FIT 2099: Assignment 2 Report

Applied class 05

Team Members:
Tee Zhi Hong (34570403)
Joel Wong Sing Yue (35478527)
Nigel Wong Wei Lun (33524904)
Liew Yi Wei (34146385)
Abdullah Saad (33023239)

Table of Contents

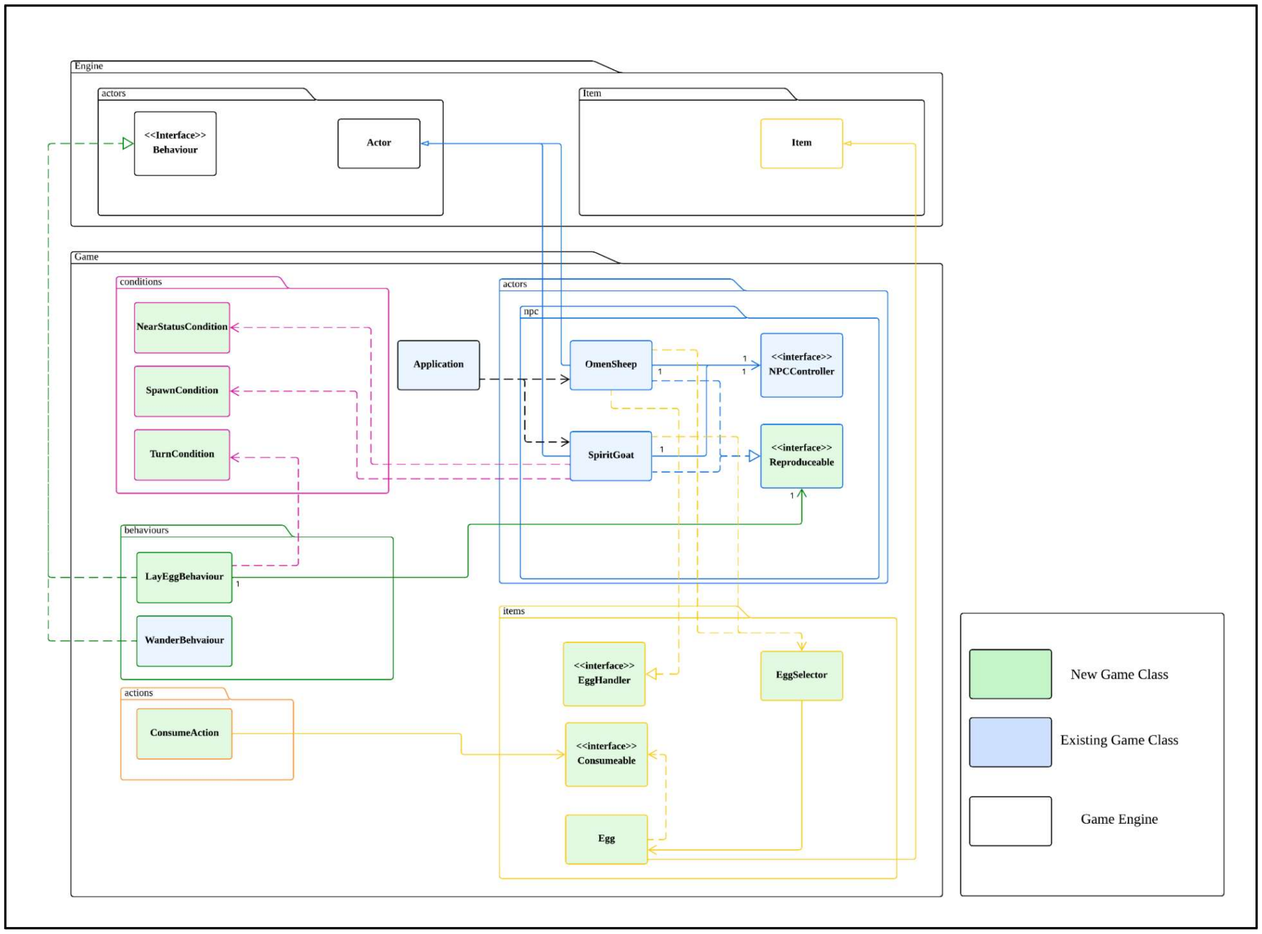
Design Rationale : REQ 1+ 2

Within REQ 1 and REQ 2, the entities of the Omen Sheep, Egg, and Golden Beetle, share many similarities and common abstract implementation. Therefore, within this section, many design rationale will bring up its implementation within both the Omen Sheep and the Golden Beetle, and the Spirit Goat occasionally.

