REQ1 - ENVIRONMENTS AND ENEMIES

New environment related classes in our implementation:

- Spawner (abstract parent class)
- Graveyard
- Puddle of water
- Gust of wind

New enemy related classes in our implementation:

- Enemy (abstract parent class)
- Heavy Skeletal Swordsman
    - Grossmesser
- Lone Wolf
- Giant Crab
- Behaviour Interface
    - BasicAttackActionBehaviour
    - AOEAttackActionBehaviour

For the *environments*, each of the new environments (Graveyard, Puddle of water and gust of wind) are child classes of the abstract ‘Spawner’ class. As each environment is responsible for spawning an enemy with a random chance, it was concluded that they can inherit the same method from a parent class, and simply change the parameters. The implementation of this abstract parent class ‘Spawner’ is DIFFERENT from our original design from A1. This is in accordance with the DRY principle, as if they didn’t inherit this method from the parent class, they would each contain the same method and this would be repetitive. Similarly, this is also in accordance with the SRP principle, as the spawner class is responsible for the spawning within each environment, and each environment is solely responsible for which enemy they are supposed to spawn. Similarly, the spawner class and each environment is open for extension, but closed for modification (OCP), as further methods can still be added without affecting the implementation of the previous methods. The implementation of the spawner class also follows the Liskov substitution principle - if each enemy that was spawned by the spawner class were to be replaced with an instance of its superclass (Actor), the code will still be able to work as spawner does not inherently rely on any special attribute within the enemy class to function. Finally, this design also follows dependency inversion, as the environments all depend on the abstract ‘spawner’ class, instead of a low level class.

For the *enemies*, each of the new enemies (Heavy skeletal bandit, lone wolf, giant crab) are child classes of the abstract Enemy class, which itself extends the Actor abstract class. The abstract ‘Enemy’ class, in accordance with SRP, is solely responsible for any methods that ALL enemies within the game will share - such as having a name, having a display character, having hitpoints, implementing behaviours, etc. This is also in accordance with the DRY principle, as it would be tedious to have to establish methods in each new enemy child class. Therefore, this also follows OCP, as it is easy to extend the current code and add new enemies, however it is closed for modification. Similarly, the implementation of enemies follows the Liskov substitution principle - if each enemy was replaced with an instance of its superclass (Actor) in any of its implementations throughout the code, the code will not crash as the implementation of enemies does not inherently rely on the enemy child class to function. This also follows dependency inversion as all the enemies rely on abstraction as they extend the Enemy class, which in itself extends the abstract Actor class.

The attack behaviours that were created both inherit from the AttackAction class (and implement the behaviour interface) - all enemies have a basic attack action they all inherit from the BasicAttackAction class, which is in accordance with the DRY principle. Similarly, only some enemies will have the AOE attack - the enemies that do have this behaviour simply inherit from the AOEAttackActionBehaviour class, instead of it being coded in every time an enemy with the AOE behaviour is added, also in accordance with DRY. This also follows SRP, as the BasicAttackActionBehaviour and AOEAttackActionBehaviour are both solely responsible for their attack actions and nothing else. Similarly, all the weapons inherit from the abstract WeaponItem class, which itself extends the abstract Item class and implements the Weapon interface - in accordance with the dependency inversion principle. The creation of these separate behaviour classes was not explicitly outlined in our A1 submission, and over the course of A2, the behaviours were separated and this can be counted as a DIFFERENCE in our implementation from A1 to A2.

Pros
A significant benefit of this implementation is that new environments are able to be easily implemented by inheriting the abstract Spawner class. The spawner and current methods are also closed for modification, but open for extension (which is relevant for req 5). Similarly, new enemies are easily able to be created by inheriting from the Enemy abstract class.

One more positive benefit of this implementation is if new enemies are to be implemented, they can implement either BasicAttackAction, AOEAttackAction, both or neither due to these behaviours existing as classes.

Cons
A negative aspect of this implementation of environments is that the environments only exist as 1 displayChar - meaning that if an environment that is larger than 1 display char is to be implemented, it would have to be entirely created from scratch and cannot be implemented with the current code.

