FIT2099 Assignment 1 Design Rationale

Authors: Emily Jap, Hayden Tran

Version 2.0

Date: 03/05/2023

NOTE: key modifications have been marked in red text

Requirement 1

Design Goals:

While following OOP design principles,

- Create the Environments, Graveyard, Gust of Wind and Puddle of Water that can spawn their respective enemies
- Create the Enemies (Heavy Skeletal Swordsman, Lone Wolf, Giant Crab) which can have: a unique ability, intrinsic weapon or carry a weapon. These things may have different actions/capabilities.
- Implement a way for enemies to be despawned when they are not following a player
- Create the Grossmesser weapon that can be dropped by the Heavy Skeletal Swordsman

Environments:

For the implementation of environments, we created a new abstract class SpawningGround that extends Ground, which has the child classes Graveyard, GustOfWind and PuddleOfWater. This is because the three new environments have similarities in that they all spawn enemies, so in line with the DRY principle, we have created the SpawningGround abstract class to implement the spawning functionality without having to repeat this functionality in individual concrete classes for each environment.

The SpawningGround class forms an association with the Actor class as it holds a HashMap of actors and their chance of spawning in that environment. This was to follow the ReD and dependency inversion principle, where instead of each environment forming dependencies with concrete enemy classes, instead there is an abstract layer (the Actor class) for which the SpawningGround class directly depends on. Additionally, we have created an abstract spawnActor factory method within SpawningGround. This forces each child class environment to be in charge of deciding where their respective enemies should spawn (east/west side), instead of having to implement this in SpawningGround. If we decide to change the way actors are spawned within their environments, (e.g. add a south side spawning functionality for Graveyard), we can change this within the Graveyard class and not have this affect other SpawningGrounds, making the code more extensible. The SpawningGround class also has a dependency relationship with RandomNumberGenerator so that we may implement a way for enemies to be randomly spawned/despawned at a location within their corresponding environment in the tick() method.

Enemies:

For our implementation of enemies, we have created an abstract class ‘Enemy’ which has abstract subclasses ‘Skeleton’, ‘Canine’ and ‘Crustacean’. Each new enemy class HeavySkeletalSwordsman, LoneWolf and GiantCrab extend from these classes, respectively. This is because each enemy shares characteristics such as follow behaviour, resettable. Further, each type of enemy also has shared characteristics, hence to follow the DRY principle we have created these abstract classes so that when more enemies are added in the future, we can have it extend from an existing type/create a new one.

Each enemy has an association relationship with the Behaviour interface as it has a list of behaviours as an attribute. This is to avoid each enemy having an association relationship with each behaviour it exhibits, so instead of each enemy directly relying on the implementation of each behaviour, it instead relies on the Behaviour interface as an abstraction layer, endorsing the dependency inversion principle. We have selected this principle as we are expecting actors to exhibit a considerable amount of behaviours, so we want to reduce the amount of dependencies where possible.

HeavySkeletalSwordsman carries a Grossmesser in its inventory and has a BecomePileOfBonesBehaviour that returns a BecomePileOfBones action when its health points are equal to or fall below zero. Inside BecomePileOfBonesBehaviour, HeavySkeletalSwordsman and its location is passed into BecomePileOfBonesAction as an association so we know what weapon to drop and where when PileOfBones is attacked. BecomePileOfBones spawns a PileOfBones at the same location as the HeavySkeletalSwordsman (using the Location class), which the player can then attack. An alternative design we were considering was to make PileOfBones a child class of Item. Ultimately we decided against this as we might want PileOfBones to be able to do certain actions in the future such as throw bones. Hence, to make the system more extensible, we decide to make PileOfBones extend Actor.

To implement the 10% chance of despawning at each turn, we have added DespawnBehaviour and DespawnAction so we can prioritise FollowBehaviour over DespawnBehaviour (as actors that are following the player cannot despawn). This design follows the open close principle as any actors that exhibit the same behaviour in the future can add this behaviour to their list of behaviours.

Weapons and Attacks:

For our implementation, the Grossmesser extends the WeaponItem abstract class as it can be dropped and is a weapon.

