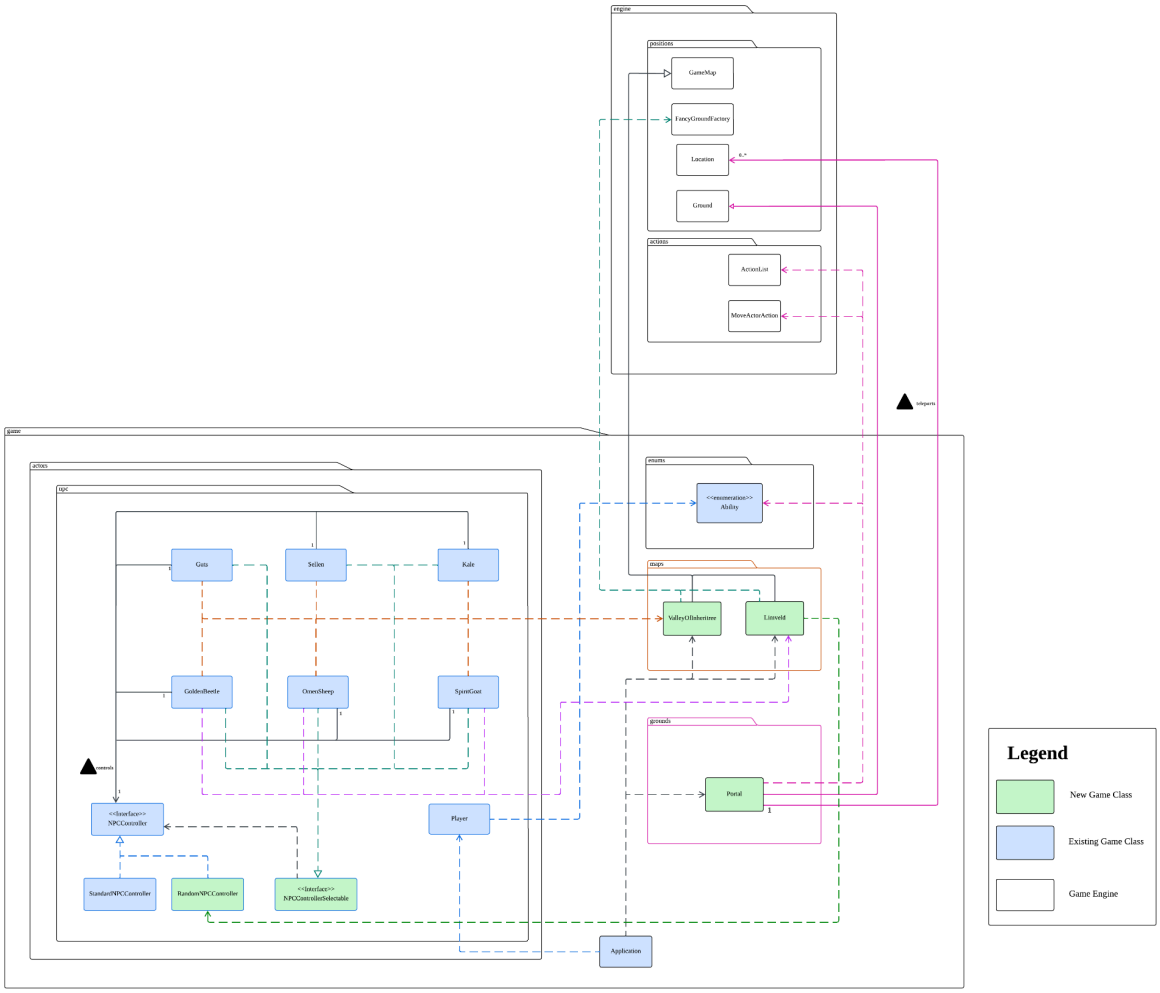
FIT 2099: Assignment 3 Report

MA_Applied05_GroupC

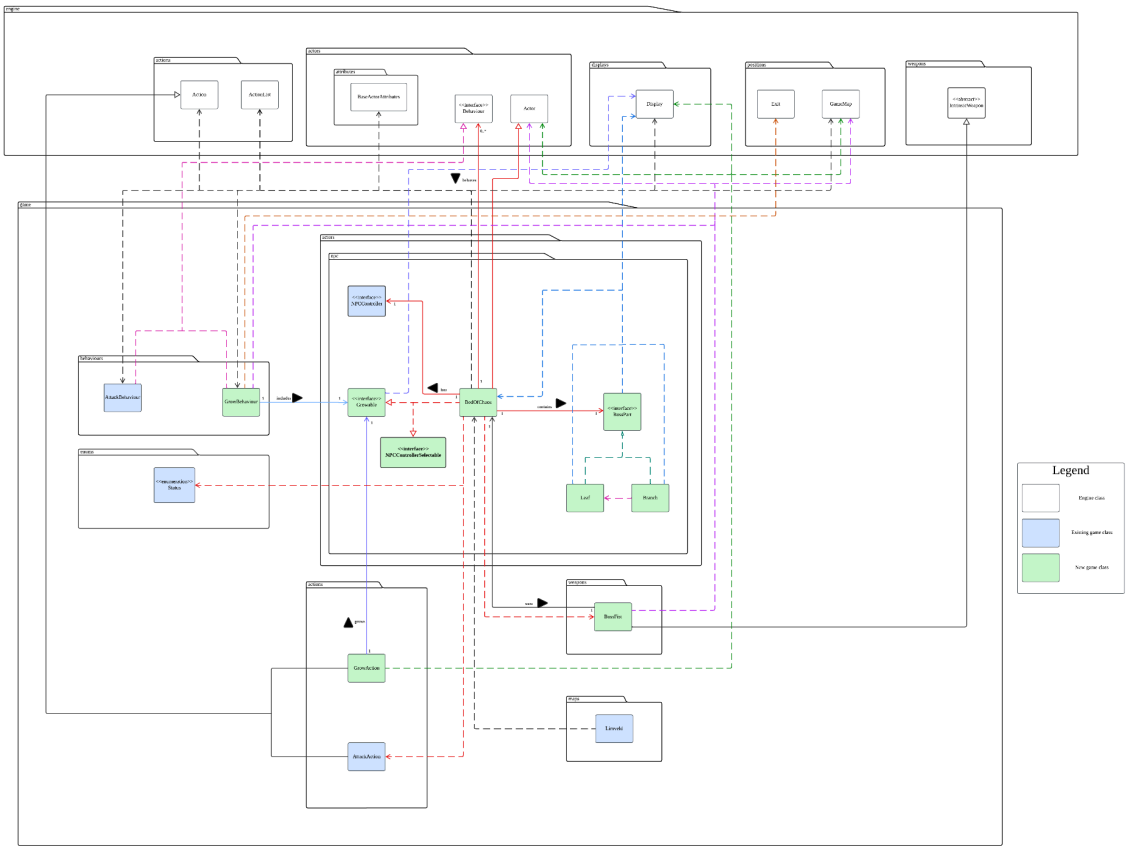
Team Members:
Tee Zhi Hong (34570403)
Joel Wong Sing Yue (35478527)
Nigel Wong Wei Lun (33524904)
Liew Yi Wei (34146385)
Abdullah Saad (33023239)

