FIT2099 Assignment 3 Design Rationale

Authors: Emily Jap, Hayden Tran

Date: 21/05/2023

Note: Changes from assignment 2 are written in red text.

Requirement 1

Design Goals:

While following OOP design principles

- Create new grounds Cliff and Golden Fog Door
- Implement the new maps and the actions associated with these newly created grounds
- Edit the Sites of Lost Grace

Environments:

For the new grounds in this requirement, we have created a Cliff and GoldenFogDoor class which both extend the abstract Ground class to follow the DRY principle as these both will need to implement similar methods as any other Ground would.

The cliff class' functionality works by using the tick() method and checking if that location holds an actor. Then we use the location.getActor().hurt method for a large number that the actor's health bar will never be greater than, thus killing them and returning a new DeathAction at that location.

To create the teleport implementation, the GoldenFogDoor ground will have parameters for: the teleport location and the GameMap of whichever map we wish to set our teleport location to. When any ground of this type is created, the parameters must be filled out to match a location on another map. To teleport, we have implemented a TravelAction which extends from Action and will be given to the Actor whenever they are within 1 block of the Door. This takes in all the parameters from the Golden Fog Door, and makes use of the map.moveActor method to move the actor. These classes are implemented with the SRP principle in mind, where all classes only hold a single job, making our design clear and easy to implement.

For future extensibility of Sites Of Lost Grace, we have refactored our TheFirstStep class to make a SiteOfLostGrace abstract class in which all different sites may extend from, this was done to follow the DRY principles as all these sites have mostly the same methods and body of code, thus reducing repetition in code. This will also follow the Open-Close principle as if any further Sites Of Lost Grace are created, we may easily extend our code to add it by simply extending from the abstract base class.

Pros	Cons
- All new grounds extend the Ground abstract class to retrieve the same attributes and methods (DRY) - Cliff makes use of the existing game engine files to damage actor at a certain point and our DeathAction - SiteOfLostGrace abstract refactoring saves lots of unneeded repetition in code (DRY)	- Currently the locations of the teleporters are hard coded in the main application file, we currently are unsure if there is a better alternative to this, but this implementation is clear and easy to implement.

Potential Improvements:

- Find a way to implement teleporters without the need to hardcode the locations of both teleporters each time we make a new one

---

Requirement 2

Design Goals:

While following OOP design principles,

- Create new grounds Cage and Barrack
- Create new enemies Dog and Godrick Soldier

Enemies:

To add the new enemy Dog, we created a new Dog class that extends the Enemy class. This was to follow the DRY principle as Dog shares behaviours and capabilities of the Enemy class (DespawnBehaviour, FollowBehaviour, resettable).

We deleted the Canine abstract class as the initial design we considered was for Dog to extend the Canine class and remove the inherited AttackType.CANNOT_ATTACK_CANINES capability (as Dogs can attack LoneWolves and GiantsDogs). This would be breaching the liskov substitution principle as it is expected for all child classes of Canine to have the AttackType.CANNOT_ATTACK_CANINES capability. A downside to this design would be reduced extensibility and violation of DRY, as if we were to have more shared behaviours between all canines in the future, we would have to implement this in all canine-type enemy classes, violating DRY. Since there are no shared behaviours between canines in the current version of the game, we decided to make the tradeoff to avoid violating the liskov substitution principle.

To add the new enemy Godrick Soldier, we created a new abstract Soldier class (which inherits Enemy) and its child class GodrickSoldier, as GodrickSoldier also have shared characteristics of Enemies. Although the Soldier class doesn't currently have additional functionality, we still added it in case there are shared behaviours between soldiers in the future, following the DRY principle. To replace the heavy crossbow, we have added a Club weapon to the weapon inventory of GodrickSoldier.

To improve the design of assignment 2, we removed the targets attribute of AreaAttackAction so that the responsibility of finding surrounding targets is placed upon the AreaAttackAction class instead of some other class that should not be responsible for for finding targets (eg. Enemy class), in line with the single responsibility principle. We also follow DRY since we only need to find surrounding targets once in the AreaAttackAction class instead of everytime we call AreaAttackAction.

Another minor change was the extraction of allowPlayerAttack and allowAreaAttack methods in the Enemy class. This was to make the block of code in allowableActions more readable, and since these methods are protected, child classes of enemy can modify their allowableActions method while calling the allowPlayerAttack and allowAreaAttack methods to avoid copying and pasting the code block from the Enemy class.

Environments:

To implement the new Barrack and Cage environments, we created the Barrack and Cage classes which each extend the SpawningGround abstract class. This was to follow the DRY principle as Barrack and Cage both share SpawningGround characteristics and attributes such as spawning actors, display characters, etc.