To handle the area attacks, we have AreaAttackBehaviour and AttackBehaviour for HeavySkeletalSwordsman and Giant Crab to decide when to perform AreaAttackAction and AttackActions with their Grossmesser and IntrinsicWeapon, respectively. As Lone Wolf only has an intrinsic weapon at this stage with no area attack capability, it only forms an dependency with IntrinsicWeapon and exhibits AttackBehaviour. At this stage we have assumed that HeavySkeletalSwordsman will

always have AreaAttackBehaviour as it is always carrying a Grossmesser (except for when it dies). An improvement to this design would be to couple AreaAttackBehaviour to the Grossmesser instead of the HeavySkeletalSwordsman, so when it drops the Grossmesser for whatever reason, the HeavySkeletalSwordsman will no longer have AreaAttackBehaviour or AttackBehaviour. This would therefore increase the extensibility of the code in case we want to add something like a stun action in the future.

Each AttackBehaviour/AreaAttackBehaviour forms an association with the Weapon interface, as a weapon is passed into the behaviour to create an AttackAction/AreaAttackAction if appropriate. AreaAttackBehaviour forms a dependency with the Utils class as it uses a getSurroundingActors() util method to find a list of potential targets in the surrounding area of the actor performing the area attack.

This design supports the open close principle as enemies that are added in the future are likely to also have AttackBehaviour and sometimes AreaAttackBehaviour, which can easily be added to the new enemy's list of behaviours without having to modify existing code. An alternative implementation could be for AttackBehaviour (and subsequently AttackAction) to handle both targeted and area attacks, however, this violates the single responsibility principle as targeted attacks and AOE attacks are handled in a separate manner and require slightly different implementations.

To prevent enemies of the same type from attacking each other (except in area attacks), the AttackType enum is used to indicate enemies of the same type. This allows our design to be extensible as, if we were to add another enemy that was a canine type, for example, we could add AttackType.CANNOT_ATTACK_CANINES as a capability for that new enemy, endorsing the open close principle. An alternative implementation would be to use instanceOf() to typecast enemies and check if they're the same type, however this would violate the open close principle as, if were to add another enemy that had the same type, we would have to add another instanceOf() check for this particular enemy, thereby needing to modify the existing code.

Pros	**Cons**
• It is easy to add new enemies due to the Enemy abstract class, AttackType enum and the different abstract enemy types we have set up • It is easy to add more types of spawning grounds and control where actors are spawned from these grounds • Actors/weapons that exhibit area attack can be easily integrated into the system by using AreaAttackAction/Behaviour	• We may want to add an enemy in the future that for whatever reason may not have a certain behaviour we have defined in the Enemy class such as WanderBehaviour, which may risk violating the Liskov substitution principle. To work around this, we can use the removeBehaviour method and keep the use of this method to a minimum, so we have decided to

• PileOfBones has access to functionalities related to actors (such as returning an action in playTurn) which can be added at a later time.	follow the DRY principle over the LSP.

Potential Improvements:

- We could introduce an abstract class for the environments to extend from as in this requirement as they all 'Spawn' something. This will allow us to repeat less code as we know that these locations will all have a similar spawn method
- For future introduction of enemies that don't despawn, we could add an interface for despawning, allowing us to easily create enemies that can and don't despawn.
- add death/despawn, handles when enemy dies

Requirement 2

Design Goals:

While following OOP design principles,

- Implement a currency called 'runes' that can be used by the player as a currency
- Implement a system which the player can buy and sell from Merchant Kale
- Allow enemies to drop a certain amount of runes

To avoid repetitions (DRY), a number of inheritance relationships exist in this system, namely:

- **MerchantK** extends the Actor class and forms a dependency with ActionList which holds a list of BuyActions for each item Merchant K sells. This was to follow the dependency inversion principle as, rather than forming multiple dependencies with many individual weapon items, Merchant K depends on the ActionList class as an abstraction layer.

  An alternative design we considered was creating a new NonPlayerCharacter abstract class for the Trader class to extend from, as when MerchantK extends Actor, it is exposed to some attributes and methods that it doesn't use such as hurt(), heal(). This design would be following the interface

segreation principle, as the NonPlayerCharacter abstract class would only implement attributes and methods from the Actor class that are relevant to NPCs such as MerchantK (ie. remove methods and attributes related to combat). We would also follow the liskov substitution principle as MerchantK would no longer be expected to fulfil combat-based expectations attached to the Actor class. Ultimately we decided against this design as it would be violating the DRY principle, as the NonPlayerCharacter class would copy most of the methods and attributes in the Actor class. We chose to follow the DRY principle over the interface segregation principle since we might want non player characters such as Merchant K to be attacked in the future, so if we extend from Actor the option is readily available. Additionally, there were only a small amount of methods we don't want non player characters to be exposed to at this time, so the benefit of following ISP isn't as high as following DRY.