UML Diagrams

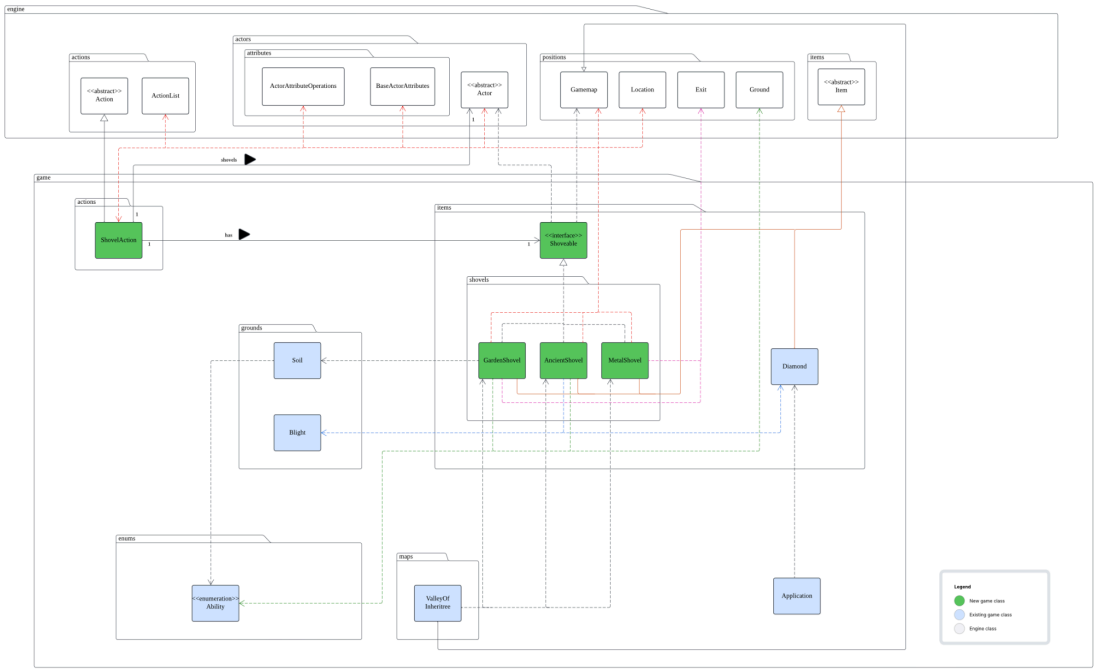
Requirement 1



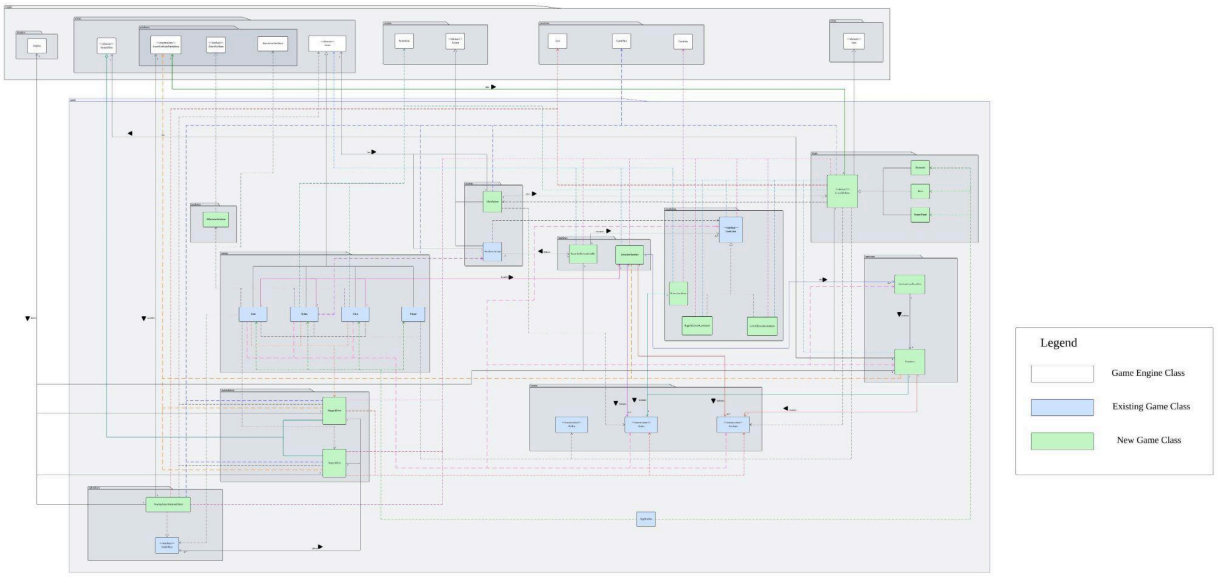
Requirement 2



Requirement 3



Requirement 4



(The Requirement 4 UML Diagram is blurred due to its large size, please refer to the "req4-uml.pdf" for more details.)

Assignment 3 Requirement 1 Design Rationale

A: Create a map called “Limeveld”

Part 1:

- Initialise a GameMap variable and use it for the Limeveld map directly inside the Application class

Pros	Cons
Easy implementation with simple logic.	The Application class has multiple responsibilities or concerns, causing it to become a “God” class (SRP violation).
Easy to maintain and debug when there are a few maps existing inside the game.	It is challenging to add new types of maps within the game without modifying the external class, which is the Application class.
	It is also challenging to maintain and debug as the number of maps available within the game increases.
	It increases the code length of the Application class which is a code smell (“Bloaters”).

Part 2:

- Create 2 GameMap subclasses, which are “ValleyOfInheritree” and “Limeveld”, that extend the GameMap class
- All entities (except the Portal class) will be initialised inside their corresponding game map class through a method called “setUp”, which is called inside the “Application” class

Pros	Cons
Provides separation of concerns between the Application class and the game map initialisation.	Will tighten the coupling between the GameMap class and its subclasses so it may become harder to maintain.
Easy to extend when a new map is introduced inside the game by extending to GameMap class.	The GameMap is a concrete class, so it may violate the LSP and have unexpected behaviour.
Easy to maintain and debug errors that are related to the game map, even with several different maps.	
Utilises the game engine since it uses GameMap from the game engine class, which contains some similar characteristics, will be contained and abstracted to reduce code redundancy.	
Increases code readability.	

B: Implement a teleportation circle that allows the Player to travel

Part 1:

- Implement a class called “Portal”, which is a class representing a teleportation circle.
- Each GameMap subclass will have its own Portal attribute and be initialised inside the “setUp” method.
- The portal location will be stored as an attribute inside each GameMap subclass.

