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).

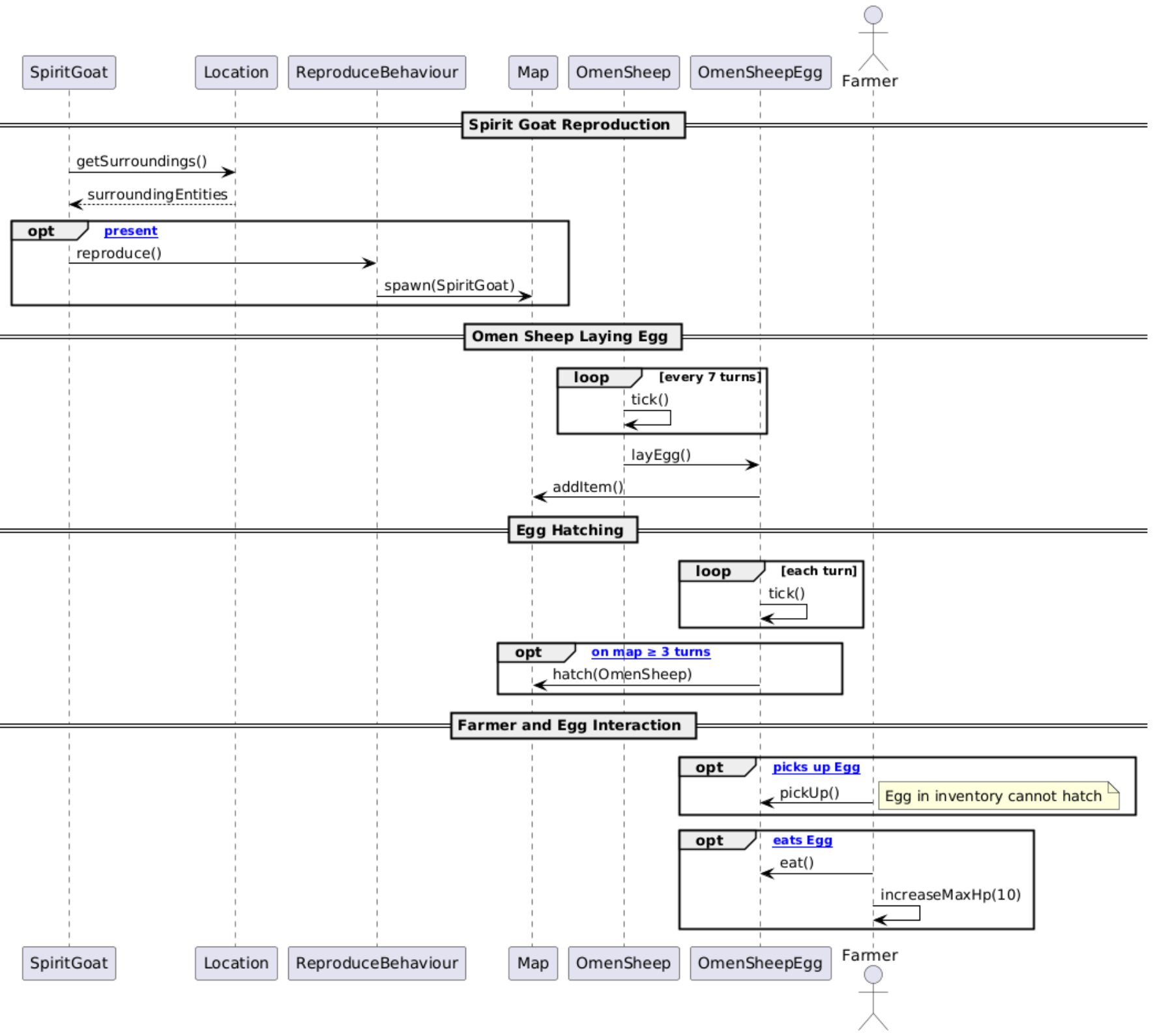
Disadvantages and Justification

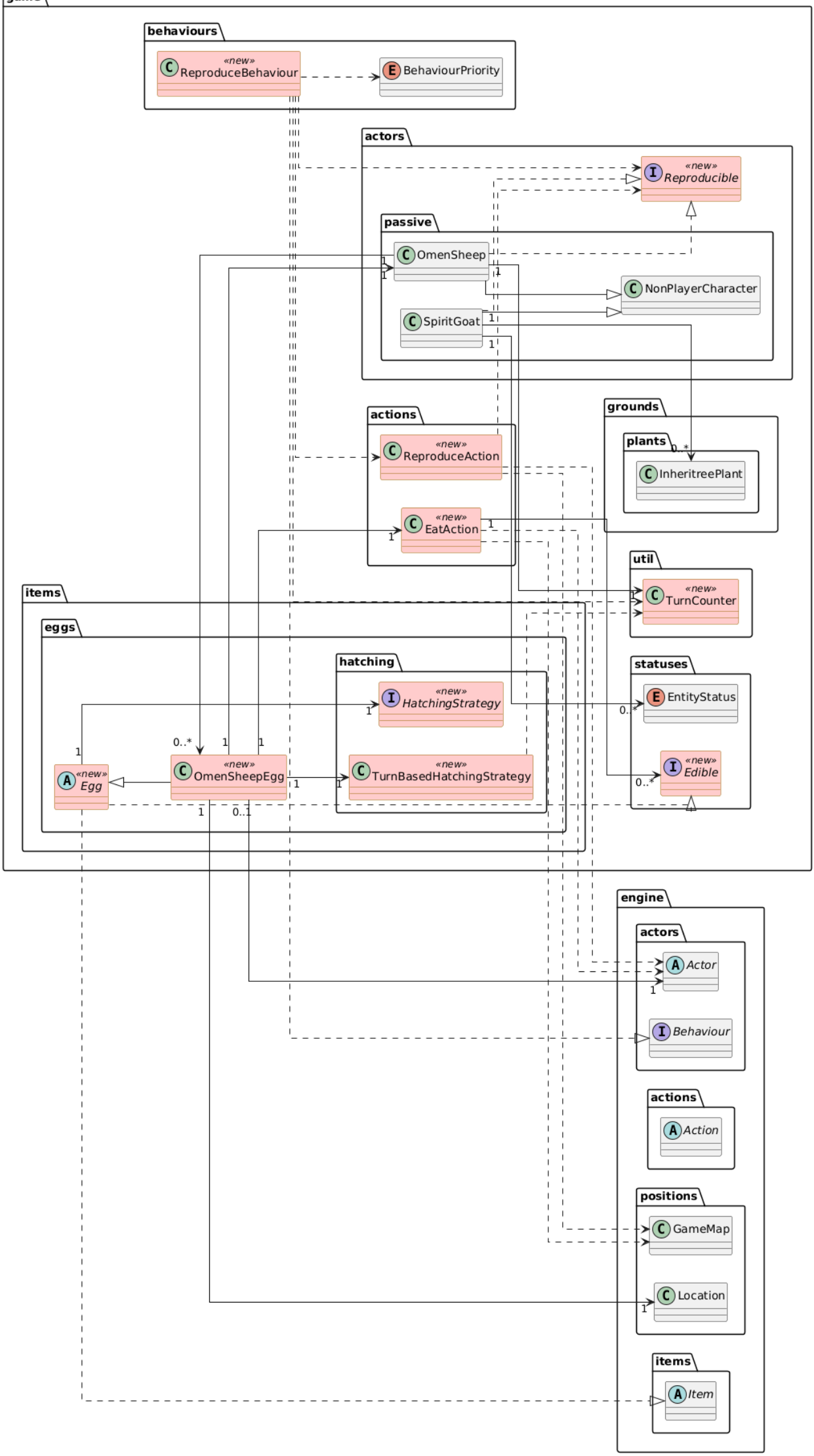
- Every Edible must override *allowableActions* to add the EatAction – violation of DRY. This can be solved by adding another abstract EdibleItem class extending Item and implementing Edible, however this is overengineering the current

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.


SpiritGoat
Location
ReproduceBehaviour
Map
OmenSheep
OmenSheepEgg
Farmer
Spirit Goat Reproduction
getSurroundings()
surroundingEntities
opt
present
reproduce()
spawn(SpiritGoat)
Omen Sheep Laying Egg
loop
[every 7 turns]
tick()
layEgg()
addItem()
Egg Hatching
loop
[each turn]
tick()
opt
on map ≥ 3 turns
hatch(OmenSheep)
Farmer and Egg Interaction
opt
picks up Egg
pickUp()
Egg in inventory cannot hatch
opt
eats Egg
eat()
increaseMaxHp(10)
SpiritGoat
Location
ReproduceBehaviour
Map
OmenSheep
OmenSheepEgg
Farmer


game
behaviours
«new»
ReproduceBehaviour
BehaviourPriority
actors
«new»
Reproducible
passive
OmenSheep
NonPlayerCharacter
SpiritGoat
actions
«new»
ReproduceAction
«new»
EatAction
grounds
plants
InheritreePlant
util
«new»
TurnCounter
items
eggs
hatching
«new»
HatchingStrategy
statuses
EntityStatus
«new»
Egg
«new»
OmenSheepEgg
«new»
TurnBasedHatchingStrategy
«new»
Edible
engine
actors
Actor
Behaviour
actions
Action
positions
GameMap
Location
items
Item

FIT2099 Assignment 2 Design Rationale – Requirement 2

Design Rationale

Requirement 2

Note: This Design Rationale is short in comparison to REQ1 as the underlying design for GoldenBeetle and GoldenEgg is discussed in REQ1 – its justifications, advantages, and disadvantages.

Implementation of GoldenBeetle and GoldenEgg

GoldenBeetle and GoldenEgg are simply extensions of the NonPlayableCharacter and Egg classes respectively – adhering to OCP as no modification of the existing code was required. Both of these also implement the Edible interface as they can both be consumed by the Player, and implement their own eat methods for their unique behaviour - for example, in GoldenBeetle.eat, the actor will gain 15 health and 1000 runes through the Capability system in the engine. The GoldenBeetle also passes the amount of turns it takes to hatch (5) for its ReproduceBehaviour, to meet this requirement.

