Engine
positions
Location
Has
1
1
Ground
actors
Actor
actions
Action
weapons
<<Interface>> Weapon
Implements
Implements
WeaponItem
IntrinsicWeapon
game
<<Uses>>
Extends
Extends
Extends
enums
<<enumeration>> Status
<<enumeration>> Species
1..4
1
Has
Has
behaviours
1..4
<<Interface>> Behaviour
<<Returns>>
Has
Implements
Implements
Implements
Implements
SkillBehaviour
AttackBehaviour
FollowBehaviour
WanderBehaviour
Extends
Extends
Extends
enemies
Mortal
1
<<Returns>>
Extends
1
Enemy
1
EnemyFactory
<<Returns>>
<<Uses>>
<<Uses>>
Extends
Extends
Extends
Extends
PileOfBones
LoneWolf
GiantCrab
HeavySkeletalSwordsman
1
<<Returns>>
environments
Spawner
Extends
Extends
Extends
Graveyard
GustOfWind
PuddleOfWater
gameactions
skillactions
Implements
<<Interface>> Skilled
Implements
<<Returns>>
SkillAction
Extends
Extends
Slam
SpinningAttack
deathactions
DeathAction
Extends
Extends
SkeletonDeathAction
StandardDeathAction
standardactions
AttackAction
gameweapons
Grossmesser
1
<<Uses>>
<<Returns>>
<<Returns>>
Has

Requirement 1 - Environments & Enemies

The diagram represents an object-oriented design for requirement 1 and displays the necessary classes and their relationships with each other in order to fulfil the needs of the overall system. The diagram is organised by package, in the top half we have classes from the provided game engine while the bottom half displays the classes and packages within the game package which have been modified and extended to implement the required features.

