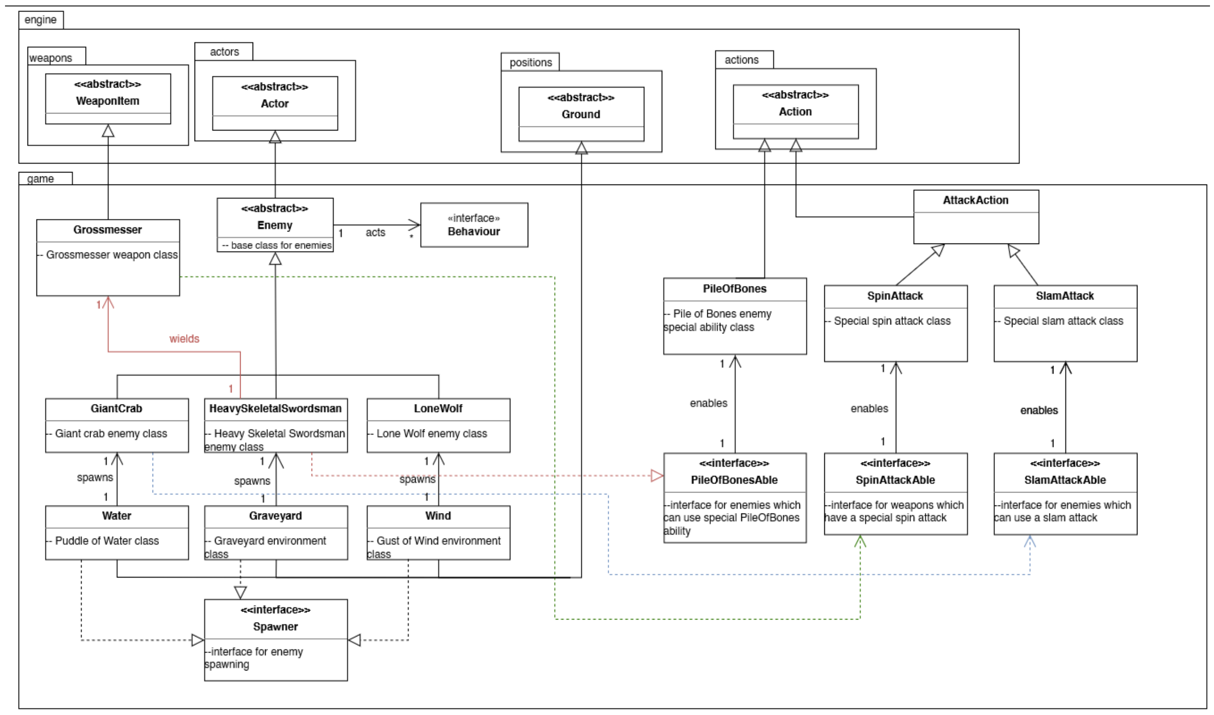
Requirement 1


engine
weapons
<<abstract>> WeaponItem
actors
<<abstract>> Actor
positions
<<abstract>> Ground
actions
<<abstract>> Action
game
Grossmesser
- Grossmesser weapon class
<<abstract>> Enemy
-- base class for enemies
1
acts
*
«interface» Behaviour
AttackAction
PileOfBones
- Pile of Bones enemy special ability class
SpinAttack
- Special spin attack class
SlamAttack
- Special slam attack class
1
wields
1
GiantCrab
- Giant crab enemy class
HeavySkeletalSwordsman
- Heavy Skeletal Swordsman enemy class
LoneWolf
- Lone Wolf enemy class
enables
<<interface>> PileOfBonesAble
-interface for enemies which can use special PileOfBones ability
<<interface>> SpinAttackAble
-interface for weapons which have a special spin attack
<<interface>> SlamAttackAble
-interface for enemies which can use a slam attack
spawns
Water
- Puddle of Water class
Graveyard
- Graveyard environment class
Wind
- Gust of Wind environment class
<<interface>> Spawner
-interface for enemy spawning

Design Rationale for Requirement 1

The diagram represents an object-oriented system for the various environments, enemies, and weapons in the game.