For the GoldenEgg, it uses the SurroundingStatusHatchingStrategy – and checks for nearby cursed entities – this HatchingStrategy accepts a status enum and checks surrounding entities (both ground and actors) – so it can be extended to any other status, such as grace – adhering to DRY and OCP.

Implementation of Runes

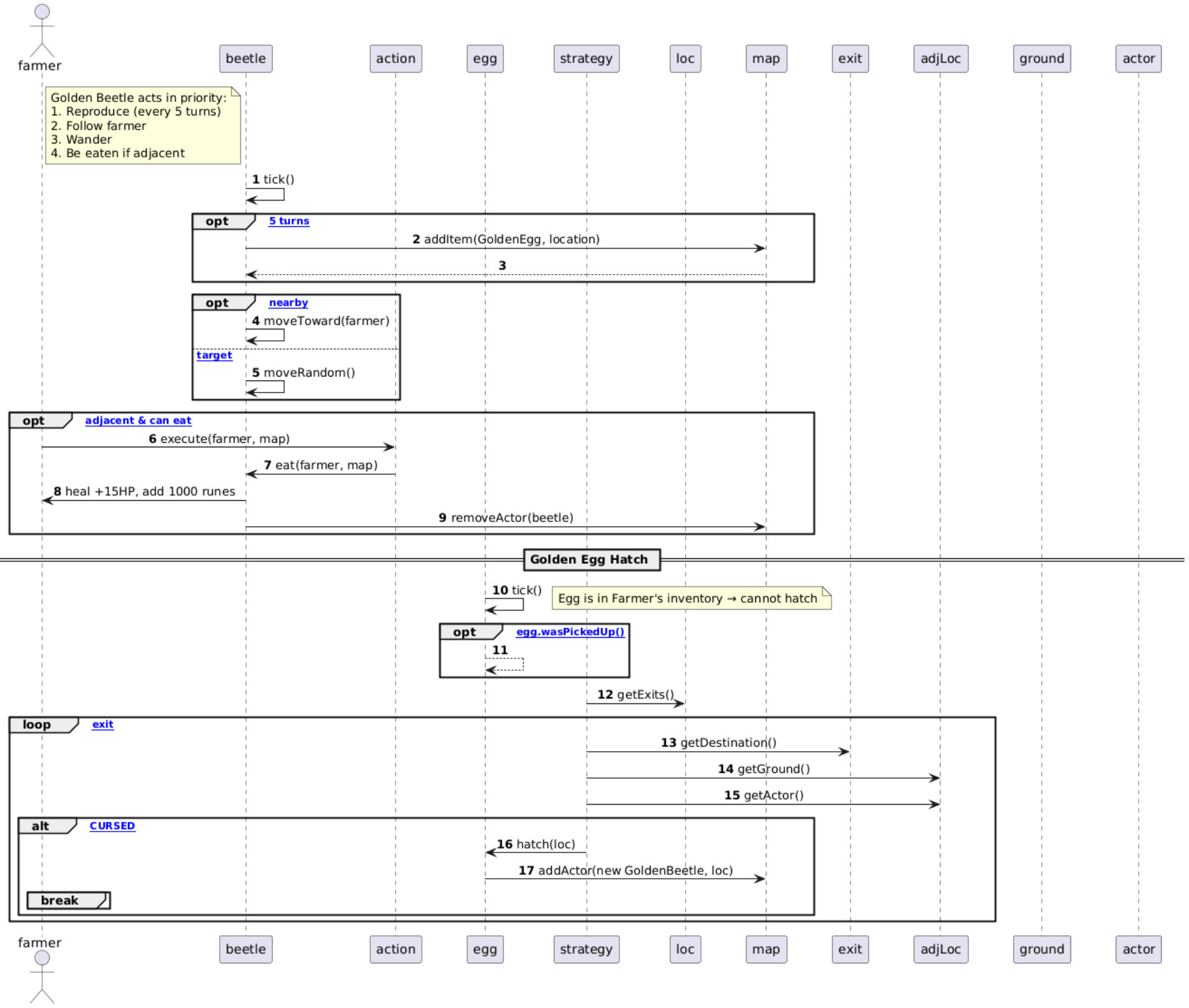
A new Enumeration – GameActorAttributes – is created, storing RUNE which can be added to the capabilities storage of the Player. This required minimal modification of the existing code and uses the already existing methods in the game engine to allow for interaction and storage of the runes – hence adhering to OCP.