One more negative aspect of this implementation is that this also applies to enemies, as each enemy and weapon are only represented by 1 displayChar on the map - if a larger weapon or enemy is to be implemented in the future, it will have to be created from scratch.

REQ2 - TRADER AND RUNES

New classes in our implementation:

- Merchant Kale (Trader)
- Great Knife
- Uchigatana
- Club
- Runes
- Buyable Interface
- Sellable Interface

- SellAction
- BuyAction
- TradeAction

In accordance with the dependency inversion principle, all new weapons inherit from the abstract WeaponItem class, which itself extends the abstract Item class and implements the Weapon interface. In accordance with SRP, each child class of WeaponItem is solely responsible for dictating their displayChar, attack accuracy and attack damage.

As the WeaponItem was not able to be modified, the creation of a buyable and sellable interface was necessary for different subclasses of weapon to inherit methods to be bought and sold. In accordance with SRP, the buyable interface is solely responsible for the method giving some weapons their buy price, and the sellable interface is solely responsible for the method giving some weapons their sell price. This is also in accordance with interface segregation.

Merchant Kale is a new class that extends the abstract Actor class, in accordance with dependency inversion. He is responsible for iterating through the Player's inventory and returning options to sell weapons, and also has unlimited stock of the Uchigatana, Great Knife and Club. Merchant Kale follows the Liskov Substitution principle, as if necessary, he'd be able to sell items, which are the superclass of weapon items.

The Buy and Sell Action both extend the TradeAction class. The TradeAction class has the necessary methods for updating runes in accordance with a transaction, following SRP. The BuyAction class follows SRP by being solely responsible for everything to do with a player's purchase - that being removing the specific number of runes from a player's inventory, adding a new item to the player's inventory and returning the dialogue to console of the player's purchase. Similarly, the SellAction class follows SRP by being solely responsible for everything to do with a player selling a weapon - that being adding the specific number of runes to the player's inventory, removing the item to the player's inventory and returning the dialogue to console of the player selling the weapon. This also follows the interface segregation principle. The implementation of the Buy and Sell Actions (not the interfaces) was not included in our A1 submission - these actions were implemented to better fit SRP. Originally, Merchant Kale was going to be solely responsible for all actions related to transactions, including adding or removing items from the player's inventory - violating SRP. These two actions separate the methods and also follow interface segregation.

The Runes class extends the abstract Item class, in accordance with the dependency inversion principle. It also follows SRP as this class is solely responsible for establishing the number of runes the player starts the game with, and updating them after a transaction / finding runes / dying in the world and dropping them.

Pros

A benefit of this implementation is that new weapons can be created and easily be given a buy or sell price upon creation, following the principle of open for extension but closed for modification.

Another benefit of this implementation is that weapons can be implemented WITHOUT having to give them a buy or sell price - or neither. For example, the Grossmesser cannot be bought from Merchant Kale - however, if the player picks one up after defeating a Skeletal enemy, it can be sold. This is a result of the Buyable and Sellable interfaces, following the single responsibility principle.

Cons
A negative aspect of this implementation is that Merchant Kale is solely responsible for any trading that occurs within the game, and is the only instance of a 'Merchant'. If another merchant style character were to be added, Merchant Kale's class would have to be essentially copied as he is a child class of Actor, and not a child class of an abstract class called 'Merchant'. This would violate the dependency inversion principle and the DRY principle.

Another negative aspect of this implementation is that if a new currency aside from runes were to be implemented in the future, an entirely new currency system would have to be created - as the rune class does not inherit from an abstract 'currency' class, violating the dependency inversion principle.

REQ3 - GRACE AND GAME RESET

New/Extended classes in our implementation:

- Flask of Crimson Tears
- ConsumeHealAction
- Site of Lost Grace
- ResetManager
- Resettable Interface
- RestAction

The Flask of Crimson Tears extends the abstract Item class. It follows SRP by being solely responsible for dictating its number of uses, how much health it heals and if it can be used - meaning if its uses equals 0, removing using the flask from the allowable actions. The Site of Lost Grace extends the abstract Ground class, following the dependency inversion principle. It follows SRP by being solely responsible for representing the ground that the Site of Lost Grace is at. The RestAction class extends the abstract Action class, and is solely responsible for checking the ground that the player is at, giving the player the RestAction option in the console to rest and then calling upon the ResetManager to begin the game reset.