Each enemy type is represented in its own class, which all extend the Enemy abstract class. This is done because they share common attributes and methods which all enemies use. The Enemy abstract class itself extends the Actor abstract class, as it also shares common attributes and methods with other Actor subclasses in the game. The Giant Crab enemy's special slam attack is implemented as its own class called SlamAttack, which is a subclass of AttackAction, as it shares attributes and methods. The Heavy Skeletal Swordsman enemy's Pile of Bones ability is also implemented as its own PileOfBones class, which is a subclass of the Action abstract class. The ability of the HeavySkeletalSwordsman enemy to use the Pile of Bones ability is represented through it implementing the PileOfBonesAble interface.

Similarly, each environment is represented in its own class, which all extend the Ground abstract class. This is done because they share common attributes and methods which all grounds use. They also all implement the Spawner interface, for the enemy spawning functionality.

Finally, the Grossmesser weapon is represented in its own class, which extends the WeaponItem abstract class. This is done because it shares common attributes and methods which all weapons use. Its special attack is implemented as its own class called SpinAttack, which is a subclass of AttackAction, as it shares attributes and methods.The ability of the

Grossmesser to use the special spin attack is represented through it implementing the SpinAttackAble interface.

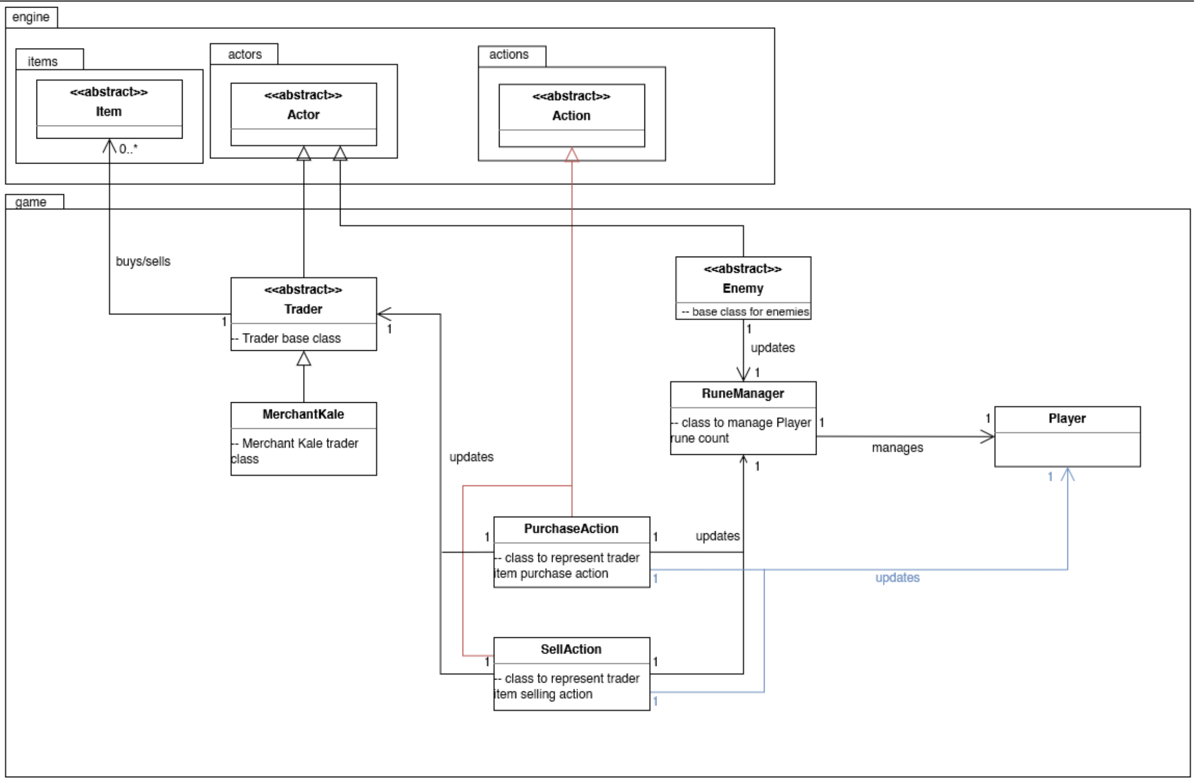
Adding new enemies, environments, and weapons by extending these abstract classes achieves abstraction, and therefore adheres to the Open-closed Principle. In other words, extending the Enemy or Ground classes to add new enemy or environments into the game is easy and can be done without modifying existing classes.