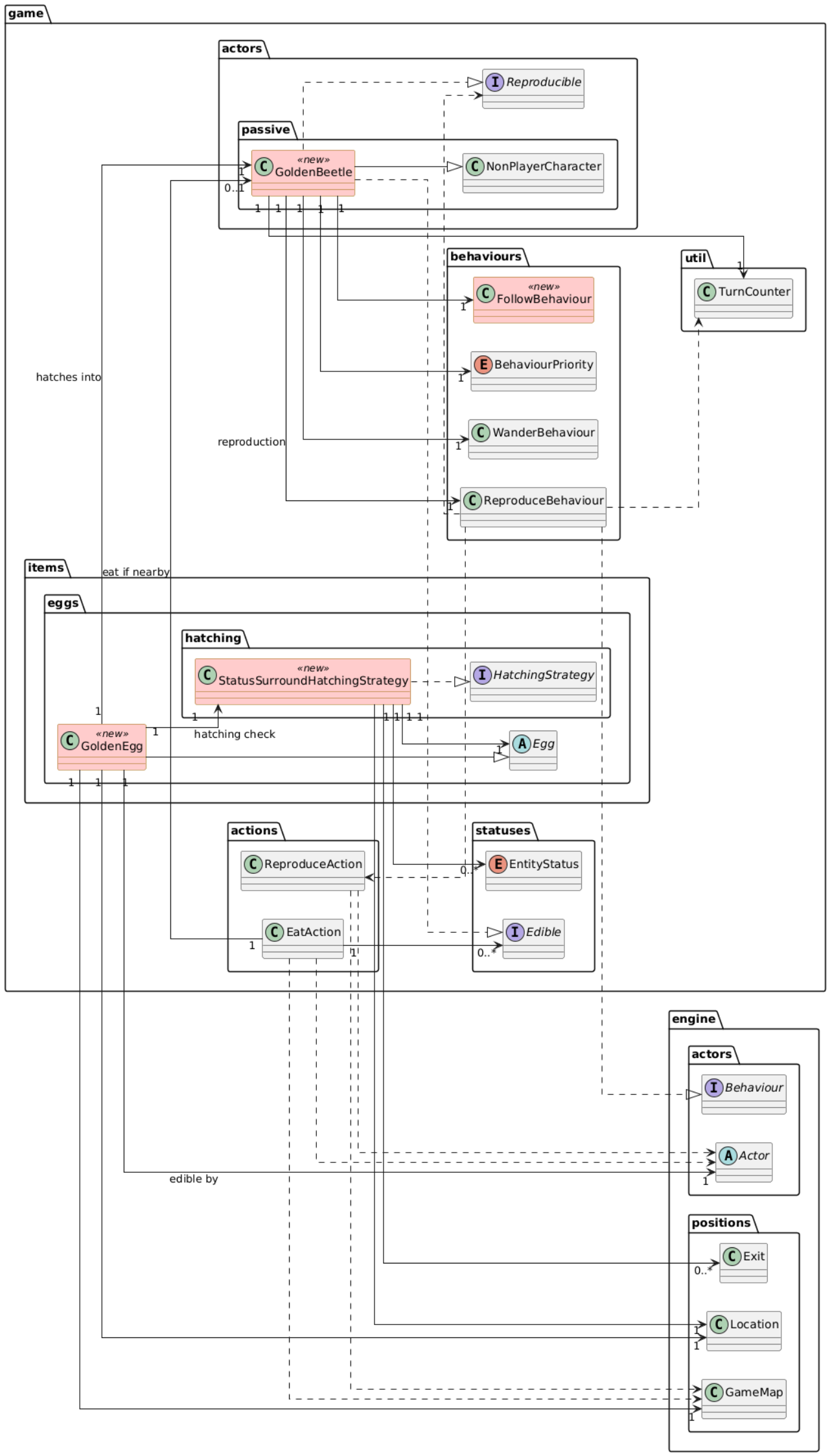
Implementation of Following

The FollowBehaviour class, implementing Behaviour, integrates with the engine code to allow for an NPC to follow a FOLLOWABLE actor, and iterates over the entire map to find this actor – once found, it will store the actor and continue to follow by getting the closest path to its position until the actor dies – then finding a new actor to follow. This meets all the requirements and adheres to SRP (only implements logic for following), and OCP (no modification of engine code required to implement). If actors need to be marked as followable, they can simply add FOLLOWABLE from the ActorStatus enum.

Disadvantages and Justification

- The following behaviour has tight coupling to the FOLLOWABLE enum – a potential better implementation would be to have actors implement a Followable interface, however this could violate ISP as there is no current unique following behaviour, and hence could also be speculative generality – so the current implementation is preferred – also avoids downcasting and instanceof.


farmer
beetle
action
egg
strategy
loc
map
exit
adjLoc
ground
actor
Golden Beetle acts in priority:
1. Reproduce (every 5 turns)
2. Follow farmer
3. Wander
4. Be eaten if adjacent
1 tick()
opt
5 turns
2 addItem(GoldenEgg, location)
3
opt
nearby
4 moveToward(farmer)
target
5 moveRandom()
opt
adjacent & can eat
6 execute(farmer, map)
7 eat(farmer, map)
8 heal +15HP, add 1000 runes
9 removeActor(beetle)
Golden Egg Hatch
10 tick()
Egg is in Farmer's inventory → cannot hatch
opt
egg.wasPickedUp()
11
12 getExits()
loop
exit
13 getDestination()
14 getGround()
15 getActor()
alt
CURSED
16 hatch(loc)
17 addActor(new GoldenBeetle, loc)
break


game
actors
Reproducible
passive
«new»
GoldenBeetle
NonPlayerCharacter
behaviours
«new»
FollowBehaviour
BehaviourPriority
WanderBehaviour
ReproduceBehaviour
util
TurnCounter
hatches into
reproduction
eat if nearby
items
eggs
hatching
«new»
StatusSurroundHatchingStrategy
HatchingStrategy
hatching check
«new»
GoldenEgg
Egg
actions
ReproduceAction
EatAction
statuses
EntityStatus
Edible
edible by
engine
actors
Behaviour
Actor
positions
Exit
Location
GameMap

