closed for modification, which is the OCP. Firstly, LSP can be found where the **Scimitar** uses the **Skills** abstract class inherited by the **SpinAttack** class for its spin attack. This is so that we may reuse the functionality of the parent class without changing the child class, Not only that, the **Scimitar** will be in the **Trader's** inventory, ready to be bought by the **Player** with **Runes** and sold to the **Trader** for **Runes**. This means the **Scimitar** has a "buyingPrice" and

“sellingPrice” variable like other weapons which determines their price during a **TradingAction**. Hence, the **Player** “runeCount” will be updated based on the respective **TradingAction** performed. Since SRP is used extensively in our code, all these extra extensions we made (like **Scimitar**) can be easily extended without modifying the base code.