Within **REQ 1** and **REQ 2**, the shared functionalities to be implemented for both the Omen Sheep, Spirit Goat and Golden Beetle :

Design 1 : Surrounding status based reproduction
The Spirit Goat itself and Golden Beetle's Egg trigger the reproduction according to the surrounding entities' status.
Design 2 : Turn-Based egg laying
The Omen Sheep lays an Egg after every seven turns, and the Golden Beetle lays an after five turns
Design 3 : Picking up the Egg and consuming the egg
The Egg created by the Omen Sheep and Golden Beetle can be picked up by the Farmer, and will not hatch while within the inventory. The Egg created by the Omen Sheep and Golden Beetle can be consumed by the Farmer, and provide unique effects.

REQ 1 : UML DIAGRAM



Design 1 : Creating a Reproduceable interface that will be implemented by NPC classes that are capable of reproducing.

The interface *Reproduceable* consists of the *reproduce* method, with the signature consisting of the *actor* object that will be producing an entity, the location that stores the *location* of the actor reproducing, and the *destination* that of where the new entity will spawn if its required to be spawned elsewhere from the Reproducing actor's location.

The usage of a "*Reproduceable*" Interface ensures that a contract is defined for NPCs that are capable of multiplying itself in unique ways. This also allows the game engine to interact with any NPCs that are "*Reproduceable*" in a consistent method

Advantages	Disadvantages
Common logic can be written to work with	The interface only enforces the method

any “*Reproduceable*” NPCs. Adhering to the Liskov Substitution Principle where the game logic can call the *reproduce* method on any *Reproduceable* NPC without knowing its concrete class.	signature, not the actual logic, so implementations may be inconsistent.
Utilising the *Reproduceable* interface allows extensibility, new NPCs that can reproduce can just implement and use the method, reducing code duplication. Adhering to the Open/Closed Principle where the *Reproduceable* interface is open for extension	Makes code slightly more complex as the *reproduceable* interface only consists of one method as of now.

Design 1.1. Implementing a Condition Interface, allowing the ability to introduce a condition classes that determines if certain methods can be performed according to the predefined conditions. (nearStatusCondition, spawnCondition, TurnCondition)

A Condition Interface was created as a contract for different Condition check Classes to adhere to. This Condition Interface consists of a isFulfilled method which returns a boolean value on whether or not specific conditions are satisfied to perform other actions.

The implementation of a nearStatusCondition provides a common method that allows any Actor to implement and check if any entities surrounding it hold specific Status Enumeration that may affect it. The Spirit Goat utilises this to satisfy its condition to reproduce, checking if the surrounding entities hold the BLESSED status, and then reproducing by spawning a new Spirit Goat around anywhere suitable through the SpawnCondition

The spawnCondition class checks if the given location allows an actor to stand on before spawning an Actor. If not, the surrounding location of the given location will be checked to spawn an actor onto the first valid ground.

The TurnCondition checks if a number of specific turns has been processed to fulfill specific turn-based conditions. The Omen Sheep and Golden Beetle then utilises this method

Advantages	Disadvantages
Allows extensibility, as new condition classes can be created without affecting the list of pre-existing conditions.	The overall code may be complex and difficult to trace and understand if error occurs.
New types of conditions can be created by implementing the Condition interface,	

allowing for easy expansion of game logic without modifying existing code.	

Design 2.0. : Creating an Egg class so that Omen Sheep and Golden Beetle are capable of laying an egg

The Egg class defines the attributes and properties of an Egg. It implements the tick() method that belongs to the Location class, allowing it to experience the passage of time through each tick. It also consists of the AllowableAction for the Farmer to perform actions on it.

Advantages	Disadvantages
The use of composition with EggHandler and Condition enables flexible behavior for different egg types and hatching requirements without code duplication.	
This design promotes code reuse, easy extension for new egg types, and clear separation of concerns, making the system maintainable and scalable.	

Design 2.1. : Creating an EggHandler Interface and EggSelector that allows the respective NPCs that laid the egg to determine how the Egg behaviours are executed

The EggHandler class is an interface implemented by NPCs which are capable of laying an egg. This puts the responsibilities of the egg object onto the class that produces it. Instead of creating respective egg classes for different eggs.