- **Runes** extends the abstract Items class as it shares capabilities similar to other items in the game such as pick up, drop.
- **BuyAction** and **SellAction** extends the Action abstract class as they share similar methods with other actions in the game (eg. execute). They also form associations with the Buyable/Sellable interfaces as each instance of BuyAction/SellAction will require a parameter of type Buyable/Sellable in order to execute the trade.

  Alternatively, we could hold the player, merchant and items as attributes in the two actions, then implement a getter function in MerchantKale that will be able to access the items to buy/sell and their prices. However, in order to reduce dependencies we decided not to go with this implementation.
- **Club** and **Grossmesser** inherit from the WeaponItems abstract class as they all share methods and attributes included in the WeaponItems abstract class.

RuneManager class and RuneSource interface: we have created a RuneManager class to store all RuneSources in the game, and keep track of all rune owners (actors) and how many runes they currently have. The RuneSource interface is implemented by PileOfBones, GiantCrab and LoneWolf (along with other enemies in the game) as they transfer runes to the player upon death. The RandomNumberGenerator class is used to generate a random number of runes for an enemy to transfer upon death (within their specified range). The RuneManager class also has methods related to rune handling such as transferRunes, addRunes.

This design follows the single responsibility principle as, instead of each rune source being responsible for storing the number of runes they hold, and each having its methods related to runes, this responsibility is appointed to a separate class.

Weapons and Buyable/Sellable interfaces: Each tradeable weapon implements either buyable or sellable interfaces, or both if they can be both bought or sold. This follows the Interface Segregation Principle as weapons that can only be sold but not bought do not have to be exposed to methods related to buying. This design also supports the open close principle as any items that are added in the future can implement these interfaces to make the item tradeable without the need to modify existing code.

An alternative design we considered was using a Tradeable enumeration to mark weapons that can be bought/sold. However, this design risks violating the DRY principle as items that can be bought/sold share methods such as getBuyPrice/getSellPrice, so we thought it was better to bundle these methods in an abstraction layer by using interfaces.

Alternatively, buyable and sellable could be made into its own abstract classes, and each weapon could inherit from either buy/sell abstract classes. However, this implementation would not work as some weapons can be both bought and sold, but classes can only extend from one abstract class.

Pros	**Cons**
• Buyable/Sellable items that are added in the future can be easily implemented without having to modify existing code • Merchant K can become more interactive in the future (eg. be attacked by the player or enemies) making the design more extensible • More traders can be added in the future and inherit shared trader attributes by extending from the Trader class (DRY) • Any classes that require rune to be managed can use the RuneManager class.	• The RuneManager class is a singleton pattern which violates the dependency inversion principle as any class that uses the RuneManager depends directly on the RuneManager as a concrete class. We have decided to follow the single responsibility principle over the dependency inversion principle in this case as there are many classes that require rune management, and so to have each of these classes implement their own rune management methods would result in a lot of repetition. • Each sellable WeaponItem has an association with SellAction, if we add more sellable weapons in the future there will be a lot of associations with SellAction. We decided to make this tradeoff as the alternative would be to use downcasting to identify an instance of SellAction.

Requirement 3

Design Goals:

While following OOP design principles,

- Implement an item called ‘Flask of Crimson Tears’ that the player will always have, (cannot drop, always start with) that heals the player a certain amount of times
- Implement a game reset which despawns all enemies, reset player’s hit points and reset the amount of uses on the Flask of Crimson Tears
- Implement a unique ground which can reset the game if the player rests on it

FlaskOfCrimsonTears

Extends the abstract ‘Item’ class. This class has a dependency relationship with Player. The Player is dependent on this item, as the methods in this item are used in Player, and any changes to the item itself will affect the Player using the methods. For this item, a ConsumeAction has been added that will add health to the player whenever the action is called, and reduce the amount of usages. This class follows the SRP as it only handles one task. We previously had a separate action called ConsumeFlastOfCrimsonTearsAction, however this violated the Open Close principle as adding a new Consume action would result in us having to rewrite a bunch of the same code. Hence, we created a general Consume Action to rectify this violation and allow our code to be more extensible.

