FIT2099 Assignment 2 Design Rationale – Requirement 1

Design Rationale

Requirement 1

Implementation of Offspring Production

To represent actors that can reproduce, the Reproducible interface is created, providing them with a *reproduce* and *checkReproduceCondition* method – so each Reproducible can implement their own unique reproduce logic and the conditions that allow it to reproduce. This allows for each Reproducible to implement their own reproduction logic (whether they drop an egg or create new offspring) as well as the conditions for allowing reproduction. This adheres to SRP (Reproducible is only responsible for defining unique reproduction method and condition) and OCP (none of the existing code must be modified to make or create a new actor that can reproduce, and any unique reproduction logic can be implemented). To meet the requirement of spirit goat and omen sheep being able to reproduce, they use this interface and have their respective reproduction methods implemented uniquely (e.g., *checkReproduceCondition* returns true if surrounding entities have the BLESSED_BY_GRACE enumeration).

To ensure that the offspring production is checked every turn, the ReproduceBehaviour method (implements Behaviour interface) is added to every Reproducible – this design choice uses the existing framework for behaviours in the game engine that allows for actions to execute. ReproduceBehaviour also has a TurnCounter attribute and takes in how many turns it takes for the Reproducible to reproduce as an argument – for Reproducible that do not have turn-based reproduction conditions, '1' can be inputted for the condition to trigger true every turn. The TurnCounter could have also been placed in the Reproducible class and checked in *checkReproduceCondition*, but the more shared reproduction logic (turn-based logic, even every 1 turn is shared) that can be handled outside the Reproducible and rather inside ReproduceBehaviour, the less code repetition and better adherence to SRP. ReproduceBehaviour's use of interfaces to access the methods also adhere to DIP (ReproduceBehaviour doesn't rely on concrete methods).

ReproduceBehaviour will return a ReproduceAction if the condition check is successful, extending Action to allow for use of the game engine's action handling every turn – it also relies on the abstract methods of Reproducible – calling its *reproduce* method to execute its unique reproduction event – adhering to DIP. The ReproduceAction is also solely responsible for ensuring the reproduction actually occurs, adhering to SRP, and this implementation is OCP friendly (no modification of existing code was required, just extension of Action). Furthermore, it can accept any Actor so long as it's implemented Reproducible, adhering to LSP.

Disadvantages and Justification:

- Not all Reproducible have a non-turn based reproduce condition, meaning that for these, *checkReproduceCondition* will just be an arbitrary method that always returns true, violating ISP as the method is not needed in this case. This is justified because the turn-based logic being in ReproduceBehaviour adheres to SRP and DRY – if it was in implementations of Reproducible it would violate these (SRP because actor given too much responsibility with turn logic, DRY because this would have to be repeated for every Reproducible instance). This could be mitigated with an implementation similar to HatchingStrategy, having unique implementations for different types of reproduction conditions, however the overall project becomes too polluted with too many different types of classes handling Reproduction behaviour – hence overengineering for the scale this problem presents.
- Coupling is present in ReproduceBehaviour and ReproduceAction as it takes in a Reproducible as an argument. This is done so that Reproducible's methods can be used in ReproduceAction – which allows adherence to ISP (ReproduceAction relies on Reproducible interface rather than using downcasting/instanceof to call the Reproducible methods). An alternate solution could be to include Reproducible in one of the Status enumerations, however these are already quite congested and will become even more so in the future – this implementation rather leverages the existing interface that is needed for the requirement – the Status enumerations should be saved for cases where there isn't a corresponding interface for the enumeration.
- Coupling is also present with the ReproduceBehaviour holding an instance of TurnCounter – however doing this eliminates having to handle turn counters in the behaviour itself, helping eliminate strong forms of connascence and adhering to DRY (Connascence of Algorithm, because any form of change in turn-based logic/incrementation must be changed across all methods that implement turn-based logic)

Implementation of Eggs and Hatching

The Egg abstract class is created, an extension from Item so it can have the behaviour of other items in the game, such as being able to be picked up. Other eggs all have the shared behaviour of having an abstract hatch method, a tick method that checks if the hatch condition is valid for hatching, as well as a method returning true if the egg has been picked up. This adheres to OCP – new eggs can easily be implemented with their own unique hatch method and extends from the game code without modifying the existing engine. This also adheres to LSP as eggs can be treated as any normal item in the game engine. This allows for the requirement of implementing OmenSheep eggs.

To adhere to eggs having different types of hatching methods, an interface HatchingStrategy is created – the two currently created are turn-based and surrounding-status based, however new types of hatching methods can easily implement this design – adhering to OCP. HatchingStrategy encapsulates the hatching logic and conditions, meaning Egg implementations don't have to worry about repeating these – adhering to DRY. Furthermore, if there are similar types of hatching methods that share same behaviour (e.g. nearby status check) this can easily be implemented as one hatching strategy and repeated for different eggs. The Egg abstract class also relies on the HatchingStrategy interface's tick method to check when the egg should hatch, adhering to DIP instead of having concrete implementations of Egg rely on unique tick methods. Furthermore, all HatchingStrategy implementations will need to apply tick logic to define how the egg hatches, adhering to ISP (method will always be used). HatchingStrategy is also only concerned with the conditions for hatching and seperates these from the Egg implementations, adhering to SRP.

Disadvantages and Justification:

- Potential speculative generality in the HatchingStrategy interface as must add class for every new unique case of hatching not covered by existing classes – could be overengineering, however the current two: turn-based, and surrounding-status based, are quite common mechanisms in different parts of the code and will most likely be reused. This implementation is more also OCP and DRY friendly than having to repeat these common mechanisms.
- Middle-man delegation with the *tick* method in egg, which simply forwards to HatchingStrategy – this is done because it is the only way to connect the unique hatching conditions to the game's engine loop (since egg's tick method is called as an extension of item)

Implementation of Eating

The EatAction class is created and extends from the Action class, using the engine's code to allow for the action to be added to the user's possible actions. This adheres to OCP – no modification of existing game code was required to implement this and it fits seamlessly with the game engine. This also adheres to SRP – EatAction is only concerned with executing the *eat* method of the Edible object it accepts and then providing this action to the Player.

The Edible interface is created for items that can be eaten, with their own unique effects when consumed, through the *eat* method. To meet the requirement, OmenSheepEgg implements this interface and uses the eat method to provide stamina to the Player when consumed. This implementation adheres to SRP (Edible is only concerned with the effects of consuming the Edible), OCP (new Edibles can easily be created with unique *eat* effects without modification of existing code), and ISP (every Edible will need to implement the *eat* method to specify its effect of being consumed) and DIP (EatAction relies on the Edible interface instead of a concrete implementation of an Item with an eat method).

Disadvantages and Justification

- Every Edible must override *allowableActions* to add the EatAction – violation of DRY. This can be solved by adding another abstract EdibleItem class extending Item and implementing Edible, however this is overengineering the current scope – as only Egg is edible – and the cost of repeating the override does not outweigh the cost of creating this new class.

- The *execute* method in EatAction is essentially a middle-man for the Edible's *eat* method, however this is the only way for the Edible to have its consumption effects occur in the game code, as the engine calls the *execute* method as it runs – hence a necessary code smell.

Changes from original design (A1)

The only major change made was the splitting up of Status enum into different enums (ActorStatus, EntityStatus), as certain statuses were doubling up between GroundStatus and Status. This new definition means that there is less repition for statuses that are shared between Ground and Actor (can be put into EntityStatus) and also more clearly defines what statuses belong to which types of entities.