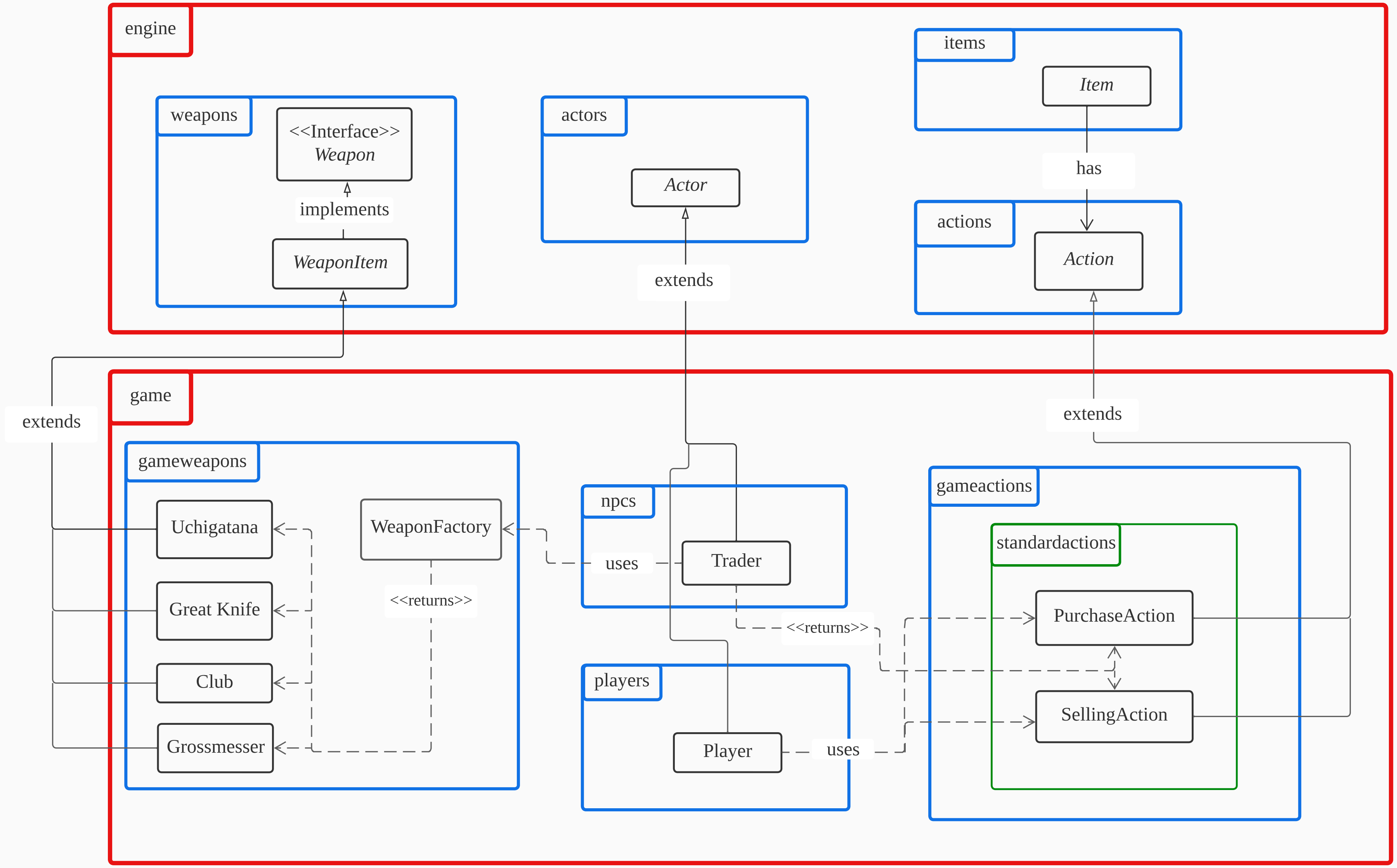
The first design choice I would like to highlight is the inclusion of the abstract Spawner class that extends the Ground abstract class in the engine. I chose to design the system in such a way so that the Ground classes that spawn enemies such as Graveyard, PuddleOfWater and GustOfWind will inherit from this Spawner class and are forced to implement its abstract methods. This adheres to the DRY (Do not repeat yourself) design principle as all 3 of these concrete classes use similar logic to spawn enemies which can be defined within the Spawner abstract class itself. This also makes the design highly extensible as if we decide to change how spawning behaviour is managed in the future, we can simply make changes to the Spawner class directly instead of having to modify every Ground class which spawns enemies. This approach also adheres to the open and closed principle, as the Spawner class is meant to be extended by its children to spawn specific types of enemies instead of directly modified itself. Additionally, this design obeys the LSP (Liskov substitution principle) as Spawner objects can be used in Place of any Ground object without breaking the system. The one con of this design is that we end up coupling the Spawner class and its children quite tightly due to their inheritance relationship, however this is not a significant disadvantage as the Graveyard, PuddleOfWater and GustOfWind classes all follow the SRP (Single responsibility principle) and are not overly complex or bloated as most of their needs have been abstracted into the Spawner abstract class.

The next design choice I want to highlight is the use of the factory design pattern by implementing an EnemyFactory class. This class follows the SRP principle as its only responsibility is to create and return new Enemy objects through the use of static methods. I have chosen to handle the creation of objects in this manner as it allows us to make use of IoC (Inversion of control) and DI (Dependency injection) patterns when dealing with other Classes that depend on Enemy Objects such as the Spawner class and its children as well as a few other classes that spawn enemies. This makes our design more testable and robust as the objects which depend on the Enemy class won't have to worry about creating the object themselves resulting in loosely coupled classes. The alternative to this approach would have been to implement the EnemyFactory class as a singleton object with a static getFactory method, however due to the stateless nature of this class it is unnecessary to maintain an instance of it as the creation of new EnemyObjects can be done statically. The con of this design is that it creates some mental debt as the creation of enemies has now been abstracted into the factory, thus every time a new Enemy class is created, the developer must remember to add a static factory method to the EnemyFactory class.