The First Step

The First Step is our first Site of Lost Grace and is a class which extends the abstract ‘Ground’ class. This class will possess the unique ability to reset the game and will have dependency relationships with the Player so that they can have access to the RestAction. This design achieves the SRP in which the ground is only responsible for providing an action to the player where possible. This design however, may be improved for extensibility by making an Abstract Class ‘Site Of Lost Grace,’ that represents all future sites of lost grace similar to the first step as stated in the requirements.

Runes

In the current design ‘Rune’ extends Item and acts as a placeholder for when the actor dies and holds the amount of runes the player drops. This item is set to be non portable so that the Drop/Pickup actions aren’t accessed and we can create our own Retrieve actions instead as per the requirements.

Another design we considered for the runes being dropped and holding runes was setting the ground the actor died to in DeathAction to a RuneFloor which was a RuneSource in RuneManager. However with this implementation we ran into issues

with downcasting, violating Liskov Substitution Principle as any RuneSource that is used in RuneManager should be able to be used like any RuneSource.

For the amount that actors/rune sources hold, this is all managed through a hashmap through our RuneManager. This class has all the operations such as creating instances of Rune Holders, adding/subtracting runes.

For our implementation, we have included in the DeathAction so that whenever the player dies and does not have the capability 'Rested' they will drop a new instance of a Rune item, that holds the amount of runes they held, at where they died.

Actions

RetrieveRuneAction and RetrieveAction

We have created an abstract Retrieve Action currently specific to Runes similar to the Pickup and DropItem methods provided in the engine class. This makes the game more easily extensible for any other items that may need to be retrieved as well (OCP). When standing over the Rune, the player will be given the RetrieveRuneAction through the use of the tick() method in Items.

RestAction

We have created a rest action that whenever created from The First Step and called by the Player, the capability RESTED will be given to the player, and the runReset() method from the ResetManager will be called, removing all enemies from the map, resetting Flask of Crimson Tears usages, player health and setting player location to The First Step. This class follows the SRP where it doesn't handle more responsibilities than expected of it (resetting instances and adding RESTED to player).

RestLocationManager

A manager is created so that whenever a player rests, this location is stored statically so that it can be accessed anywhere for when the player dies again. This prevents the RestAction from doing more than it needs to, keeping in line with the SRP.

Resettable and ResetManager

The actors (Player, Enemy) and items (Runes, Flask of Crimson Tears) all implement a Resettable interface. This interface serves to have a function that will be called whenever entities are reset and will only be implemented for those we wish to be resetted in our implementation (Interface Segregation Principle). This implementation will allow for us to have the same function but a slightly different

reset method for each entity as they require. Also, using the interface will allow for flexibility for any other entity to be added.

The game reset calls upon the reset manager which takes all the instances registered by the Resettable interface, then runs the runReset() method. The game can be reset on two occasions, whenever a player dies (Death Action), or when the player rests (Rest Action) on a Site of Lost Grace. However in the current requirement the player always respawns at The First Step regardless of resting or not.

To follow the reduce dependency principle, the abstract Enemy class has an association with the ResetManager. Currently all enemies are assumed to be resettable so we have implemented the same reset() method from the Resettable interface in the and registered every enemy instance as resettable in the Enemy class. However, for future extensions where some enemies may not be resettable, we may simply edit the Interface method to not include them as an instance so they do not need to be reset.

Pros	Cons
- Uses a Location Manager which is static and can store and return the location where we last rested at any time (SRP). In charge of one task - We have a Consumable interface and general ConsumeAction for any item we wish to consume, and easily add more items in the future that may be consumed (OCP) - Resettable interface is implemented in Enemy Abstract class so that all the classes that extend from it don't have to reimplement it and thus reduce repetition in code (DRY)	- The retrieve action has the exact same methods as PickUpAction, maybe extend RetrieveRuneAction from this instead violating the DRY principles - The DeathAction is cluttered with player death and enemy death, could make a separate DeathAction for enemies and player to make code more readable (SRP) - 'The First Step' doesn't extend from an abstract class 'Site of Lost Grace.' In the current implementation there is only one site, but if more were to be added, this abstraction would allow for easier extensibility (OCP)

Requirement 5

Design Goals:

While following OOP design principles,

- Splitting the map into a west site and east side so that they can spawn different enemies
- Create the new enemies, Skeletal Bandit, Giant Dog and Giant Crayfish
- Create the new weapon Scimitar, carried by the Skeletal Bandit

Spawning and Environments:

In this requirement enemies are spawned in their respective grounds like in requirement 1. The grounds all extended from a SpawningGround

In this requirement we will spawn enemies in their respective environments in a similar way to requirement 1, however we will need to implement a way to split the map so that enemies will spawn on different sides of it.