Creating and implementing interfaces adheres to the Dependency Inversion Principle, by creating an abstract interface, and not having concrete classes depend on each other.

It also follows the DRY principle of not repeating code, by reusing common attributes and methods through inheritance.

A disadvantage of implementing and extending the system with abstract classes is that the logic and code is split among several classes rather than in one place, which can make it difficult to understand. The size of code is also larger, and takes longer to code.

Requirement 2



Design Rationale for Requirement 2

The diagram represents an object-oriented system for the Traders and Runes in the game.

The player's runes are managed by a RuneManager class, which is responsible for increasing and decreasing the number of runes the player has, represented by a 1 to 1 association. When an enemy is killed by the player, the RuneManager class handles awarding runes to the player. This is represented by the association relationship between Enemy and RuneManager.

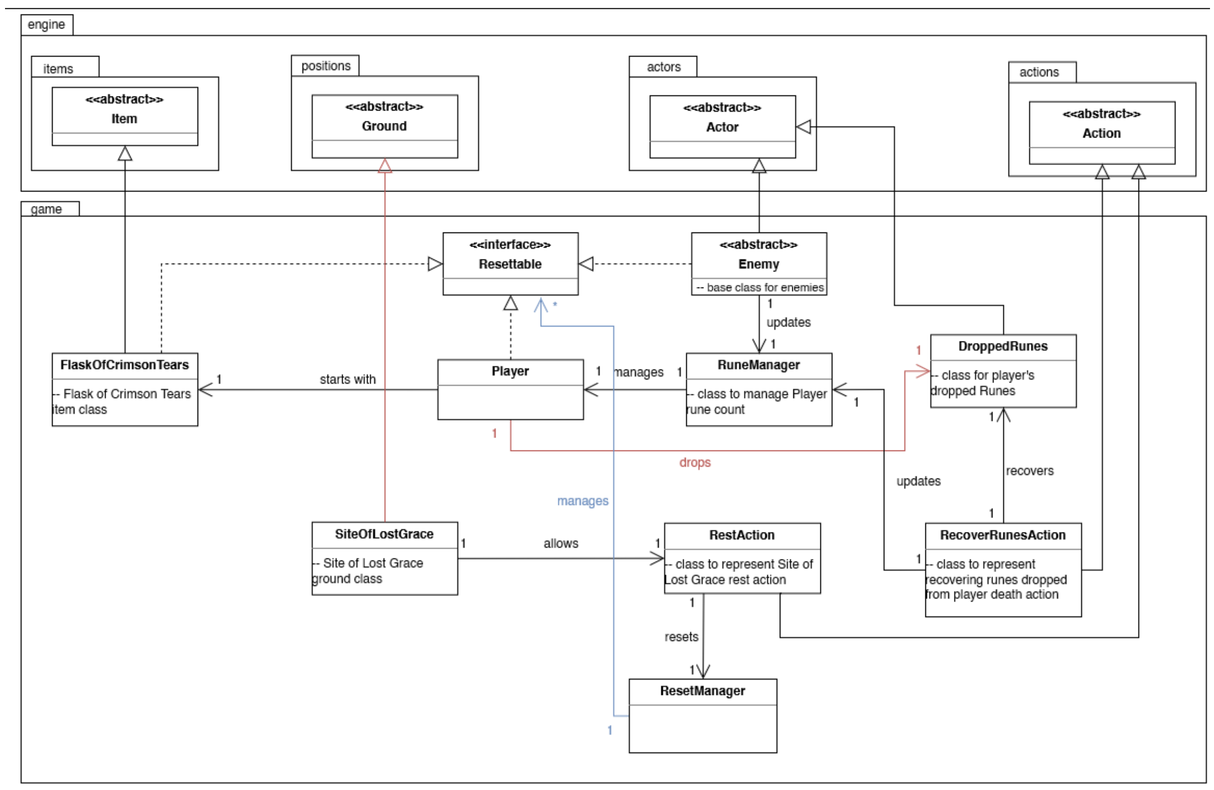
The trader Merchant Kale is implemented as its own class MerchantKale, which extends the Trader abstract class. Trader itself is a subclass of the Actor abstract class, as it shares common attributes and methods. Trader has an association of 1 to many with the Item abstract class, to represent the items it is willing to buy or sell.