A few decisions were considered, such as creating additional egg classes that extend to the Egg class for each respective creature that can lay an Egg, or creating a hatch behaviour that determines how different eggs hatch. However, creating an EggHandler class was the preferred choice.

Advantages	Disadvantages
Code reusability and flexibility, the EggHandler interface allows any actor to define custom egg behaviors without duplicating code or creating a new egg subclass for each creature.	Tight coupling to NPCs, the handler is often an actor (e.g., OmenSheep), which can blur responsibilities and create tight coupling between the egg and its parent actor.
By delegating egg behavior to the	Testing egg behavior may require mocking or

EggHandler, the system avoids a rigid inheritance hierarchy (e.g., OmenSheepEgg extends Egg). Instead, egg behavior is composed at runtime, making the design more flexible and maintainable.	instantiating handler actors
The egg's behavior is not tied to a specific egg class (like OmenSheepEgg), but to the handler, making the system more modular and easier to test or modify.	
Adding new creatures with unique egg behaviors only requires implementing EggHandler, not creating a new subclass for each egg type. This reduces code duplication and maintenance effort.	

Design 3.0. : Implementing the Consumable Interface that creates a contract for entities that can be consumed by the player, paired with the ConsumeAction.

A new consumeAction was added to the list of actions, allowing the Player to perform an action that consumes the Egg when picked up. And a Consumeable interface was introduced for entities that can be consumed to implement Consumeable.

Advantages	Disadvantages
The Consumable interface abstracts the concept of eating the item itself, while ConsumeAction encapsulates the logic of performing the consumption. This keeps item logic and action logic decoupled.	Limited flexibility, If some consumables require more complex interactions, the interface may need to expand, risking interface bloat.
Provides extensibility, any entity can implement Consumable to provide custom consumption behavior.	
ConsumeAction can be reused for any item that implements	

Design Rationale : REQ 2

This design rationale was developed based on the material presented in the FIT2099 Assignment Rules (+ Survival Kit) link:
https://edstem.org/au/courses/20991/lessons/71748/slides/476741

A:Create the Golden Beetle

Design 1: Create the Golden Beetle class as an independent class without inheriting from any superclass.

Pros	Cons
The code is simple and straightforward without any inheritance	Code duplications if there are future creatures need to be created. playTurn() and allowableAction etc need to be reimplemented
Code flexibility provided as it can modify its own behaviour within a class.	Harder to maintain as any common behaviour between creatures made will need to modify every class.
No inheritance relationship needed to manage	Not scalable as creating new creatures will cause duplication of codes.

Design 2: Golden Beetle makes usage of the NPC Controller interface and extends from Actor class.

Pros	Cons
Provide code reusability, shared logic and codes are controlled by npc controller.	Code complexity increases as more classes are maintained.
Easier to maintain as any common changes made only need to modify the NPC Controller.	
Easier for different creatures in future to have unique behaviour without modifying	

existing code.	

<u>B: Make the Golden Beetle able to lay eggs, eaten by farmer.</u>

Design 1: Golden Beetle reuses Reproduceable, Consumable and EggHandler from REQ1

Pros	Cons
Clean separation of different logic	Code complexity increases and requires additional management.
Can easily make extensions for new logic. Adding new behaviours do not have to modify existing codes.	Hierarchy complexity increases
Promotes code reuse and utilises the given classes and interfaces in the game engine.	Changes made to classes or interfaces may result in modification of all its subclasses.

Design 2: Code all beetles consume, lay eggs and reproduce logic all inside the playTurn() method without using interfaces or behaviour classes.

Pros	Cons
Simple code setup without introducing many classes and interfaces.	Golden beetle class become bloated and complex.
Allow easy tracking of problems since codes and logic are located in one class	Low reusability as logic cannot be reused for other actors.
	Hard to extend new behaviours and logic.

<u>C: Prioritise the egg laying process by Golden beetle over follow behaviour.</u>

Design 1: Reuse the LayEggBehaviour from REQ1. Providing a higher priority than Follow behaviour.

Pros	Cons
Can modify the lay egg priority easily by utilising the game engine behaviour class.	Behavior conflict risk, must ensure lay egg behaviour comes before other behaviour else the creature will not lay egg.
Providing code reusability, other creatures which can lay eggs can reuse the behaviour.	