Enemies and Weapons:

To avoid repetitions (DRY), a number of inheritance relationships exist in this system, namely:

- **Enemies**: SkeletalBandit, GiantDog, GiantCrayfish are child classes of their respective Enemy class (Skeleton, Canine, Crustacean) as done for the enemies in req.1, following the DRY principle. They will all implement the RuneSource interface as they will have the capability to drop runes to the player upon death.

  As with the enemies in req.1, SkeletalBandit, GiantDog, GiantCrayfish all have a list of behaviours as an attribute, hence they form an association with the Behaviour interface (in line with DIP). As in requirement 1, the new enemies have FollowBehaviour and WanderBehaviour in their behaviour list. AttackBehaviour and AreaAttackBehaviour are also added to the list of behaviours as each new enemy uses either intrinsic weapons such as Head and Giant Pincer (which perform AOE and targeted attacks), or the Scimitar carried by the Skeletal Bandit, which performs targeted attacks and a special spinning AOE attack. These behaviours decide if an AttackAction or an AreaAttackAction should be performed by the enemy (ie. if there is a target within the enemy's exits, AttackBehaviour/AreaAttackBehaviour will initiate its corresponding attack). Each AttackBehaviour/AreaAttackBehaviour forms an association with the Weapon interface, as a weapon is passed into the behaviour to create an AttackAction/AreaAttackAction if appropriate. AreaAttackBehaviour has a dependency with the Utils class as it uses the static utils method getSurroundingActors() to get a list of potential targets

within range.

As mentioned in requirement 1, this design supports the open close principle as enemies that are added in the future are likely to also have AttackBehaviour and sometimes AreaAttackBehaviour, which can easily be added to the new enemy's list of behaviours without having to modify existing code.

SkeletalBandit will exhibit the same PileOfBonesBehaviour as HeavySkeletalSwordsman in the requirement 1 implementation mentioned above.

GiantDog and GiantCrayfish have a dependency on IntrinsicWeapon as they both implement the method getIntrinsicWeapon() which returns their unique intrinsic weapon. Any skills that are acquired from the intrinsic weapon, such as slam, is accounted for by implementing a getSkill() method in the appropriate actor's class, which returns an Action such as AreaAttackAction for that skill. Since each enemy is now in charge of the special skill of their intrinsic weapon, this doesn't follow the single responsibility principle, so we tried thinking of alternative designs. We had thought of creating a separate class for each intrinsic weapon that would each implement the Weapon interface and form an association relationship with their respective actor. We decided against this, however, since if we were to add more enemies to the game with their own intrinsic weapons, our design could become repetitive and bulky as we'd add a new class for each enemy's intrinsic weapon. Additionally, it would make the existing IntrinsicWeapon class redundant. Hence, we decided to make the tradeoff between SRP and DRY in order to integrate with the engine code.

To prevent enemies of the same type from attacking each other (except in area attacks), the AttackType enum is used to indicate enemies of the same type. This allows our design to be extensible as, if we were to add another enemy that was a canine type, for example, we could add simply extend that class from the Canine class which already instantiates it with Status.CANNOT_ATTACK_CANINE reducing the need for repetition (DRY). An alternative implementation would be to use instanceOf() to typecast enemies and check if they're the same type, however this would violate the open close principle as, if were to add another enemy that had the same type, we would have to add another instanceOf() check for this particular enemy, thereby needing to modify the existing code.

- **Scimitar:** Inherits WeaponItem and similar to the weapons in req.2, uses the Tradeable enum to add Tradeable.SELLABLE and Tradeable.BUYABLE as capabilities, in line with the open close principle as mentioned in req.2.

As done with the grossmesser in requirement 1, Scimitar will have the capability PERFORM_AREA_ATTACK, giving the player the option to perform an area attack when possible.

Pros	Cons
- Uses subclasses for similar enemy types to extend from (e.g Canine for Lone Wolf and Giant Dog) which will have the capabilities so they don't attack each other as well (DRY) - A method getSurroundingActors is added in Utils which can be accessed for our AreaAttacks, this reduces clutter in our code where we just want to implement what it does, not any other additional methods (SRP)	- Enemies are in charge of their own special skills in intrinsic weapon violating SRP