The third design choice I would like to highlight is the inclusion of a Mortal abstract class which extends Actor and Enemy abstract class which extends Mortal. The idea behind this is that the mortal abstract class defines an actor who is capable of dying, and hence should have some logic that supports DeathActions while the Enemy class defines an actor who is an enemy to the player which will involve the use of Behaviours, specific enums, and SkillActions. Since all enemies can die, it would also make sense to have Enemy extend Mortal as now all children of the Enemy class must implement support for DeathActions. I have chosen not to implement Mortal as an interface for the key reason that death and actors are closely related in that many children of the actor class will need to support DeathActions while many objects that implement death require methods from the Actor class such as the hurt method, hence it makes sense to involve it in the inheritance hierarchy. You may then ask however, why I choose to separate this functionality into two separate abstract classes rather than simply baking it all into the Enemy abstract class, this decision was made in order to adhere to the Interface segregation principle, not every Actor who can die is necessarily an Enemy, hence why should Actors such as the Player have to implement methods from the Enemy class which is does not depend on or use. Furthermore, this architecture also adheres to the DRY principle, as a lot of code in concrete enemy classes is reused from abstract classes, the LSP principle, as all children of Mortal and Enemy can be substituted for any Actor object, the SRP principle as the Enemy and Mortal classes have a single well defined responsibility, and finally the DI (Dependency inversion) as other parts of the system such as DeathActions and Spawners don't have to depend on concrete implementations of a class and can instead depend on the abstractions provided by Mortal and the Enemy class respectively. Ultimately, this approach allows for greater extensibility as now that death and enemies are not coupled, we can extend the system more freely as not every Actor needs to extend Enemy to be Mortal. A major con of this approach however is that the system has become more complicated due to the higher number of abstractions.

Another design choice worth highlighting is how behaviours have been implemented. Since all behaviours implement the Behaviour Interface, they must implement the GetAction method which returns an Action. This follows the design by contract paradigm as we have defined a formal, precise and verifiable interface within the Behaviour interface that will allow other components of our system to extend and use it easily.

The fourth major design choice made is the inclusion of a DeathAction abstract class, SkillAction abstract class and Skilled interface. DeathActions and SkillActions are both a form of action and naturally extend the Action class, however they are both abstract as there is no single concrete definition of a death or skill action in the game that would be sufficient to cover all required features. For example, when a normal enemy dies, it simply drops runes and weapons, however when the HeavySkeletalSwordsman dies, it spawns a PileOfBones instead of dropping runes and weapons. Hence, it's cleaner to loosely define how a DeathAction or SkillAction should behave through abstract methods and leave the detailed implementation to the concrete children of the class to implement. This fulfils the open and closed principle as both abstract classes are closed to modification while their children are open to extend their functionality. This also adheres to the LSP principle as DeathActions, SkillActions and their children can be used in place for any Action object without breaking the system. The Skilled interface is similar to the Mortal abstract class such that it was designed to allow other classes to use a specific action, in this case it was created as a way of providing other classes a common interface for interacting with SkillAction objects. However, it was designed as an interface instead of an abstract class as the classes that use SkillActions are Grossmesser which is a WeaponItem and GiantCrab, an Enemy. Since these classes are not closely related, it would not make sense to create an inheritance relationship between them. The use of the Skilled interface applies the DI (Dependency inversion) principle. In this case, the high-level modules GiantCrab and Grossmesser do not depend on the concrete implementation of skills but rather on the abstraction layer provided by the Skilled interface. The con behind this design however is that it introduces a lot of extra classes and complexity into the system, though I believe this will ultimately be worth it as this approach will allow for greater extensibility in the future as more Skill and Death actions can be added in the future without worrying about higher level modules breaking.


engine
weapons
<<Interface>>
Weapon
implements
WeaponItem
actors
Actor
extends
items
Item
has
actions
Action
game
extends
extends
gameweapons
Uchigatana
Great Knife
Club
Grossmesser
WeaponFactory
<<returns>>
npcs
uses
Trader
<<returns>>
players
Player
uses
gameactions
standardactions
PurchaseAction
SellingAction

<u>**Requirement 2 - Traders and Runes**</u>

Requirement 2 focuses on Traders, a non-playable type of character that sells and purchases goods from the player. The UML diagram for Requirement 2 outlines the classes needed to implement the functionality of Trader in this assignment.

The first design choice made was to create a Trader class in a new package called “npcs” in the game package. Trader is a subclass of/inherits from the Actor abstract class. As Traders are not currently killable, this class is not a subclass of/does not inherit from from the Mortal abstract class of Req1.