The player can buy and sell items and weapons from Traders through the PurchaseAction and SellAction classes respectively, which extend the Action abstract class. Both actions interact with the RuneManager class to update the player's rune count after the purchase/sale.

The use of abstract classes to represent Traders adheres to the Open-closed Principle, allowing new Traders to be added without modifying existing code. It also adheres to the DRY principle by reusing code through inheritance in each Trader.

The use of a RuneManager class to update the player’s runes adheres to the Single Responsibility Principle, by handling the changes in rune count in its own class.

Requirement 3



Design Rationale for Requirement 3

The diagram represents an object-oriented system for the Flask of Crimson Tears item, Site of Lost Grace ground, Game Reset feature in the game.

The Flask of Crimson Tears item is represented by its own class, FlaskOfCrimsonTears, which extends the Item abstract class. Because the player starts with it, it has a 1 to 1 association relationship from Player to FlaskOfCrimsonTears.

The Site of Lost Grace ground is represented by its own class SiteOfLostGrace, which extends the Ground abstract class. It allows the player to rest on it, represented by the RestAction class, which extends the Action abstract class.

The Game Reset feature is implemented using the given ResetManager class and Resettable interface. Classes which are “resettable” implement the interface. This includes FlaskOfCrimsonTears, Player, and Enemy. The ResetManager handles resetting all the objects which implement the Resettable interface, represented by the 1 to many association between ResetManager and Resettable. It resets the game when the player performs the Rest action at a Site of Lost Grace ground, represented by the 1 to 1 association between ResetManager and RestAction.