REQ3 – Design Rationale: NPC Dialogue

REQ3 introduces special NPCs (Sellen, Kale, and Guts) capable of generating contextual monologues based on the player's status (e.g., health, inventory) and surrounding ground (e.g., BLIGHT). The system must support multiple types of dialogue logic while remaining modular, extensible, and cleanly separated from presentation or behavior logic. Additionally, REQ3 involves hostile actors that require cleanly defined, reusable combat behavior logic.

Design 1: Hardcoded dialogue directly in NPC

Pros	Cons
Simple and fast to implement	High code duplication; not scalable
No extra classes or wiring required	Logic and presentation tightly coupled
Easy to debug when NPC count is low	Difficult to test or update behavior independently

Design 2: Centralized ListenAction

Primary Design

Pros	Cons

Cleanly separates output (Action) from logic (NPC)	Slightly more setup per NPC
Highly reusable across multiple NPCs	Each NPC must still prepare appropriate lines
Centralized formatting and interaction contract	Custom behaviors still live inside NPC

Design 3: Passive

MonologueBehaviour

Used by Sellen

Pros	Cons
Adds atmospheric, lore-rich worldbuilding	Not interactive; no player-triggered response
Fully decoupled from player actions	Cannot respond to changing player or map state
Lightweight and modular	Limited contextual flexibility

Design 4: Inline contextual logic in NPC (Used by Kale & Guts)

Pros	Cons
Supports dynamic dialogue based on player stats or environment	Logic tightly coupled to each NPC class
Enables immersive, responsive interactions	Low reusability unless abstracted
Simple and readable for small-scale use	Harder to share logic between NPCs

---

Design 5: Interface-based Attack Logic via

HostileAttacker

Combat Behavior Design

Pros	Cons
Removes instanceof and downcast in AttackBehaviour	Requires extra interface and constructor setup
Follows OOP principles (OCP, DIP)	Slightly more boilerplate per hostile actor

Behavior is now injected, not hardcoded	Requires coordination between NPC and behavior
Enables clean testing, reuse, and logic separation	Must manage capability + logic explicitly

Justification

The final design of Requirement 3 integrates four layered strategies to support scalable NPC interaction, dialogue delivery, and hostile behavior management. It combines Design 2 (ListenAction) as the reusable interaction mechanism, Design 3 (MonologueBehaviour) for passive ambient dialogue, Design 4 for contextual monologue generation within individual NPCs, and Design 5, which introduces an interface-driven attack logic system via HostileAttacker.

To support modular dialogue presentation, ListenAction is used across all interactable NPCs to decouple the *what* (message generation) from the *how* (rendering and formatting). This enables NPCs to follow a unified interaction contract while centralizing output logic in a single reusable class. For instance, characters like Kale and Guts dynamically generate monologues based on player state (e.g., health, inventory), but offload the presentation to ListenAction, thereby separating logic from UI. For non-interactive ambiance, MonologueBehaviour enables characters like Sellen to periodically speak without direct player involvement, enriching world immersion with minimal coupling.

In terms of combat logic, a key architectural enhancement is the introduction of the HostileAttacker interface. Hostile NPCs such as Guts implement this interface to define custom canAttackTarget() logic (e.g., only attacking conscious players with sufficient HP). This interface is injected into AttackBehaviour, allowing hostile logic to be reused or extended without modifying the behavior controller. This approach aligns with the Dependency Inversion Principle (DIP) and eliminates the need for instanceof or downcasting, promoting polymorphic flexibility and reducing long-term coupling.

The design strongly adheres to SOLID principles:

- Single Responsibility Principle (SRP): Dialogue generation, rendering, and combat logic are each handled by separate, focused modules (NPC, ListenAction, HostileAttacker, etc.).

- Open/Closed Principle (OCP): New types of dialogue or combat logic can be introduced without altering existing code.

- Liskov Substitution Principle (LSP): All NPCs using ListenAction or MonologueBehaviour conform to the same behavioral contract and can be substituted interchangeably.

- Interface Segregation Principle (ISP): Only relevant NPCs implement HostileAttacker; passive ones remain unaffected, ensuring minimal interface overhead.

- Dependency Inversion Principle (DIP): High-level behavior logic in AttackBehaviour depends on abstract interfaces rather than concrete implementations like Guts.

This architecture also effectively avoids several code smells:

- Duplicated Code: Dialogue output and combat validation are centralized.

- Shotgun Surgery: Changes to dialogue formatting only affect ListenAction.

- God Classes: NPCs maintain single, focused responsibilities without becoming bloated.

From a connascence perspective, this design reduces both Connascence of Name and Connascence of Algorithm between NPCs and behaviors. Behavior classes like AttackBehaviour do not need to know the internal logic of their actors, but simply depend on well-defined contracts such as canAttackTarget(). This lowers coupling, enhances readability, and simplifies future refactoring.