To improve the design of assignment 2, we applied an abstract factory pattern to improve the way that enemies are spawned from their respective environments. We created an EnemyFactory interface and EastFactory and WestFactory classes which implement the EnemyFactory interface. The EnemyFactory interface determines what type of enemies are spawned on each side of the map eg. Canines, Skeletons, etc. The EastFactory and WestFactory classes handle the spawning logic of specific enemies (unique to the side of the map). For example, the spawning logic of HeavySkeletalSwordsman is handled in WestFactory as they are spawned on the west side of the map.

This design is advantageous as each child class of SpawningGround no longer need to know the logic behind how each enemy is spawned, as this responsibility has been extracted to the east and west factories, thereby following the single responsibility principle. If we decide to change the spawning logic of an enemy in the future, we can do this inside the factory classes instead of modifying each concrete spawning ground. Additionally, this design supports the open closed principle as, if we were to add more spawning grounds in the future, it is fairly straightforward to choose where and which enemies to spawn by using the east and west factories without having to modify existing code.

Pros	Cons
● Shared characteristics of Soldiers and more soldier type enemies can be easily added by extending the Soldier abstract class ● The map can be split into smaller regions such as south east, south west, etc by adding more enemy factories ● More spawning and damage grounds can be easily added in the future by extending from their respective abstract classes	● The new abstract factories can result in some repetition for enemies that are spawned on both east and west sides of the map, although we have made this tradeoff as it may be possible that we want to change which enemies spawn on each side when more enemies of the same type are added in the future. ● As more enemy types are introduced in the game, we have to modify the EnemyFactory interface to spawn that new enemy type. This tradeoff can be justified by the assumption that adding new enemy types will not happen as often as reusing the existing enemy types to spawn them in new environments when the system becomes much larger.

Potential Improvements:

- Create a spawn factory for enemies to spawn anywhere on the whole map, instead of only east or west

---

Requirement 3

Design Goals:

While following OOP design principles:

- Create a Golden Rune item that will have a consume method to generate a random amount of runes
- A new trader which can accept the Remembrance of Grafted with an exchange action
- New weapons for the functionality of the exchange action

Items:

We have created the two weapons Axe of Godrick and Grafted Dragon that can be received through Trader Enia, these weapons both extend the WeaponItem class as they will share the methods and attributes from this class. To exchange these items, we also created an Item called the Remembrance of Grafted. This item will implement the Sellable and Exchangeable interfaces.

We have created an Exchangeable interface which is currently only implemented by the Remembrance of Grafted. Our Exchangeable interface has some similar methods to the Sellable interface such as getAction and getItem, however our Exchangeable items don't require the getsellPrice() method, so we separate these two interfaces. We could have implemented everything our exchange actions do through the Sellable interfaces method, but decided against this as it would violate the Interface Segregation principle where we want interfaces to be singular purpose and also not force certain classes to implement unnecessary methods. This interface will be used in conjunction with the ExchangeAction where we will take the Exchangeable item and an item to receive as a parameter, then add either an AxeOfGodrick or GraftedDragon into the players inventory depending on the input parameter.

For the Golden Rune, we have extended it from the Item abstract class, to follow DRY principles due to having the same attributes and methods. This, as per the specifications, will have a consume method, so the Consumable interface will be implemented in this class. Through the tick() method, we will check the amount of charges of the item, and if larger than 0, the ConsumeRuneAction() will be added to the player inventory. We were considering refactoring our current ConsumeAction() to an abstract class that can cover all types of consumption, such as giving special effects, however we thought that implementing this would be too complicated, making the class cover too many areas and thus violate the Single Responsibility Principle. Although this item is also ‘consumed’ the methods will be different to our previous consume, which was adding health to the player, while ConsumeRuneAction will edit the amount of runes the player holds. Thus we created a new action that will overwrite the execute method of Action to add the runes to the player inventory, and remove the item.

Actor:

We have updated our design by making a Trader class that all current and future traders will extend from. This will follow the Open-Close principle as we will easily be able to add more traders in the future with similar methods by extending it from the Trader class.

FingerReaderEnia extends the Trader class and will not have a list of BuyActions as it does not have any item to sell like MerchantK. When instantiating this trader, we will give it the Status.ACCEPT_GODRICK_DROP capability so that only traders holding this capability will be able to accept the Remembrance of Grafted to exchange items. With this we have also added a new slightly modified isExchangeTraderNearby() method in the utils class that checks if a trader with the capability is nearby. This static method is now used in the tick() method of the Remembrance of Grafted item to add the necessary sell and exchange actions if the correct trader is nearby or just a sell action if the trader doesn’t hold the capability.