Design 2: Use a counter and decrease it inside the playTurn() method, once the counter <= 0 then golden beetles will lay eggs.

Pros	Cons
Do not require additional behaviour class, utilise the playTurn logic in engine to lay eggs.	Not scalable as other creatures have to repeat certain repeated code to achieve the egg laying logic.
Easier to debug as codes are all located in the playTurn() method.	Did not utilise the Behavior priority given by the game engine, there is no clean way to prioritize this action.

D: Define the hatching logic for golden beetle egg

Design: Reuse the NearStatusCondition from REQ1 to define that an egg can only hatch to golden beetle when cursed entities nearby it.

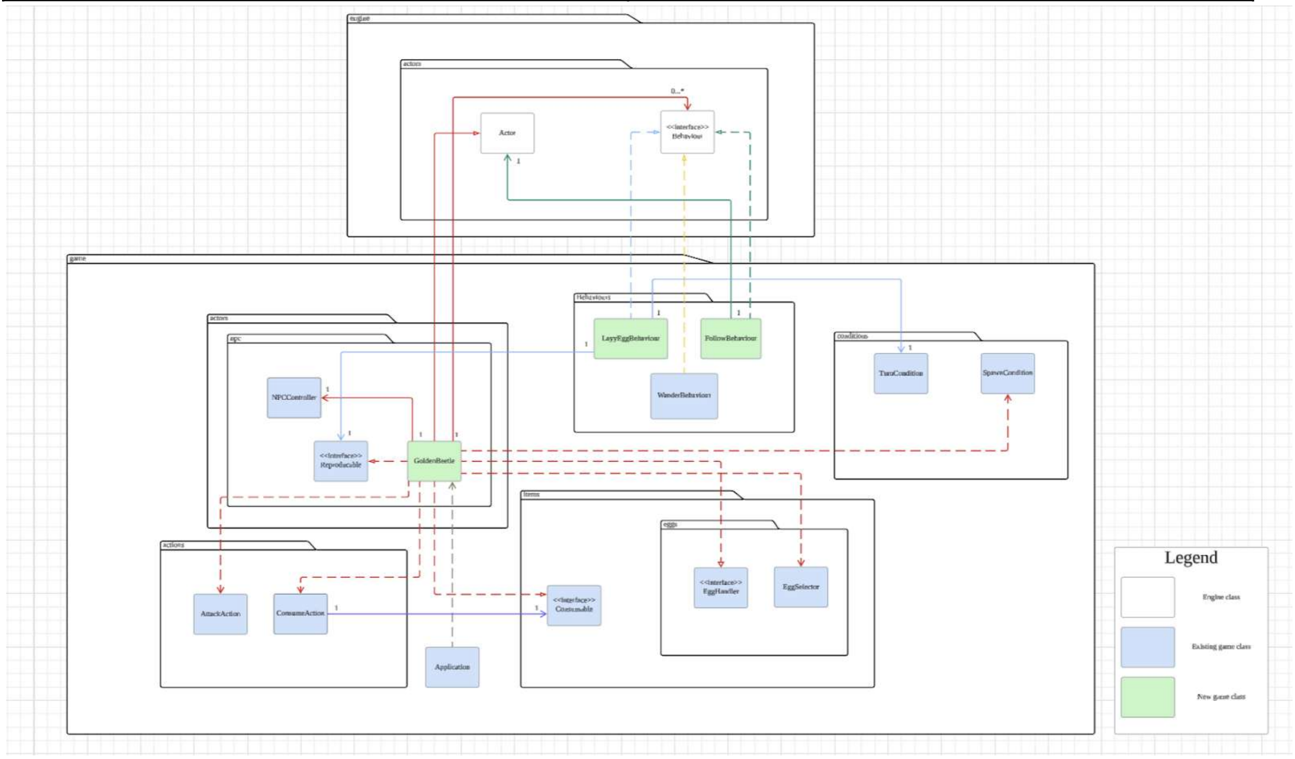
Pros	Cons
Reuse the code from requirement 1 stick to the DRY principle.	The clarity of egg hatching is not obvious, the reader has to track through the method and classes to understand.
Maintain consistent behaviour across the overall game that relies on its surroundings.	Increases the code complexity and more classes are added.
Provide easy adaptable Status can be changed to other status in the future not limited to CURSED only.	