Lastly, the system aligns with Design-by-Contract principles. NPCs commit to providing valid outputs (dialogue or attack decisions), while ListenAction and AttackBehaviour guarantee consistent and structured execution of those outputs. Each component operates under clearly defined preconditions and postconditions, ensuring predictable behavior and facilitating robust testing and extension.

edu.monash.fit2099

engine

actors

attributes

BaseActorAttributes

<<Abstract>>
Actor

<<Interface>>
Behaviour

<<Abstract>>
Action

positions

GameMap

Game

status

entitystatues

Behaviours

weapons

BareFist

AttackBehaviour

ENUMS

DEATH(1),
ROTTING(2),
ATTACK(3),
FOLLOW(4),
WANDER(5),
PICKUP(6),
DROP(7),
WAIT(8),
LOW(9);

MonologueBehaviour

WanderBehaviour

actors

Attributes

GameActorAttributes

<<interface>>
HostileAttacker

NonPlayerCharacter

Player

Passive

hostile

Guts

Kale

Sellen

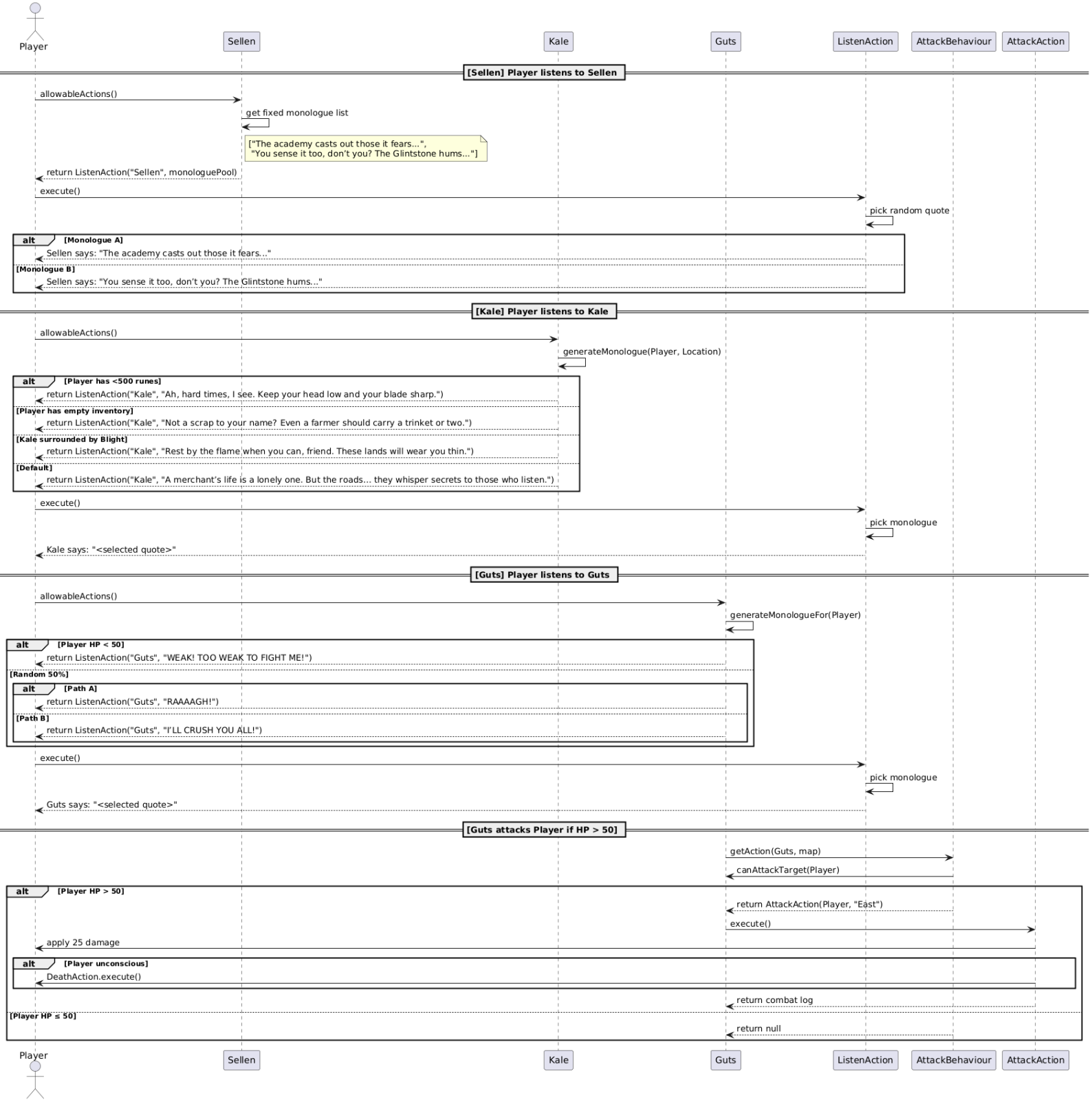
actions

ListenAction

MonologueAction

AttackAction

REQ3 - Player interacts with Sellen, Kale, Guts (Dialogue + Combat)



REQ4 – BUYING WEAPONS

The diagram models the buying process required by REQ4: the Farmer may purchase weapons directly from Merchant NPCs (Sellen or Kale). Weapons (Broadsword, DragonslayerGreatsword, Katana) extend the engine's IntrinsicWeapon for combat stats and implement the Sellable interface to encapsulate buying behavior. Merchants (Sellen, Kale) offer BuyActions conditionally based on the player's runes, injecting different constructor parameters to trigger merchant-specific side effects.