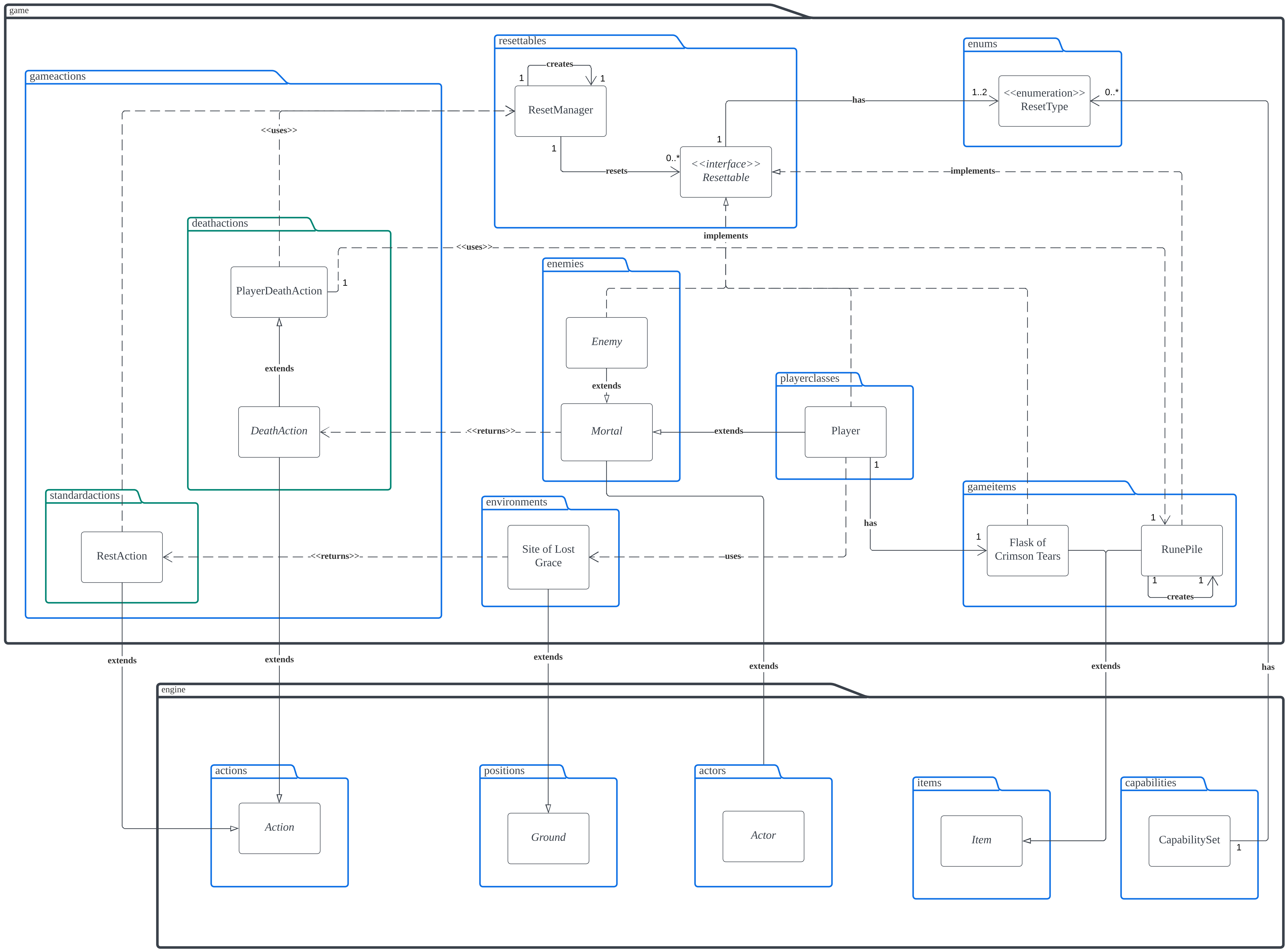
The second design choice made was to create PurchaseAction and SellingAction classes in the standardactions package. These classes extend the abstract class Action, and instances are returned by a Trader when the Player is next to a Trader and are used to purchase or sell items to the Trader.

The third design choice was to create a WeaponFactory class in the gameweapons package. The WeaponFactory class can return any weapon and will be used by the Trader to generate the weapons sold to a Player.

All three design choices adhere to the Single Responsibility Principle (SRP). All four added classes have only one responsibility; the Trader class is used to create Trader instances the player can interact with, the PurchaseAction and SellingAction instances are provided by the Trader to the player, so that the Player can choose to carry out the action, and the WeaponFactory class has the sole purpose of generating Weapons when a player makes a purchase.