The ConsumeHealAction extends the abstract Action class, and is used when the player consumes the Flask of Crimson Tears. In accordance with SRP, the class is solely responsible for verifying if the flask can be used, healing the player by the flask's set amount and printing the player's healing to the console.

The Resettable Interface is implemented by EVERYTHING that can be reset - meaning that every child class of Enemy and every child class of Player is reset. However, enemies and the player are reset differently - enemies are simply despawned, whereas the Player's health and flask uses are reset to maximum values. These reset rules are implemented in the abstract parent class Enemy, and in the Player class. If these reset rules were implemented in the reset manager, then this would violate the single responsibility principle, as each class is responsible for how they are reset.

In accordance with SRP, the ResetManager is the class solely responsible for resetting all the enemies and the player upon resting at a Site of Lost Grace. This is also in accordance with interface segregation, as the resettable interface tells which classes if they can be reset, the classes themselves are responsible for how they are reset and the reset manager, when called, carries out the reset. This is also in accordance with the liskov substitution principle, as if an instance of the Actor class were to implement the resettable interface, when the reset manager is called, it would still be able to be reset.

Pros
A benefit of this implementation is that it is easy to create an item that needs to be reset - the resettable interface simply needs to be implemented. Another benefit of this implementation is that each item has unique reset conditions - such as the flask, which is reset back to 2 uses when the player rests at the Site of Lost Grace.

Another benefit of this implementation is that if a new enemy is to be added in the future that does NOT get reset when the player rests at a Site of Lost Grace, it can simply override the method from its parent class Enemy.

Cons
A negative aspect of this implementation is that the ConsumeHealAction takes the Actor holding the flask of crimson tears as a parameter - meaning that if another item that ‘heals’ the player is to be added in the future, this ConsumeHealAction class will have to be modified, violating OCP.

Another negative aspect of this implementation is that the reset only works in one specific way for all resettable objects - the enemies will always be despawned and the player’s health and flask is always set back to maximum. In the future, if a modified site of lost grace is implemented, that has a ‘partial reset’ function (IE only resets health and not the flask, or vice versa), a completely new reset method will have to be created from scratch.

REQ4 - CLASSES (COMBAT ARCHETYPES)
New/Extended classes in our implementation:
- Samurai
- Bandit
- Wretch
- QuickstepAction
- UnsheatheAction

All of these new combat classes extend the already existing Player class, in accordance with the DRY principle. All of these combat classes are solely responsible for dictating the player’s initial health and which weapon they begin the game with, in accordance with the single responsibility principle. This also follows the Liskov substitution principle, as an instance of the superclass (Player) is able to work in the same position as an instance of its subclass (Samurai, Bandit or Wretch). The samurai, bandit and wretch classes are all open for extension, but closed for modification, in accordance with OCP.

The QuickstepAction and UnsheatheAction both extend the AttackAction class, which in itself extends the abstract Action class. Both of these actions are unique to the Great Knife and Uchigatana respectively. In accordance with SRP, both of these action classes are solely responsible for dictating how much damage is dealt to the actor on the other side of the attack, as well as printing to console the results of the attack action. Additionally, the attributes of AttackAction were made to protected to allow UnsheatheAction and QuickstepAction to access the parents attributes.

Pros
A benefit of this implementation is that the Bandit, Samurai and Wretch classes can all be easily extended if new features are to be added in the future - such as the addition of multiple weapons.

Another benefit of this implementation is that more combat archetype classes can be implemented in the future with ease - such as a stealth class or an archer class, given that the necessary weapons exist also.

Cons
Currently, a significant negative of the implementation is of QuickstepAction, which takes the actor holding a Great knife as a parameter. This means that if another weapon that allows the user to do the QuickstepAction is to be added in the future, the QuickstepAction will have to be modified, violating OCP.