E: Define the condition to hatch of the Egg layed by Golden Beetle

Design: Reuse the EggSelector class defined in requirement 1

Pros	Cons
Reuse the code from requirement 1 stick to the DRY principle.	The clarity of egg hatching is not obvious, the reader has to track through the method and classes to understand.
Easy to maintain as the logic of assigning hatch condition is centralized in a single class.	Increases the code complexity and more classes are added.
Provide scalability as it is easy to extend new egg types with new hatching	

conditions.	



Above is the uml of requirement 2 final result, which utilises Part A design 1, Part B design 1, Part C design 1, Part D and C design.

Golden Beetle extends the Actor class from engine and composes its functionality using external behaviour classes like FollowBehaviours and LayEggBehaviour which actually delegates logic to behaviours, reducing responsibility bloat in Golden Beetle class(Single Responsibility Principle SRP). Moreover, using interfaces and external behaviour classes, golden beetle functionality is open for extension but at the same time closed for modification(Open-Closed Principle). Next, the design of interfaces such that the subclasses which implements it can be substituted without error for example we can substitute an actor as Reproduceable since interfaces define clear contracts.(Liskov Substitution Principle). Behaviour logic like Follow and LayEgg Behaviour is encapsulated inside the external reusable behaviour class which reduces the duplication of code for other creatures that may reproduce or follow(Don't Repeat Yourself DRY Principle). For instance, if a new actor also needs to follow the player or lay eggs under certain conditions, the behaviour classes can be reused easily.

In Part C, egg-laying behavior is prioritized over the following behaviour by effectively re-using the LayEggBehaviour from REQ1 which avoids the hard coding logic directly in the playTurn() method to prioritise it over follow behaviour. Ensuring that the action is reliable and predictable, using SRP and OCP while also providing scalability.

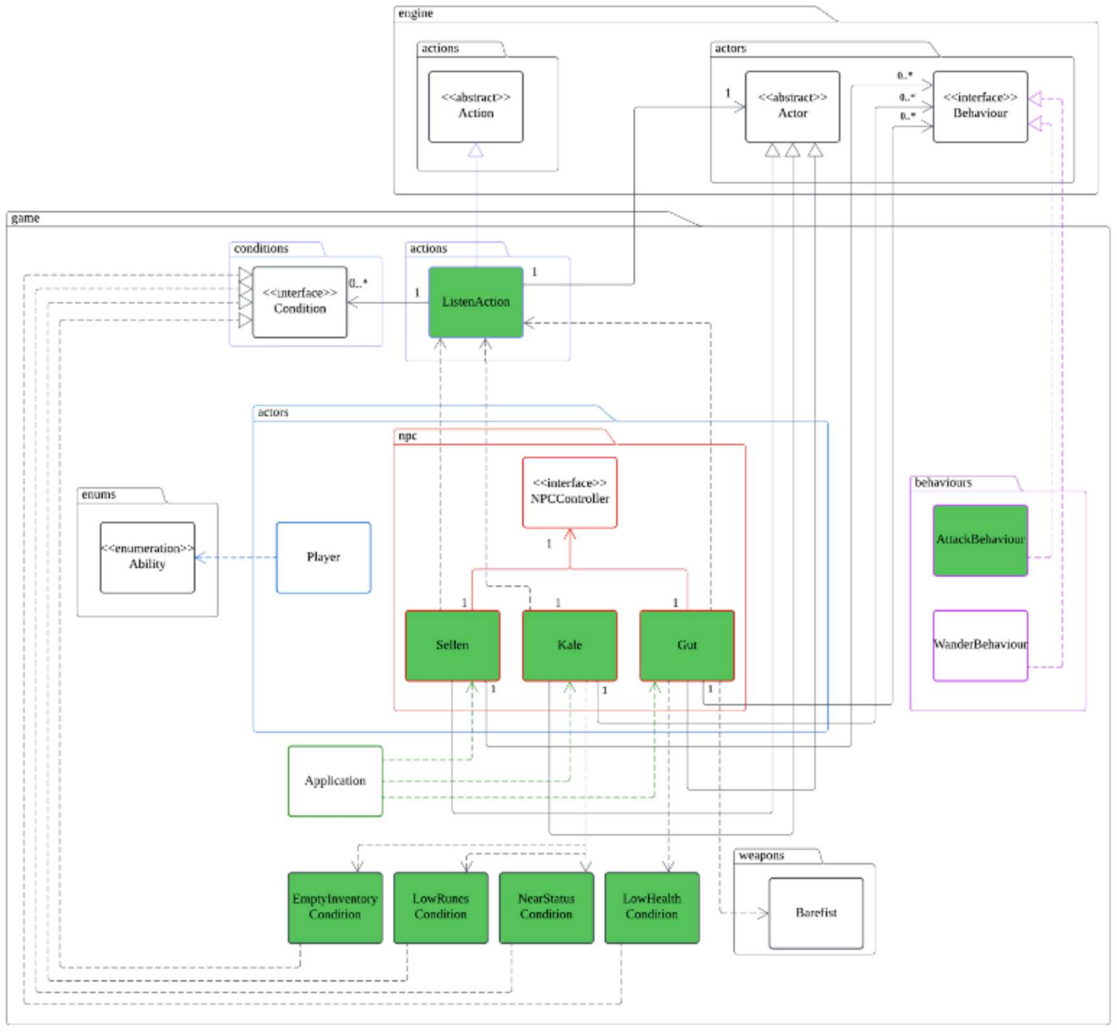
In Part D, the hatching logic for Golden Beetle eggs reuses the NearStatusCondition from REQ1. The egg will only hatch to Golden Beetle if cursed entities are nearby. (DRY), and also providing easy modification for different status (e.g. change from CURSED to other status)(OCP)

