FIT2099 Assignment 1 Design Rationale

Authors: Cecilia Ho, Emily Jap, Hayden Tran

Date: 17/04/2023

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 have extended Graveyard, Gust of Wind and Puddle of Water from the abstract Ground class as each environment has common attributes and methods described in the abstract class (DRY).

Each environment also has a dependency relationship with their respective enemy that they spawn/despawn. This is due to the fact that each environment passes their respective enemies that they spawn/despawn as dependencies into their tick() method. Each environment also has have 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.

Enemies:

For our implementation of enemies, we originally thought of creating an abstract class called ‘Enemy’ which extends the ‘Actor’ class from the engine. This design was to be done to follow the DRY principle, as we knew that all enemies will share similar behaviours such as AttackBehaviour, FollowBehaviour and WanderBehaviour. While we agreed that this was true, we also realised that we would have a multi-level inheritance relationship, which can make maintenance more difficult as these three levels of inheritance are tightly coupled together (a bug/change in higher level classes may make it difficult to debug). Additionally, we may want to add an enemy in the future that for whatever reason may not have a certain behaviour such as WanderBehaviour, which may risk violating the Liskov substitution principle. In the end, although it may be at the expense of violating the DRY principle, we decided to have each enemy inherit directly from the Actor abstract class to allow for more flexibility in how we define enemies in the future.

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.

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. BecomePileOfBones spawns a PileOfBones item at the same location as the HeavySkeletalSwordsman (using the Location class), which the player can then break within three turns due to the Breakable interface (implemented by PileOfBones) and BreakAction.

An alternative design we were considering was to make PileOfBones a child class of Actor, as it can be hit by the player like any other enemies in the game. Ultimately we decided against this as, if PileOfBones was an actor, it would have the core functionalities of an Actor disabled, such as performing actions. This would therefore risk violating the Liskov substitution principle, as PileOfBones would be an actor that can't 'act' or have some sort of agency. Furthermore, an advantage of inheriting from the Item class is having access to its tick() method, which can allow PileOfBones to count the three turns before it is revived.

Weapons and Attacks:

For our implementation, the Grossmesser extends the WeaponItem abstract class as it can be dropped and is a weapon. WeaponItem implements the Weapon interface which has a getSkill() method, allowing weapons such as Grossmesser to return its special spinning skill (an AreaAttackAction).

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.

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.

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 NonPlayerCharacter abstract class and forms a dependency with the BuyAction and SellAction as these are added to Merchant K's list of allowable actions. Merchant K also forms an association with the WeaponItem class as MerchantK holds a list of WeaponItems that can be bought from them. This was to follow the dependency inversion principle as, rather than forming multiple dependencies with many individual weapon items, Merchant K is only dependent on the WeaponItems class as an abstraction layer. To improve extensibility, we could have Merchant K hold a list of Items instead so that they could sell WeaponItems as well as other items such as potions, books, etc that may be added at a later time. However, for now we have assumed that MerchantK only sells WeaponItems, but such

a change would not be too difficult to implement.

Although it would be possible to inherit from the Actor abstract class as MerchantK shares similar methods and attributes to other actors in the game (eg. allowable actions, name, display character), they also don't use many combat-related methods that are part of the Actor abstract class such as adding weapon to inventory. Hence, creating a separate abstract class for Merchant K to inherit follows the interface segregation 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.

Additionally, if we want to add more non-playable characters in the future, such as a blacksmith, it could inherit from the NonPlayerCharacter class and add any unique attributes/methods specific to the new character, making the system extensible and endorsing the open close principle.

- **Runes** extends the abstract Items class as it shares capabilities similar to other items in the game such as pick up, drop. It would also have the ability to be added to an actor's inventory just like other items in the game. The RandomNumberGenerator class is used to generate a random number of runes for an enemy to carry (within their specified range).

- **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 Items class as each instance of BuyAction/SellAction will require a parameter of type Items 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.