Pros	Cons
- Made a trader class which represents the base class for all future Traders and easy to extend (DRY, OCP) - Create an interface and method that will be specific for items that will be exchanged, instead of adding to another one with some similar methods such as sellable (Interface Segregation Principle) - Created a new type of consume Action that is specific for consumption of GoldenRune (SRP)	- ExchangeAction is currently hard coded to the two specific items and is not very extensible (violating Open-Close principle). To fix this we could make the action abstract and can have multiple classes extending from it having their own specific exchanges and methods, but for this current implementation is suitable as it is simple - We found difficulties in implementing a method that can check whether the trader has a capability or not, so we had to create 2 Utils methods that are practically identical, besides checking the capability, violating the DRY principle. To fix this must find a way to check whether the trader from the Utils initial method has the capability or not

Potential Improvements:

- Create a new method so that we don't have to run a list of exits twice when accessing traders that have the capability Status.ACCEPT_GODRICK_DROP

Requirement 5

Design Goals:

While following OOP design principles,

- Create a new Lesser Dragon enemy that can breathe fire within 2 blocks away from the dragon
- Create a new fire ground from which enemies take damage
- Allow the lesser dragon to attack enemies within 2 blocks away from the actor

Enemy:

To add the Lesser Dragon enemy, we created a new abstract Dragon class that inherits the existing Enemy class. We then created a concrete Lesser Dragon class that inherits from the Dragon class. This design follows the DRY principle as the

Lesser Dragon has shared characteristics of Enemies such as follow behaviour, wander behaviour etc. We also added a RangedAttackBehaviour that is shared between all Dragons and a BreatheFireBehaviour that is added to Lesser Dragon. As with assignment 2, we added a AttackType.CANNOT_ATTACK_DRAGONS capability to all dragons to prevent dragons from attacking each other.

The RangedAttackBehaviour allows dragons to attack other actors that are two blocks away from the dragon. To facilitate its implementation, we have created a getRangedActors static method in the static Utils class to find potential targets that are two blocks away from a dragon. This design follows the open closed principle as any other actors that are added in the future can easily attack from a range by adding the RangedAttackBehaviour. Since RangedAttackBehaviour also accepts a weapon in its constructor, it can make a weapon become ranged when used by a non player character.

The BreatheFireBehaviour allows LesserDragon to set its surroundings on fire (within two blocks away from the dragon). The BreatheFireBehaviour follows the interface segregation principle as any actor that can breathe fire can add BreatheFireBehaviour to their list of behaviours. It also allows for dragons that don't breathe fire to not be exposed to methods that are related to breathing fire.

To facilitate the breathe fire functionality, we have created a static getRangedLocations method and addValidLocations method in the static Utils class so the dragon can set surrounding grounds on fire. To get the ranged locations, we had to call the addValidLocations method multiple times to ensure we excluded locations that were out of the range of the map. This implementation would not be good in the future if we had to add a larger radius of range. However, we assume that this is reasonably unlikely and, as part of making this tradeoff, we have tried to minimise the amount of times the getRangedLocations block of code appears by placing it making the method static in a static class. This will allow for other classes to use the getRangedLocations method in the future if we were to implement some functionality that requires range, supporting the open close principle.

Environment:

To allow lesser dragons to spawn, we have created a new DragonBarrow environment that extends SpawningGround. This was to follow the DRY principle as DragonBarrow works similar to other spawning environments in the game. We also added a new Status.FIRE_IMMUNE capability to the DragonBarrow class to prevent dragons from burning their own spawn point. This capability supports the open close principle as any other objects that implement capable are able to use Status.FIRE_IMMUNE to prevent taking on fire related damage, without the need to modify existing code.

To support the breathe fire functionality, we have created a new abstract class DamageGround which extends ground. This type of ground is intended to inflict damage on any actors that stand on it. We have created a new FireGround class that extends from the new DamageGround class as any actors that stand on fire ground take on fire related damage, unless they have the Status.FIRE_IMMUNE capability. The FireGround lasts for three turns before turning into Dirt.

This design supports the open close principle as new types of ground that deal damage can be easily added in the future by extending the DamageGround class. The DamageGround class being abstract means it doesn't have to deal with all special cases where a ground might deal damage (eg. FireGround implementation may be different to PoisonGround, which may apply special status effects), and instead leaves this implementation to the child classes (single responsibility principle).

Pros	**Cons**
- Enemies that can carry out ranged attacks can be easily added by using RangedAttackBehaviour - Enemies that can breathe fire can be easily added by using BreatheFireBehaviour - Additional functionalities that require checking the actors or locations within range (two blocks away) can utilise the methods in the Utils class - Additional grounds that deal damage can be easily implemented by extending the DamageGround class - Additional objects that we want to avoid taking fire-related damage can have Status.FIRE_IMMUNE capability	- If we want to add a larger radius of range in the future our code will become bulky and repetitive, however as we have discussed we have assumed this to be unlikely.

Potential Improvements:

- We could add a FireCapable interface which could be implemented by LesserDragon in case we want to add more complex fire-related capabilities in the future. However, since we're not sure how complicated these capabilities will really be in the future, BreatheFireBehaviour is a good option for now as it integrates easily with the existing engine.