In Part E, the EggSelector from REQ1 is reused to define which creature will hatch and the condition which centralizes hatching logic, making it easy to maintain and scalable in the future.

The design is friendly for future extension and reduces code duplication.

Requirement 3: Design Rationale

UML diagram for requirement 3



Design Rationale for REQ3

Design 1: Implementing NPC classes using concrete classes as well as the listen function directly, without interface or abstract classes.

Pros	Cons
Simplicity as the design is straightforward as all behaviors are directly implemented in the NPC classes.	Code Duplication where the same logic for similar monologue condition are re-implemented, leading to redundancy if multiple entities share similar curing logic, violating DRY.
Less Complexity as there is no need for abstract classes or interface, making it easier to understand	Hard to extend as adding new NPCs with different conditional logic will need to modify many classes, violating Open-Closed Principle.
Direct control as each NPC has its own behavior and monologue conditions, allowing easy modifications to NPC's monologue pool	Tight coupling where each NPC class is tightly coupled with its own monologue behavior. Adding new features related to monologue need refactoring of other NPC classes.

Design 2: ListenAction handles the listening logic where conditional logics are handled as well.

Pros	Cons
Extensible as new NPCs that can be listened can be added easily using the getMonologue()or the getConditionalMonologues() method.	Complexity as the design introduces abstraction where new developers could take time to understand how the code works.
Allow adding newly listenable NPC class to be created easily without modifying existing classes.	Interface overhead as more classes and interface are created, where it will increase the number of files of the project.

The diagram above shows the final design of requirement 3, which utilizes design 2 instead of design 1 as stated above.

In this implementation, the system allows the “Farmer” to interact with three non-playable characters (NPCs) which are Sellen, Kale and Guts, using “ListenAction” class. Each NPC has a unique set of monologues, where it is output based on certain conditions, such as depending on the Farmer’s health, inventory or even NPC surroundings. The Farmer can be able to perform the “Listen” action if the NPC is within the surroundings of the farmer, then the monologue will be displayed randomly or on specific conditions. New conditional classes has been created where the NPC classes will have methods of getMonologues() or getConditionalMonologues() depending on conditional classes to allow NPCs to be listened. This design also allows easy extension, allowing the ease of adding new NPCs which can be listened.

The responsibility of defining monologues is moved into each NPC classes that can be listened allowing NPCs to focus on primary behaviors such as wandering or attacking which adheres to the single responsibility principle. The NPCs and their behaviors are opened for extensions but closed for modifications. This is where new conditions classes are created for displaying the monologues and NPCs or conditions can be added without modifying existing code. New conditions class are added where multiple npc class can output the monologues according to the conditional logic implemented. This can be seen in the example Kale class where it uses hashMap to store the priority, where it uses LowRunesConditions to check if farmer has lower than 500 runes then the monologue will be shown vice versa. This allows the design to adhere to the Open/closed principle where addition of new conditions without the need of modifying existing code. As a conclusion, design 2 provides a more flexible and extensible solution, which is made suited for larger games where new NPCs can be added following the OOP principles and promoted cleaner and more maintainable code.