- **Uchigatana**, **GreatKnife**, **Club** and **Grossmesser** inherit from the WeaponItems abstract class as they all share methods and attributes included in the WeaponItems abstract class.

Weapons and Tradeable enum: Each tradeable weapon have a dependency on the Tradeable enumeration as this enumeration is added to the list of capabilities for each weapon. The tradeable enumeration indicates whether a weapon can be bought, sold, or both. This design supports the OCP as any items that are added in the future can use this enumeration to make the item tradeable without the need to modify existing code.

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. Alternatively, another way to achieve abstraction would be to use interfaces, for which a class can implement multiple if they need to, or just implement one. 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. However, we have seen in that game that an actor inherits the capabilities of the items they pick up. Hence, if the player picks up an item that is sellable, they now have the capability to sell that item. We wanted to take advantage of this built in capability, and so we decided to choose enumeration over interface.

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

Resettable

The actors (Player, Enemy) and items (Runes, Flask of Crimson Tears) in this UML diagram all implement a Resettable interface. This interface serves to have a function that will be called whenever entities are reset (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.

Alternatively, we could have each resettable entity be reset by the ResetManager. However, this goes against the Single Responsibility Principle, in which each class should have one responsibility and Open-Close Principle, where we should not tamper with the existing code given to us.

In the current implementation, the game can reset on two occasions, whenever a player dies, or when the player rests on the Site of Lost Grace.

SiteOfLostGrace

The Site Of Lost Grace extends the abstract ‘Ground’ class. This class will possess the unique ability to reset the game and will have dependency relationships with the ResetManager. This will also have a dependency relationship with Player, in which the Site is dependent on the player to make changes.

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. To implement this we were thinking of making a ConsumeAction which the player will reduces the number of potions and increases players’ HP.

Requirement 4

Design Goals:

- Create the player called Tarnished
- Allow the player to choose a combat archetype (Samurai, Bandit or Wretch) to determine the weapon they start with and HP before the game starts
- Create the different weapons (Uchigatana, Great Knife, Club) with their special skills as required

Player: The player extends the Actor class from the code given to us as they possess the common attributes of other actors. This is done to avoid repetition of code and reimplementing certain attributes and methods (DRY). For future extension or if we need to change things to be player specific, we may override the existing methods such as the OnDeath method being different for enemies and players.

Combat archetypes (classes): We have created an abstract class Archetype that the 3 required archetypes extend from as they have the common methods of setting the weapon and HP of the player. This follows the DRY principles as we prevent repetition of code. The Player has an association with the Archetype abstract class so that we can avoid unneeded dependencies on every single archetype, following the Dependency Inversion Principle.

Weapons: Similar to requirement 1, we have extended the weapons from the abstract class ‘WeaponItem’ as they all have similar methods and attributes. This is done to avoid unnecessary repetition and not have to rewrite code (DRY).

To implement the special skills, we can overwrite the getSkill() method of the parent class these weapons extend from. For the skills in the current requirements, we will create different classes for different capabilities.

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 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. As done in requirement 1, we have decided that each environment class (Graveyard, GustOfWind, PuddleOfWater) uses the Location class in each tick() method to spawn/despawn enemies. Within each tick() method, we would check the x coordinate of the location to determine if it can be spawned in the correct half of the map.

For another implementation, we may use an interface to split the map and prevent the environments from managing too much at once (by doing so prevent the violation of the SRP). However, we have decided against the implementation of this solution because this implementation is not as clear as the agreed upon implementation we have for this requirement.

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 the Actor class for reasons similar to the enemies in req.1 – They also have a depency with Runes which they carry in their inventories, which is dropped upon death by the player.

  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.

Giant dog and giant crayfish 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 Status 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 Status.IS_CANINE as a capability for that new enemy. 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.

  Scimitar also has dependencies with AttackAction and AreaAttackAction has these are added to their list of allowable actions, which grants the holder the ability to perform those actions.
