REQ2 - Be Wary Of Dung (Beetle)

Implementation A: Extending PeacefulCharacter by creating a new GoldenBeetle class that implements ProduceEgg and Edible based on REQ1

Pros	Cons
Reuses existing non-hostile NPC structure	Requires careful implementation of overridden methods such as onProduceEgg and getRunesGranted, which may cause unwanted errors if not done so correctly
Maintains consistency with the game's current implementation of passive Actors	Has tight coupling with PeacefulCharacter making future changes difficult if the base class is also changed
Easy to integrate with existing iteration of code	

Implementation B: Modified the Edible interface to add in new methods such as getRunesGranted() [defaulted to 0], onConsume() and getConsumeDescription()

Pros	Cons
Enhances flexibility by allowing different edible actors to have custom consumption outcomes	Increases interface complexity and maintenance of more classes
Supports future gameplay mechanics like consuming different creatures for varying benefits, making code extensible	Classes that implementing Edible now need to account for more methods, even if not all are relevant
Centralises edible-related logic by condensing it into one interface	Adding too many default behaviours to the interface can increase the risks of violating the Interface Segregation Principle

Implementation C: Created a HatchCondition interface which defines different conditions in which an Egg can hatch (such as only being near cursed entities)

Pros	Cons
Handles hatching logic independent from the Egg class, adhering to the Open Closed Principle	Can increase the risks of potential over-engineering if more, and complex hatching conditions are introduces
Highly extensible since new hatch conditions can be added without modifying the Egg class	Can complicate debugging when hatch logic doesn't behave as expected due to hidden conditions
Can allow different Eggs of the same type to have different hatch conditions, increasing customisability of the game engine	

Implementation D: Abstracted the Egg class so it can be split up into different subclasses such as GoldenBeetleEgg and OmenSheepEgg

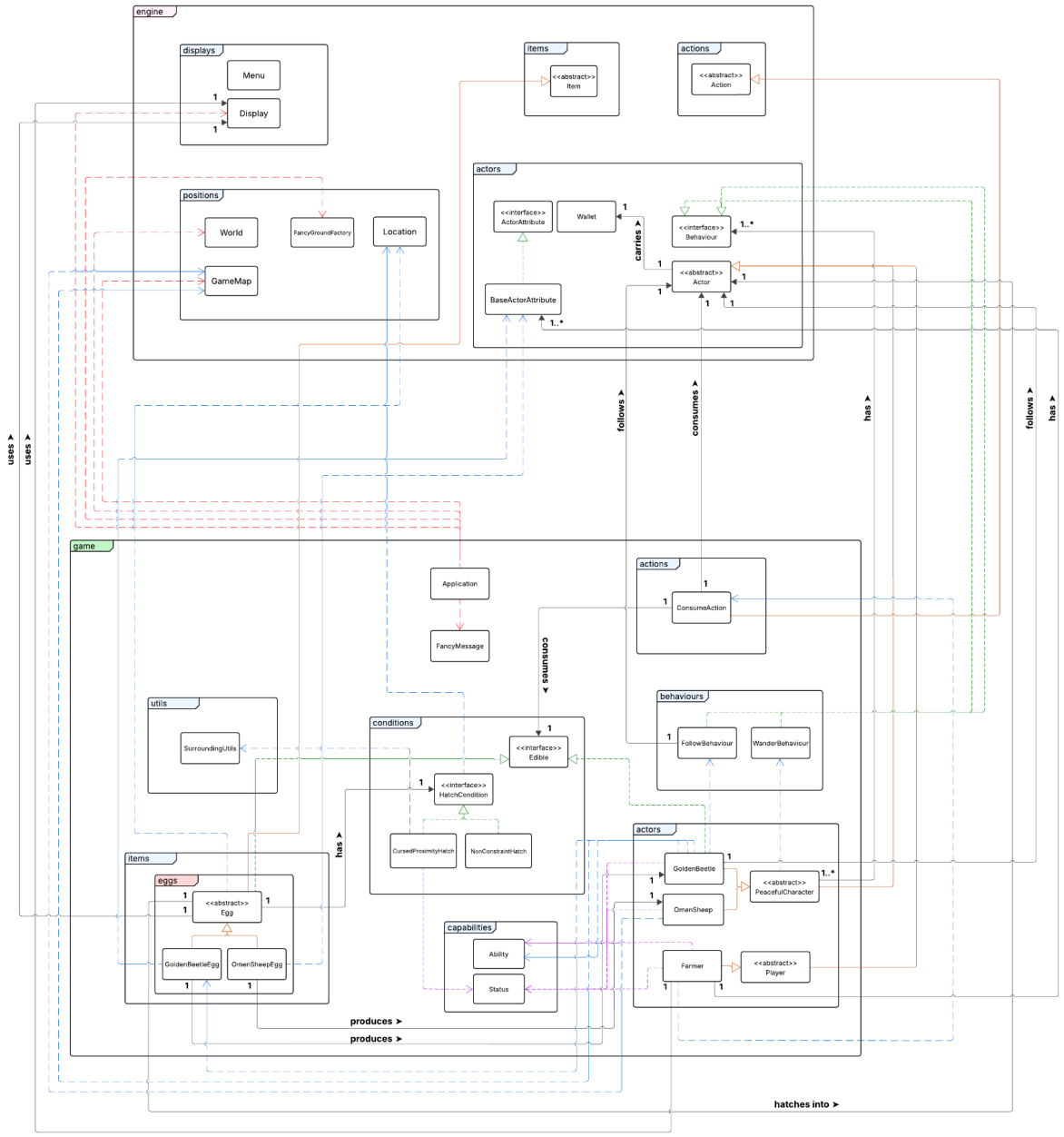
Pros	Cons
Promotes reuse of common egg logic while allowing subclass-specific behavior	Increases complexity by introducing a deeper class hierarchy
Simplifies management of egg behaviors like hatching, placement, and tick logic	Can introduce duplicated code if subclasses share similar behaviour
Centralises core-egg-related logic by condensing it into one base class, increasing extensibility	Adding too many default behaviours to the interface can increase the risks of violating the Interface Segregation Principle