Pros	Cons
Separates the concerns between the teleportation circle and the application class.	The GameMap sub-classes will have multiple responsibilities, which increases code complexity.
	If there is a modification related to the teleportation circle, need to adapt changes across all GameMap subclasses, which is tedious if there is a large amount of it.
	Causing multiple dependencies inside each GameMap subclass.

Part 2:

- Implement a class called “Portal”, which is a class representing a teleportation circle.
- The Portal class will have an attribute that stores a list of locations where the Portal is located.
- A Portal object is initialised inside the Application class.
- The same portal object is added across multiple locations inside Valley of Inheritree and Limveld and these locations are added inside the Portal object inside the Application class.
- The “allowableActions” method inside the Portal class will provide possible destinations except the current location.
- The portal location will be stored as an attribute inside each GameMap subclass.

Pros	Cons
The implementation logic will be contained in a single class instead of other external classes, so it provides a separation of concerns.	The Application class would still have multiple responsibilities, which increases code complexity.
Reduce the number of dependency relationships inside the game design.	
Easy to maintain and debug.	
Provides code reusability as a Portal object can be added to several locations	
Easy to extend	

C: Provide a selection of behaviour modes (priority or random mode) for all NPCs that can do so during the initialisation.

Part 1:

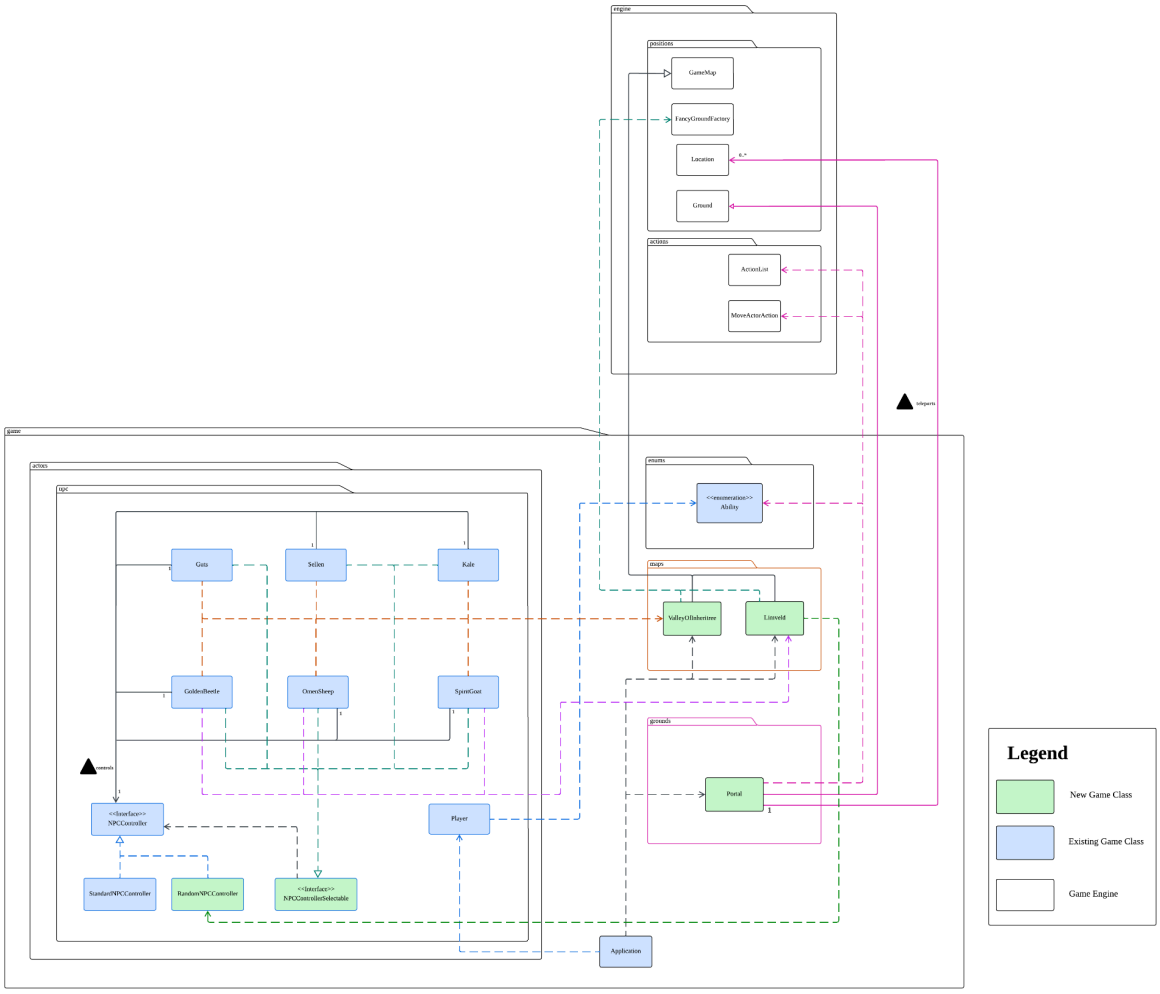
- Add an NPCController parameter inside the class constructor for each NPC that can provide a selection of behaviour modes.