Requirement 4: Design Rationale

A: Implementing weapon items inside the game.

Design 1:

- Implement an enumeration class called “WeaponItemEnum” which contains all kinds of weapon items for identification purposes.
- A concrete class called “WeaponItemFactory” creates a WeaponItem object for a specific weapon item.
- Contains a “WeaponItemFactory” for each merchant class that sells weapon items to create a weapon item.

Pros	Cons
- Prevents multiple layers of abstraction..	- Need to have type checking and downcasting before creating an AttackAction that includes the weapons, which are considered code smells.
- Easy to maintain and debug when all weapons have the same characteristics.	- Hard to maintain and debug when all weapons have different characteristics.

Design 2:

- Make WeaponItem an abstract class, and the concrete class representation of weapon items extends to the abstract class.

Pros	Cons
- Reduces code redundancy for all concrete classes representing	- Multiple layers of abstraction may tighten the coupling between the

weapon items, as they have similar characteristics.	parent and child classes.
- Easier to add new weapon item classes since it only extends the abstract WeaponItem class without any modifications.	- Hard to maintain and debug as it is hard to detect which abstraction layer is the problem.
- Easy to maintain and debug when many weapon items have different characteristics.	

B: Provide functionality that enables the Actor to buy a purchasable item and provides side effects on the purchasing Actor.

Design 1:

- Implement an interface that represents a particular merchant that sells a specific item, such as “BroadswordSeller”.
- For each merchant or seller interface, there is a method called “sell(PurchasableItemName)” which checks whether the Actor has enough Runes to buy a purchasable item. It also applies the effects of selling a purchasable item from a specific merchant.
- Implement a purchasing Action subclass for each purchasable item, such as “BuyBroadswordAction”. The “execute” method inside the class applies the purchase effect to the buying Actor from the item.

Pros	Cons
- Provides a separation of concerns for each purchasable item and the effects of buying a purchasable item from the merchant.	- A buy Action class and a seller interface need to be implemented for each weapon item, which increases the code complexity.
	- Increases code redundancy if the side effects of a purchase are similar to all purchasable

	items.
	- Does not provide code reusability if the effects of purchasing a weapon item are similar to another.

Design 2:

- Implement a Purchasable interface that represents purchasable items.
- Implement a SideEffect interface that represents all side effects.
- Implement an Action subclass called PurchaseAction, which processes purchase actions.
- The side effects of buying a purchasable item will be shown before the result of the action.
- The side effects of purchasing it, regardless of the merchant, are implemented in each Purchasable item class.
- A method is created for each purchasable item sold by them to implement the effects of selling it by the merchant to the buyer for each merchant class.

Pros	Cons
- Allowing new purchasable items to use existing effects enables code reusability.	- Will increase code complexity if there are a few purchasable items in the game.
- The SideEffect interface allows for representing all kinds of side effects.	
- The side effect classes implementing the SideEffect interface can be applied to other scenarios, enhancing reusability and also encapsulating each side effect into its own class, which reduces connanscence.	

C: Allows the Actor that is hostile to an enemy to use the weapon items from their inventory to attack other actors.

Design 1:

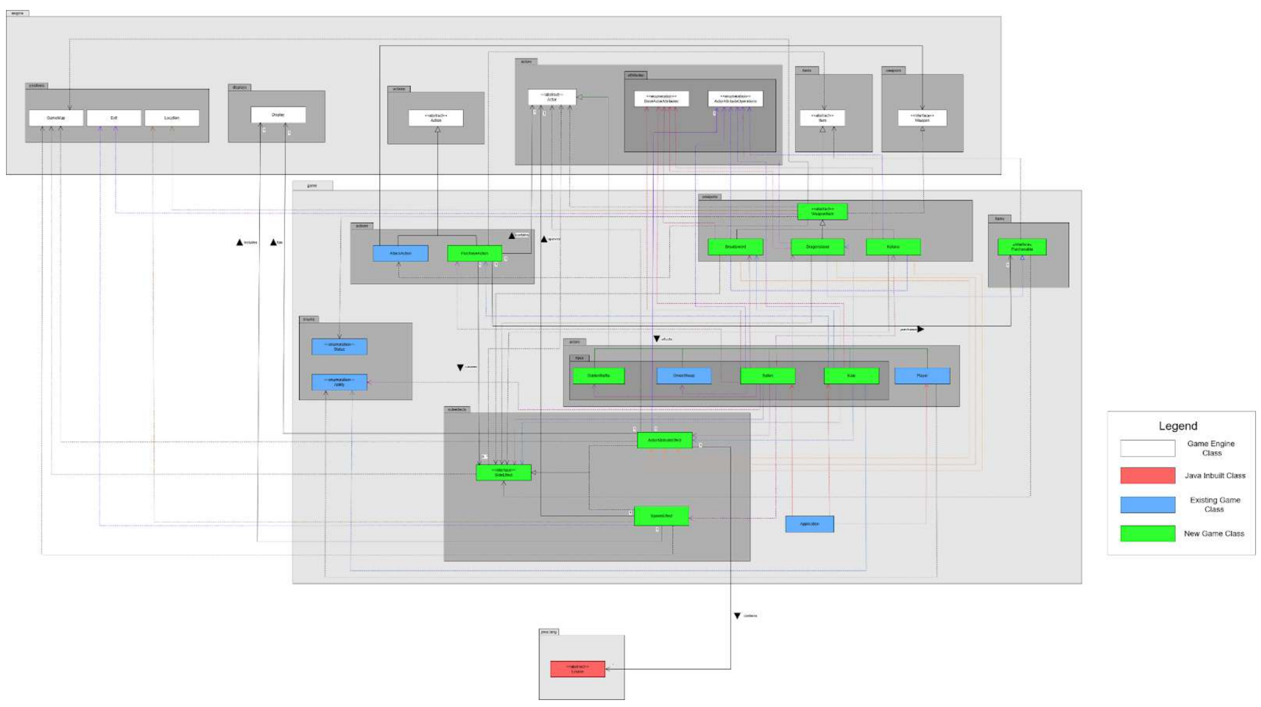
- Create a class called AttackOption, which checks whether a weapon exists in their item inventory to obtain its attack option.
- The "getAttackOptions()" is called statically inside the "allowableActions" method for each NPC to obtain all possible attack actions.

Pros	Cons
- Reduces redundant code for all NPC classes for obtaining all possible attack actions on them.	- Introduces procedural or functional programming practices.
- Easy to extend and maintain when there is a new weapon item is introduced inside the game.	- Type checking is required to determine what kind of weapon item it is from the inventory, which is one of the code smells.
	- Downcasting is needed to convert the Item data type to the Weapon data type for the AttackAction.

Design 2:

- Implement and override the “allowableActions” method inside the WeaponItem abstract class.
- Inside the “allowableActions” method, the location of a weapon owner is obtained from the GameMap. A for loop traverses the surrounding tiles to obtain attack options on surrounding actors.

Pros	Cons
- Eliminates type checking and downcasting through inheritance to obtain all possible attack actions for each weapon item.	- The “allowableActions” method implementation is used to give a list of actions that can be done on its owner and not on other actors.
- Easy to extend and maintain when a new weapon item is introduced inside the game.	



The diagram above shows the final design of requirement 4, which utilises each Design 2 of parts A, B, and C stated above to implement weapon items, purchase actions for purchasable items and attack options for each weapon item.

The abstract WeaponItem class is created, and weapon items in requirement 4 are extended to the abstract class. Since all weapon items share some characteristics and behaviours, an abstract class is created to have the same attributes and behaviours across all weapon items contained and abstracted to avoid code repetition (DRY).

The Purchasable interface has been created and implemented across all purchasable items. It allows external classes a dependency relationship on the purchasable items that will depend on the interface instead of a concrete class representation of purchasable items, reducing the number of dependencies for the external class (DIP and ReD).

The ActorAttributeEffect and SpawnEffect classes have been created such that these effects are reusable by other scenarios, reducing redundant code (DRY). Also, it encapsulates implementation details of each side effect in its concrete class, providing separation of concerns, which reduces connascence to other classes and increases the connascene locality in each side effect to provide maintainability and ease of debugging. (990 Words)