Sellable Interface for Cohesion & DRY

The sellable interface is a minimal interface with getSellPrice(), onSell(Player,GameMap), and getName(). This encapsulates all purchase logic and side effects within each weapon class, so BuyAction depend only on Sellable. This would allow future implementation of Sellable items that aren't weapons.

Weapon Classes

Each weapon class extends IntrinsicWeapon (reusing engine combat mechanics) and implements Sellable. Each class also defines merchant-specific behaviour which is passed in via constructor parameters. This leaves the weapon classes open for extension. New weapons or side effects can be added by creating new classes, without altering existing code (Open/Closed Principle).

Interface Segregation

NPC classes only reference the Sellable interface. Any new sellable item can be offered without modifying merchant logic.

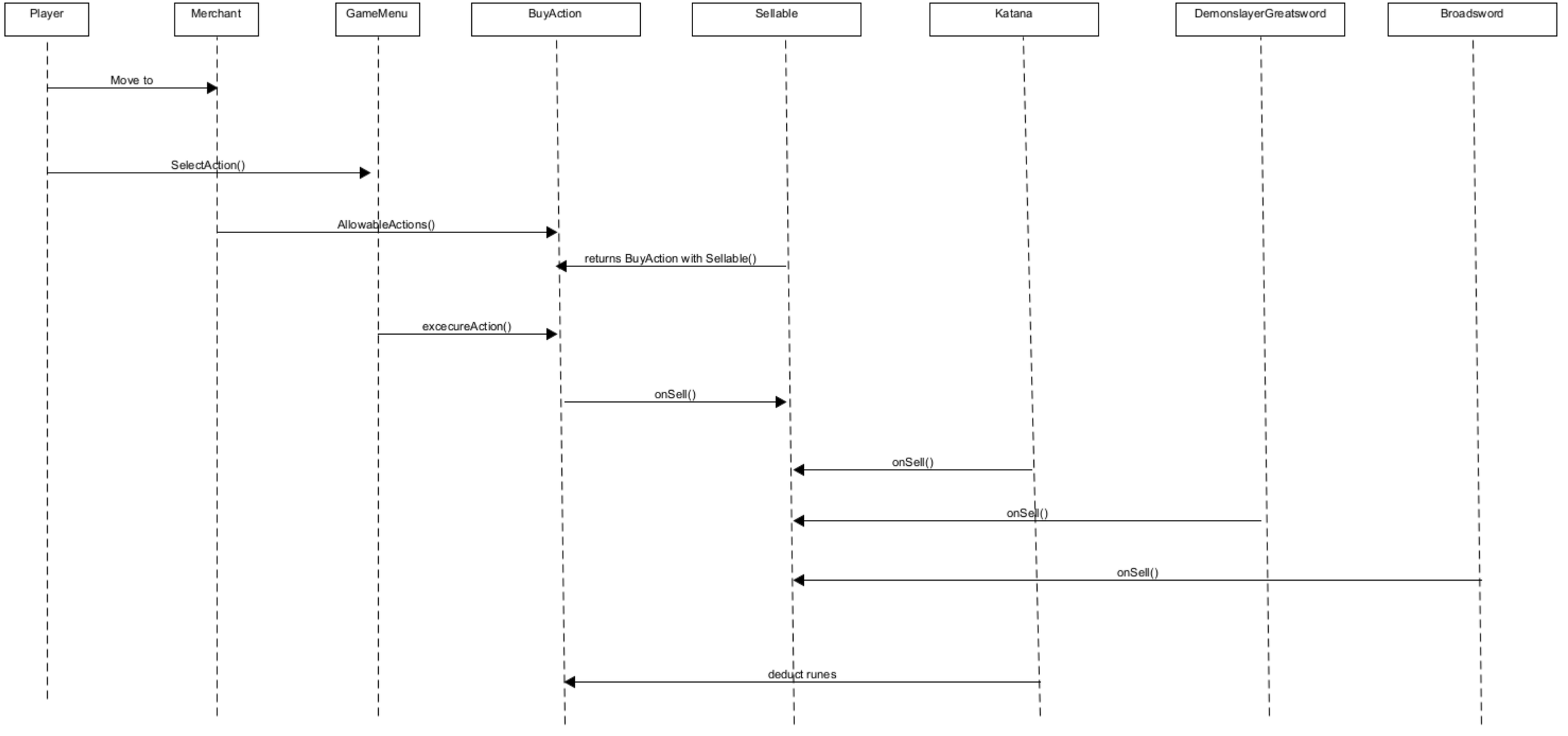
Splitting combat (IntrinsicWeapon) from purchase (Sellable) respects Interface Segregation. Every interface has one clear responsibility.