The first and second design choices adhere to the Open-Closed Principle. All three classes in these choices extend abstract classes in order to add new features/methods to the already defined abstract classes. Using abstract classes also adheres to the Liskov Substitution Principle, since all methods in the Actor and Action abstract classes are present in the Trader and PurchaseAction/SellingAction subclasses respectively, so instances of these subclasses can be used when a instance of the parent class is called for.

However, none of these classes implement any extra functionality through interfaces at the moment. Therefore, the other SOLID design principles, interface segregation principle and dependency inversion principle, are not applicable to these design choices.


game
resettables
creates
1
1
ResetManager
1
resets
0..*
1
<<interface>>
Resettable
implements
has
enums
1..2
<<enumeration>>
ResetType
0..*
gameactions
<<uses>>
deathactions
PlayerDeathAction
1
<<uses>>
extends
DeathAction
enemies
Enemy
extends
Mortal
<<returns>>
playerclasses
Player
extends
1
has
implements
gameitems
1
Flask of
Crimson Tears
1
RunePile
1
1
creates
standardactions
RestAction
<<returns>>
environments
Site of Lost
Grace
uses
extends
extends
extends
extends
extends
has
engine
actions
Action
positions
Ground
actors
Actor
items
Item
capabilities
CapabilitySet
1

<u>Requirement 3 - Game Reset</u>

Our diagram for requirement 3 involves a few key components. There is a special ground The game reset occurs when the player dies or rests, which resets the player, enemies, and certain items. We also had to consider how to distinguish between the two types of resets - certain items such as runes would only be reset during the player's death.

PlayerDeathAction and Site of Lost Grace

These are the two main events that trigger a reset. The first is the Site of Lost Grace is simply a special ground (extends Ground) that returns a rest action that, if the player selects it, triggers a reset.

The second event is the player's death. The PlayerDeathAction is currently the only death action that resets the game, it makes sense to extend the DeathAction and specify the implementation details for the player which calls on a reset. This is in line with the Open Closed principle, since the DeathAction is closed for modification, but open to extension via its child classes.

Resettable Interface

The resettable interface is implemented by actors such as Player and Enemy, and by items such as Flask of Crimson Tears and Runes. This means that when an action is triggered that is supposed to reset the game, it will call on the ResetManager, so that all resettables that have been added to the ResetManager will use their reset method.

The Resettable interface only includes two things: a ResetType enum, and a reset method. This closely follows the interface segregation principle, as only classes that are affected by a reset will implement Resettable, while all other classes will not need to implement this.

ResetType Enumeration

The ResetType enum is used by anything that implements resettable. There are two types, rest and death, and while most actors/items have both, items like Runes only reset upon the player's death. This enum makes it easier to distinguish between which objects should reset.

ResetManager

This is the main class that uses reset actions. The ResetManager is a singleton class that stores everything that can be reset, so any Resettable will be added upon their creation. Then, when a reset is called, the ResetManager is used to call their respective reset methods, which are part of the Resettable interface.