Another negative aspect of the implementation is of UnsheatheAction, which takes the actor holding a Uchigatana as a parameter. This means that if another weapon that allows the user to do the UnsheatheAction is to be added in the future, the UnsheatheAction will have to be modified, violating OCP.

REQ5 - MORE ENEMIES

New/Extended classes in our implementation:

- Skeleton (Abstract enemy class)
    - Skeletal Bandit
- GiantEnemy abstract class
    - Giant Dog
    - Giant Crayfish
- Graveyard
- Gust of Wind
- Puddle of Water

The 5th requirement in this assignment is to EXTEND the already existing environments and enemies from requirement 1. The Skeletal Bandit inherits from the abstract Skeleton class, which inherits from the abstract Enemy class, which inherits from the abstract Actor class. The other new enemies, all ‘Giant’ enemies, all share the same AOEAttackActionBehaviour, and therefore all the new Giant enemies inherit from an abstract ‘GiantEnemy’ class, which inherits from the abstract Enemy class, which inherits from the abstract Actor class. The GiantEnemy abstract class, in accordance with SRP, is solely responsible for giving all subsequent child classes the AOEAttackActionBehaviour, which is also in accordance with the DRY principle - without the implementation of the GiantEnemy abstract class, all the new Giant enemies would have the same few lines of code. This is in accordance with dependency inversion. The GiantEnemy class was not originally present in our A1 submission - this was because all enemies were to be extended from enemy. Upon further inspection while working on A2, it was discovered that many enemies shared the same AttackActions - and thus the GiantEnemy Abstract class was created to allow the child classes to inherit the method - DIFFERENT from our A1 submission.

A part of the requirement for these new enemies is spawning dependent on location; meaning that environments on the west side of the map spawn specific enemies, and environments on the east side of the map spawn other specific enemies. In accordance with OCP, the Spawner class from Req 1 was thus EXTENDED to include a method that calculates the map's location on the map and returns a boolean value for if the environment is west. The environments themselves are responsible for designating the enemies they are responsible for spawning depending on their location, in accordance with SRP. The graveyard, if west, spawns a Heavy Skeletal Swordsman, and if east, spawns a Skeletal Bandit. The puddle of water, if west, spawns a Giant Crab, and if east, spawns a Giant Crayfish. The gust of wind, if west, spawns a Lone Wolf; and if east, spawns a Giant Dog. All of these enemies can be replaced with an instance of their superclass and the code will still function, in accordance with the Liskov substitution principle. This requirement has been an example of how our code follows OCP, as our implemented code from req 1 was EXTENDED to accommodate these new requirements.

Pros
A benefit of this implementation is the way that the IsWest calculator was implemented. Instead of using hard coded values, the code uses the variable names. This means that if the map is to be enlarged or if a new map with a different size is implemented in the future, the IsWest calculator will still function as it takes parameters, not hard coded integers / magic numbers.

Another benefit of this implementation is that the Spawner class is open for further extension; for example if an implementation of different enemies spawning in the north and south were to be requested, the Spawner will be able to be extended to accommodate this request.

Cons
A negative aspect of this implementation is that if the environments are to be extended to pick from a range of enemies to spawn on either east or west, the code will have to be modified, violating OCP. Currently, each environment only supports spawning 1 enemy type varying on the east or west parameter.

Another negative aspect of this implementation is that if there is a future implementation of a 'Giant Skeleton' class, meaning that it has both the AOEAttackActionBehaviour and the PileOfBones ability, it will have to inherit from one class and the other ability will have to be manually added into the class instead of inheriting. This means that a new child class of

Enemy will have to be added that implements both the AOEAttackActionBehaviour and the PileOfBones ability for the hypothetical Giant Skeleton enemy variant to inherit from.

CONTRIBUTION LOG LINK:

https://docs.google.com/spreadsheets/d/1ESvnFQNk3XJ2QLDXeMLZOv2KC0u1jdanL7zptNXUyIY/edit#gid=1582995291