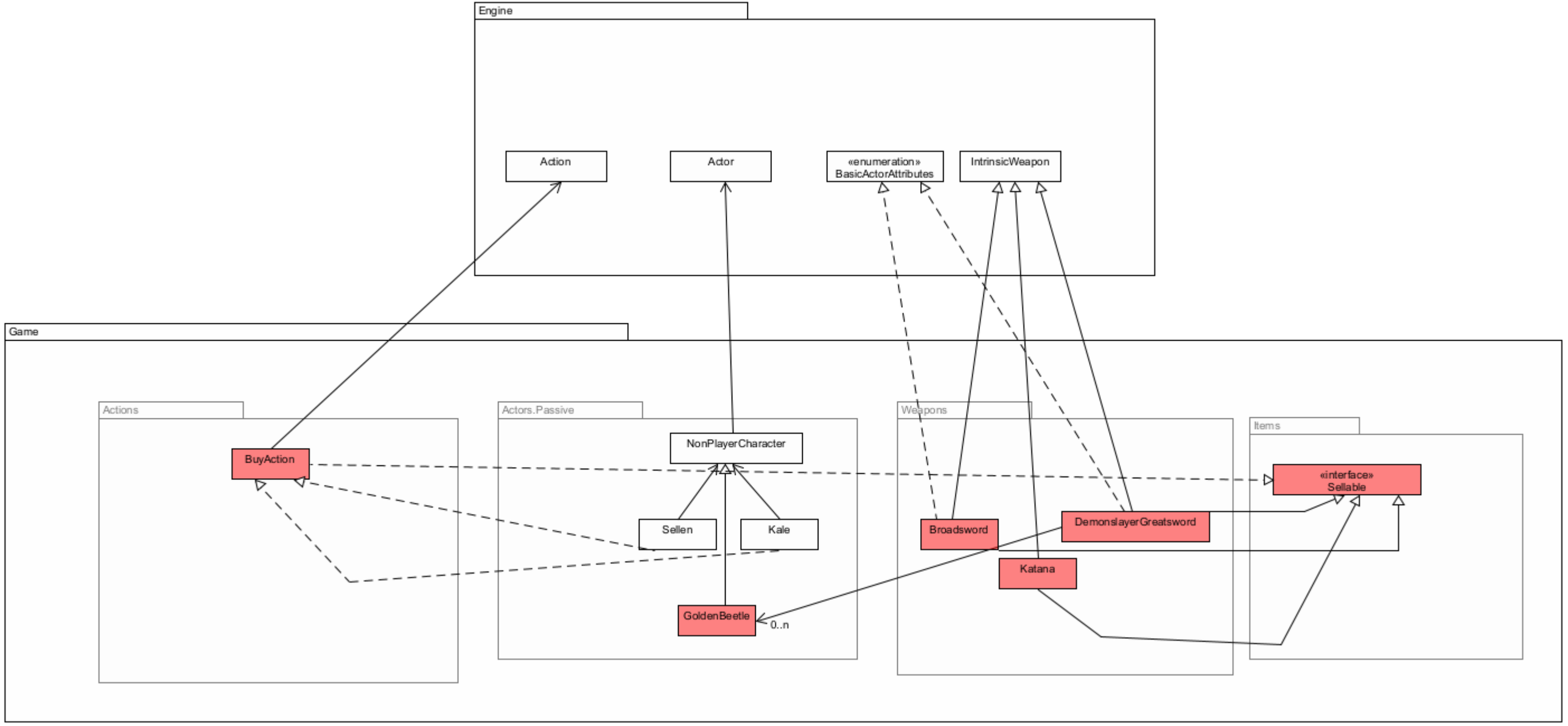
Trade-Offs & Limitations

- **Duplication in NPC Offerings:** Both Sellen and Kale list their buyable items manually; a shared builder under the merchant could reduce boilerplate. However this would just move the complexity for side effects rather than removing it.
- **Per-Weapon Side Effects:** Housing spawn logic and multiple attribute changes in each class increases class size, but ensures each weapon's unique behaviour stays local and specific to that weapon.

Principles Applied

- **Single Responsibility:** Sellable items handle only their purchase effects.
- **Open/Closed:** New weapons or merchants can be added by subclassing and implementing interfaces, without modifying core engine or existing classes.
- **Low Coupling:** NPCs and BuyAction depend only on Sellable and Action, weapon classes depend on engine abstractions.
- **High Cohesion:** Each class has one clear purpose, combat, purchase effect, or merchant behaviour.


Player
Merchant
GameMenu
BuyAction
Sellable
Katana
DemonslayerGreatsword
Broadsword
Move to
SelectAction()
AllowableActions()
returns BuyAction with Sellable()
excecureAction()
onSell()
onSell()
onSell()
onSell()
deduct runes


Engine
Action
Actor
«enumeration»
BasicActorAttributes
IntrinsicWeapon
Game
Actions
BuyAction
Actors.Passive
NonPlayerCharacter
Sellen
Kale
GoldenBeetle
0..n
Weapons
Broadsword
DemonslayerGreatsword
Katana
Items
«interface»
Sellable