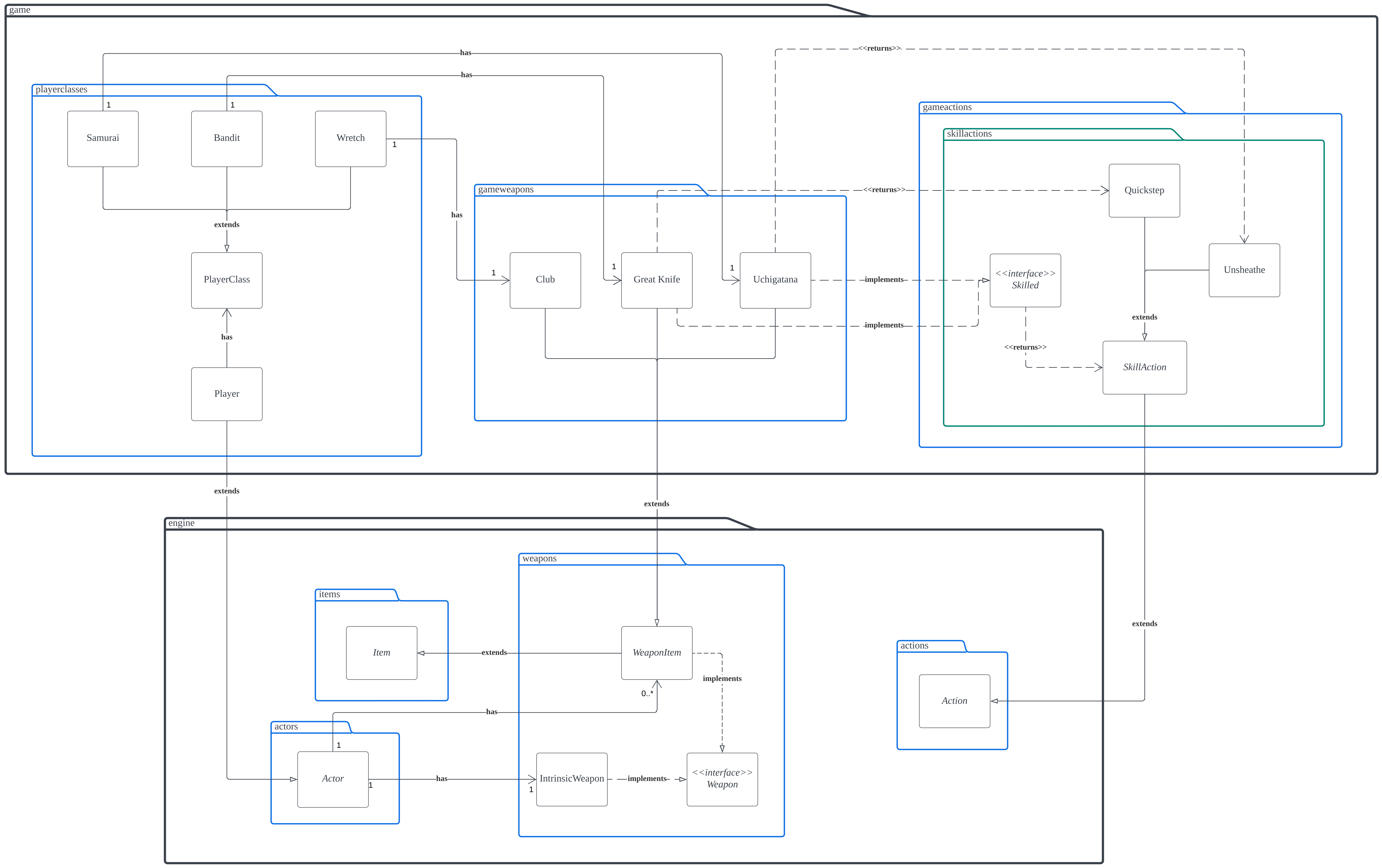
The ResetManager adheres to the single responsibility principle (SRP) as it is the sole class responsible for resetting everything in the World, and does not have any other responsibilities. This makes it more cohesive as there is only one reason to change this class. It also follows the open closed principle since it is closed to modification, but open to extension (by implementing more Resettables with new functionality, or Reset Enums). It also follows the dependency inversion principle, as it depends on the Resettable interface rather than concrete classes. This means that Resettable objects can derive from any class (Actors, Items, etc) and allows for easy extendability of the program (such as a new type of Resettable) without affecting the ResetManager class.

RunePile
We initially envisioned the concept of runes as a class that Player and Enemy have an instance of, so that it could be dropped upon an Enemy or Player's death. However, we realised that Enemies do not really drop runes in practice, but they "transfer" the amount if the player was the killer. Hence, we thought it was simpler to implement runes as an integer value innate to Player and Enemy. We still had to consider how the player would drop runes once he died, so that is how we conceived the idea of a RunePile.

The RunePile is an Item which is created upon the Player's death, and implements resettable. The RunePile is essentially just an object that stores the value of the player's runes before he dies, and if picked up, the player will regain the runes he dropped. If the player dies without picking it up, the old RunePile is discarded and a new one is created at the latest location of death.