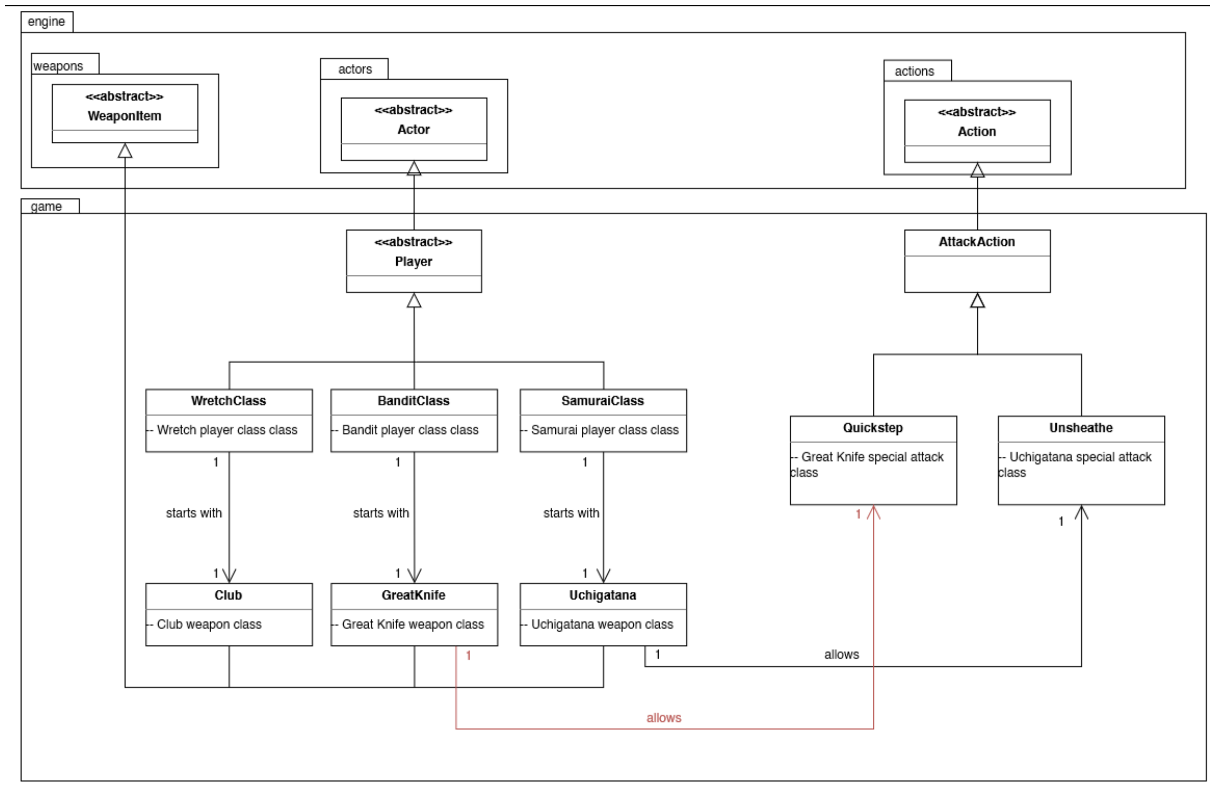
When the player dies, their runes are dropped, represented as a DroppedRunes class, which extends the Actor abstract class. The player can recover the Runes, an action represented by the RecoverRunesAction class, which extends the Action abstract class. RecoverRunesAction uses the RuneManager class to update the player’s rune count.

The use of abstract classes to represent the Flask of Crimson Tears item, Site of Lost Grace ground, Dropped Runes actor, and the two Rest and RecoverRunes actions adheres to the Open-closed Principle, by not modifying existing code. It also adheres to the DRY principle by reusing code through inheritance.

Adding functionality for classes to reset their state whenever the game resets through the use of the Resettable interface adheres to the Dependency Inversion principle, by not having concrete classes depend on each other, and using an abstract interface for all “resettable” classes.

A slight disadvantage to the approach of creating and implementing the interface is that more code has to be written. It can also cause issues if implemented poorly, forcing subclasses to implement unneeded methods, if the Interface Segregation Principle is not followed.

Requirement 4



Design Rationale for Requirement 4

The diagram represents an object-oriented system for player classes, the respective starting weapons, and their special skills.

The three player Classes(capitalised to reduce confusion), Wretch, Bandit, and Samurai are represented as classes of the same name. The Player class has been changed to an abstract class to allow for these Class classes to extend and implement.

The three associated weapons of each class, Club, Uchigatana, and Uchigatana are also represented as their own classes Club, GreatKnife and Uchigatana respectively. They all extend the WeaponItem abstract class, and have a 1 to 1 association from the Class to the weapon to represent that each Class starts with their respective weapon.