Implementation E: Modified the Farmer class to contain a new attribute called Wallet to track rune gains

Pros	Cons
Cleanly encapsulates rune tracking in a dedicated field	Tightly couples rune economy logic with the player class, which may reduce flexibility
Enables future extensions like spending or trading runes	Wallet needs to be updated consistently/correctly for all relevant actions

Implementation F: Created/Copied the FollowBehaviour present in the demo folder to allow GoldenBeetle to follow the Farmer, further creating private methods such as findAllowableActor() and handleFollowBehaviour

Pros	Cons
Encourages reusability by modularising follow logic into methods	Code duplication may be required in the future if further creatures also have follow behaviour parameters similar to GoldenBeetle
Clearly encapsulates follow logic since the dedicated methods isolate behaviour	



Note: Coloured Lines have been used to make distinctions between different/overlapping relationships

- Red = Dependencies from Application
- Blue = Dependencies
- Orange = Abstract Inheritance
- Purple = Enumeration
- Green = Interface Relationships
- Black = Association

The diagram above shows the final design of REQ 2. The GoldenBeetle class extends PeacefulCharacter and implements both ProduceEgg and Edible, allowing it to lay eggs and be consumed by the Farmer. This adheres to the Single Responsibility Principle, by encapsulating egg production/consumption within their respective classes.

The FollowBehaviour class enables the GoldenBeetle to follow the Farmer dynamically. This behavior ensures that the GoldenBeetle smoothly follows a target with the CAN_BE_FOLLOWED status only when it is in close proximity, extending to future implementations if more followable actors are introduced.

The abstract Egg class defines shared logic for hatching and consumption, while specific Eggs such as GoldenBeetleEgg and OmenSheepEgg override custom behaviours such as onConsume(), and provide different HatchCondition strategies. This follows the Open-Closed Principle by enabling new egg/hatch types to be added without modifying the existing code-base. This is ensured by the HatchCondition interface which defines specific Hatching conditions for Eggs (eg. CursedProximityHatch), thereby promoting modular extension and reducing connascence of algorithm between eggs and game rules.

Consumption logic is centralised by the ConsumeAction class which works with any class implementing the Edible interface. However, a slight type cast was used to ensure smooth logic by casting Actors who implement the Edible interface (eg. GoldenBeetle - see the screenshot below). While not ideal, this was a necessary compromise due to framework constraints that require actors to be passed in a generic form. Despite this, the design still supports the Interface Segregation Principle by ensuring only edible entities are required to implement consumption-specific methods. The Edible interface also includes a default getRunesGranted() method, reducing the likelihood of breaking changes when extending functionality and helping manage connascence of name between the interface and its implementations.

```java
/**
 * Constructor for consuming an Edible actor near an Actor
 *
 * @param edibleActor The edible actor to consume
 */
public ConsumeAction(Actor edibleActor) {  1 usage   sanjevanr
    this.edible = (Edible) edibleActor; // casts the edible actor to the Edible
    this.targetActor = edibleActor; // stores the actor to be removed from the map
}
```

The Farmer class tracks rune balances by including a new attribute Wallet separating economy logic from health or stamina systems and thus aligning with the Single Responsibility Principle.