The decision to make RunePile a singleton was due to the fact that only one can exist at any given time. This intuitively makes sense given our understanding of how the RunePile works - as soon as a reset is called that affects the player's runes, it will discard the current instance of RunePile on the ground (if any) and create a new one. Of course, if there are changes in the future where multiple sources can drop runes on the ground, we will revisit this design choice.

Pros and Cons
This design is highly extendable with regards to the core reset mechanic. The ResetManager works with any object that implements Resettable, meaning that it is very easy to create new objects that can be reset (or take an existing one and add functionality to allow it to be reset). Additionally, we can use the ResetType enumeration to include more events that trigger a reset in the future, such as beating the final boss, and only select certain Resettables to reset during this event.


game
playerclasses
Samurai
Bandit
Wretch
extends
PlayerClass
has
Player
has
has
has
1
1
1
gameweapons
Club
Great Knife
Uchigatana
1
1
1
<<returns>>
<<returns>>
implements
implements
gameactions
skillactions
Quickstep
Unsheathe
<<interface>> Skilled
extends
<<returns>>
SkillAction
extends
extends
extends
engine
weapons
items
Item
extends
WeaponItem
implements
0..*
has
actors
Actor
1
1
has
IntrinsicWeapon
1
implements
<<interface>> Weapon
actions
Action

Requirement 4 - Classes and weapons

Our diagram for requirement 4 is fairly simple. The requirement states that the player can select one of three combat classes for the game, each having their own weapon which can include a skill.

Combat Classes - Samurai, Bandit, and Wretch

Firstly, we created 3 combat classes, each being a child class of the abstract class playerClass. Initially, we thought it was more intuitive to make the Samurai, Bandit and so on extend the abstract Player class, but realised that this was unnecessary. The combat classes only determined the player's HP and weapon, and did not, for example, override any of the methods or add any new functionality compared to the original player class. We coupled the additional skills to the WeaponItems rather than the PlayerClasses, which makes sense as another Actor wielding the same weapon should be able to perform the same skill.

By using composition rather than inheritance, the code becomes easier to maintain and extend in the future. Creating more combat classes in the future would simply require determining the HP and weapon, rather than implementing every method from the player class, which is in line with DRY (don't repeat yourself). PlayerClass itself being an abstract class also follows the open and closed principle (OCP). Since the concrete combat classes extend PlayerClass, it is very easy to add new functionality or methods by extending the PlayerClass, while itself is abstract and thus closed for modification.

Weapons - Uchigatana, Great Knife, and Club

These are all weapons which correspond to the respective combat classes, which means they extend the WeaponItem class from the game engine. Among these, 2 weapons also allow the user to perform a special skill, so a new interface Skilled was created. The Skilled interface is not exclusive to weapons, it can be implemented by anything that essentially provides the user with a new SkillAction. The weapons themselves, which extend WeaponItem, follow the Liskov Substitution principle (LSP), as all child classes of WeaponItem can act as a WeaponItem or even just an Item. In the game, this would be relevant for actions such as dropping and picking up; Uchigatana and other WeaponItems can still behave like Items for this purpose.

Skills - QuickStep and Unsheathe

When WeaponItems implement Skilled, it means they provide the player with a specific skill. By using an interface we can easily distinguish between weapons with skills and weapons without them, and we will require them to return a SkillAction. This is another way we follow the Interface Segregation Principle as only the Weapons that return skills will implement this. Club, for example, does not provide any skills to the Player, thus it does not implement the interface and no unnecessary methods are needed.

Pros and Cons

A possible con of this design is that it may seem excessive in terms of the additional classes and interfaces created, which makes the design more complex and requires more time and effort to code initially. It may be difficult to explain how all the pieces work together to a new developer. However, the flexibility and extendibility of the game are greatly improved, and this design provides a solid foundation to add more features in the future, such as additional PlayerClasses with their respective WeaponItems, Skills, and so on. Additionally, we were able to greatly reduce coupling by making use of principles like the Open Closed principle. Overall, we found a good balance between adhering to SOLID principles to gain its benefits while still considering the specific needs and limitations that were described by the assignment.