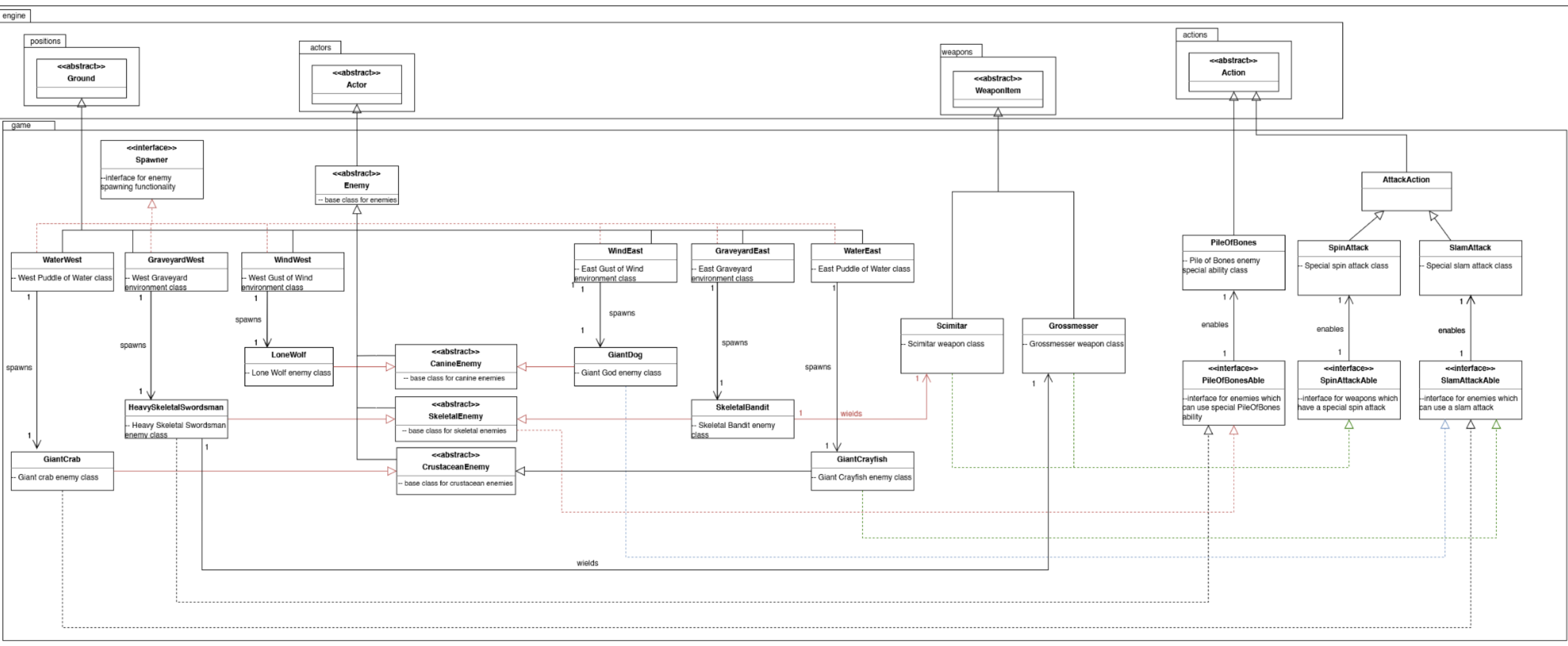
The Great Knife weapon's Quickstep skill is represented as its own class, Quickstep, which extends the AttackAction abstract class, as it is a special attack. The 1 to 1 association from the weapon to Quickstep shows that the Great Knife weapons allows its wielder to use Quickstep.

Similarly, the Uchigatana weapon's Unsheathe skill is represented as its own class, Unsheathe, which extends the AttackAction abstract class, as it is a special attack. The 1 to 1 association from the weapon to Unsheathe shows that the Uchigatana weapons allows its wielder to use Unsheathe.

The use of abstract classes to represent the player Classes and weapons adheres to the Open-closed Principle, by not modifying existing code. It allows for new Classes and weapons to be added in the future with minimal risk. It also adheres to the DRY principle by reusing code through inheritance.

A downside to this approach is that the code and logic for player Classes and weapons is spread across multiple classes rather than in one place, which can make it harder to understand. It also results in larger code.

Requirement 5


engine
positions
<<abstract>> Ground
actors
<<abstract>> Actor
weapons
<<abstract>> WeaponItem
actions
<<abstract>> Action
game
<<interface>> Spawner
-interface for enemy spawning functionality
<<abstract>> Enemy
-- base class for enemies
WaterWest
- West Puddle of Water class
GraveyardWest
- West Graveyard environment class
WindWest
- West Gust of Wind environment class
WindEast
- East Gust of Wind environment class
GraveyardEast
- East Graveyard environment class
WaterEast
- East Puddle of Water class
spawns
1
LoneWolf
- Lone Wolf enemy class
HeavySkeletalSwordsman
- Heavy Skeletal Swordsman enemy class
GiantCrab
- Giant crab enemy class
<<abstract>> CanineEnemy
-- base class for canine enemies
<<abstract>> SkeletalEnemy
-- base class for skeletal enemies
<<abstract>> CrustaceanEnemy
-- base class for crustacean enemies
GiantDog
- Giant God enemy class
SkeletalBandit
- Skeletal Bandit enemy class
GiantCrayfish
- Giant Crayfish enemy class
wields
Scimitar
- Scimitar weapon class
Grossmesser
- Grossmesser weapon class
AttackAction
PileOfBones
- Pile of Bones enemy special ability class
SpinAttack
- Special spin attack class
SlamAttack
- Special slam attack class
enables
<<interface>> PileOfBonesAble
-interface for enemies which can use special PileOfBones ability
<<interface>> SpinAttackAble
-interface for weapons which have a special spin attack
<<interface>> SlamAttackAble
-interface for enemies which can use a slam attack

Design Rationale for Requirement 5

The diagram represents an object-oriented system for enemies, weapons, and their related special skills in the game.

The three environments Puddle of Water, Graveyard, and Gust of Wind have been separated into West and East variants, each being their own class.

Each new enemy, Giant Dog, Skeletal Bandit, and Giant Crayfish have been implemented as the GiantDog, SkeletalBandit, and GiantCrayfish classes. They extend abstract classes of their respective base enemy type, CanineEnemy, SkeletalEnemy, and CrustaceanEnemy respectively, which themselves extend the Enemy abstract class.

The SkeletalEnemy abstract class implements the PileOfBonesAble interface, which enables the enemy to use the PileOfBones special ability. This allows all enemies which extend the SkeletalEnemy abstract class to use the ability.

The Giant Crab, Giant Crayfish, and Giant Dog enemies implement the SlamAttackAble interface, which allows them to use the slam special attack.

The Skeletal Bandit enemy wields the Scimitar weapon, which has been implemented as its own class Scimitar extending the WeaponItem abstract class. Scimitar and Grossmesser both implement the SpinAttackAble interface, allowing their wielders to use the spin special attack, implemented as the SpinAttack class, extending the AttackAction class.

The decision to implement abstract classes for enemy base types such as CanineEnemy was made keeping in mind that enemies do not attack other enemies of the same “type”. This also adheres to the Open-Closed Principle, by allowing new enemies of the same type to be created without modifying existing code. Subsequently, it adheres to the DRY principle, by reusing code through inheritance.

The decision to implement the ability to use these special attacks and abilities as interfaces was to allow enemies(classes) which were not necessarily similar to inherit their usage. For example, Giant Dog and Giant Crab both are able to use the slam special attack, but do not share the base enemy type (CanineEnemy vs CrustaceanEnemy). Therefore, the only way for both of them to inherit the slam attack functionality is through an interface (since multi-inheritance is not possible in Java). This adheres to the Dependency Inversion Principle, by not having concrete classes depend on each other, and using abstractions through an interface.

A slight disadvantage to the approach of creating and implementing the interfaces is that more code has to be written. They can also cause issues if implemented poorly, forcing subclasses to implement unneeded methods, if the Interface Segregation Principle is not followed. Additionally, the usage of abstraction in general means the code and logic is split across multiple classes rather than in one place, making it more difficult to understand.