Engine
positions
Location
Has
1
1
Ground
actors
Actor
actions
Action
weapons
<<Interface>> Weapon
Implements
Implements
WeaponItem
IntrinsicWeapon
game
<<Uses>>
Extends
Extends
Extends
Extends
enums
<<enumeration>> Status
<<enumeration>> Species
1..4
1
behaviours
<<Interface>> Behaviour
1..4
Has
<<Returns>>
Implements
Implements
Implements
Implements
SkillBehaviour
AttackBehaviour
FollowBehaviour
WanderBehaviour
Has
Has
Extends
Extends
Extends
enemies
Mortal
1
Extends
Enemy
1
1
EnemyFactory
<<Returns>>
<<Uses>>
Extends
Extends
Extends
Extends
PileOfBones
LoneWolf
GiantEnemy
SkeletalEnemy
Extends
Extends
Extends
GiantDog
GiantCrab
GiantCrayfish
Extends
Extends
SkeletalBandit
HeavySkeletalSwordsman
1
1
environments
Spawner
<<Uses>>
Extends
Extends
Extends
Graveyard
GustOfWind
PuddleOfWater
<<Returns>>
Implements
<<Returns>>
gameactions
deathactions
DeathAction
Extends
Extends
SkeletonDeathAction
StandardDeathAction
skillactions
<<Interface>> Skilled
<<Returns>>
SkillAction
Extends
Extends
Slam
SpinningAttack
standardactions
AttackAction
Implements
gameweapons
Sword
Extends
Extends
Grossmesser
Scimitar
1
1
<<Returns>>
traders
Trader
<<Uses>>
<<Returns>>
<<Returns>>
Has
Has

Requirement 5 - More Enemies

The diagram represents an object-oriented design for requirement 5 and displays the necessary classes and their relationships with each other in order to fulfil the needs of the overall system. The features requested by requirement 5 are additional to what has been defined in requirement 1, hence both diagrams will be similar and anything mentioned within the rationale of requirement 1 will also hold true here.

The addition of new enemies in this requirement, has given me reason to include two new abstract classes in the enemies package, the first of which being the GiantEnemy abstract class which is inherited by the GiantDog, GiantCrab and Giant Crayfish concrete classes, and the second, the SkeletalEnemy class which is inherited by the HeavySkeletalSwordsman and Skeletal Bandit classes. Both these classes extend the Enemy class and hence fulfil the Liskov substitution principle as their children can be substituted for any Actor object within the system. The reason I have chosen to create an additional layer of abstraction through these classes is to apply the DRY principle, HeavySkeletalSwordsman and Skeletal Bandit both use the SkeletalDeath action, this can be implemented within the SkeletalEnemy abstract class instead of being repeated in both concrete classes, the same is true of GiantDog, GiantCrab and Giant Crayfish which all use the Slam skill, repeating this code can be avoided by using the GiantEnemy abstract class which implements the Skilled interface and returns the Slam SkillAction. The open and closed principle is also observed here as the GiantEnemy and SkeletalEnemy classes are not open for modification but can be extended easily, the same was true for the Enemy class, as we have easily extended it here into two new abstract classes. The con of this approach is that we must implement a Species enum and add it to each Enemy's CapabilitySet individually to ensure Enemies of the same type/species will not attack each other. Instead of doing this, we could have, for example, had LoneWolf and GiantDog inherit from a Canine abstract class and make it so all Canines cannot attack each other. However, this results in more code repetition and a less extensible design should more giant enemies need to be added.

Another important decision to highlight is the addition of the Sword Abstract class which extends WeaponItem and implements Skilled. Since the Scimitar and Grossmesser both use the SpinningAttack skill but have different stats, it’s better to create a Sword abstract class which already returns the SpinningAttack SkillAction as an implementation of the Skilled interface, and allow both concrete classes to inherit from it. This adheres to the DRY principle and LSP principle as well as the open and closed principle as the Sword abstract class is not open to modification, however can be extended in the future by more WeaponItems that use the SpinningAttack skill. A con of this approach however is that Scimitar, Grossmesser and the SpinningAttack SkillAction are now tightly coupled, in the future, if Weapons are allowed to change their skills at runtime, this could become difficult to refactor.

Lastly, the requirement for different Enemies to be spawned on either the west or east side of the map has already been solved by the inclusion of the Spawner abstract class as checking whether a Spawner is on the east or west side of the map can be implemented trivially by the Spawner class and inherited by the Graveyard, GustOfWind and PuddleOfWater classes further adhering to DRY principles.