Pros	Cons
Provides dependency injection with the NPCController class object.	Need to modify each external class that uses these NPC creatures, which is tedious when the number of these classes increases.
	It is challenging to maintain and scale as the number of classes increases.

Part 2:

- Create an interface called NPCControllerSelectable, which represents an NPC whose behaviour selection mode (priority or random) can be chosen during its initialisation.
- An interface method called “selectNPCController”, which sets the behaviour selection mode (NPC Controller) during the initialisation.

Pros	Cons
Allows these NPCs to have their behaviour selection mode chosen differently during the initialisation without modifying the external classes that use them.	Cause connascence and it causes an error if the attribute for NPCController is null. However, the attribute will be set to priority by default to prevent errors.
Provides code scalability and maintainability.	
Allows new NPC classes to extend the feature easily.	



The diagram above shows the final design of requirement 1, which utilises each design 2 in parts A, B, and C stated above to implement the Limveld map, teleportation cycle and behaviour selection mode for creatures of Limveld.

The RandomNPCController implements the NPCController interface, which allows external classes that use the RandomNPCController to depend on the interface instead of the concrete class (DIP).
In addition to that, an interface called NPCControllerSelectable is implemented to represent the NPCs that can select the behaviour mode during their initialisation. Currently, all NPCs implement the interface so

their behaviour mode or NPCController can be different from each other during their initialisation without modifying the external classes that use it, although it might increase the connascence of these classes, which is outweighed by its benefits(OCP).

Furthermore, 2 classes are created to represent ValleyOfInheritance and Limveld and extend to the GameMap class, as they have similar characteristics to the GameMap class. The benefit of this is to have a separation of concerns between the Application and game map implementation (SRP). Also, it reduces the code length of the Application as high code length is a code smell (“Bloaters”).

Assignment 3 Requirement 2 Design Rationale

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 boss Bed Of Chaos

Design 1: Create the boss as an independent class without inheriting from any superclass and separate from the provided game engine.

Pros	Cons
The code is relatively simple and all the boss attributes and behaviour is all implemented in the same class.	Code duplications if there are more creatures are created in the future. Same code logic between creatures need to be reimplemented.
Provides code flexibility as all code can be maintain in same class	Harder to maintain as one common change between creatures will need to modify every single class.
Do not need to maintain any complex inheritance relationship	

Design 2: Bed Of Chaos makes use of the Actor class from the engine and NPC Controller.

Pros	Cons
Providing code reusability as common behaviours are maintained by NPC Controller.	Code complexity increases as more classes need to be maintained.
Provide better maintainability, common changes to creatures only have to modify the NPC Controller class.	Inheritance relationships need to be maintained well.
Different creatures can have unique behaviour without modifying existing code.	

B: Implement the growing part of the boss, Leaf and Branch

Design 1: Create two different classes Leaf and Branch separately and implement the effect and function it brings to the boss.

Pros	Cons
Simpler and easier to implement the code	Code duplication as it do not have a common clear structure for common methods
Fewer class reference and relationship to manage	Do not have polymorphism as we cannot treat Leaf and Branch as the same type.
	Tightly coupled logic as boss has to know all the methods in both class (method name)

Design 2: Create two different classes Leaf and Branch separately but use an interface BossPart to have a clear contract on what they can benefit to the boss.

Pros	Cons
Code reusability as shared common structure between methods only makes logic different.	Slightly more complex code as interface need to be maintained
Easier for future extension for example new parts like flowers or fruits can be extended.	Over engineer if only small usage.
	Possible to have unused methods if future boss parts do not need some specific logic.

C: Implement the growing of branches and leaves for the boss.

Design 1: Code all the growing logic inside the boss playTurn() method.

Pros	Cons
Very quick and simple to implement the logic as all code is within a method.	The grow logic is tightly coupled with the game logic as playTurn() handles too much.
Do not require any extra classes.	Hard to extend as other or future creatures cannot reuse the growth logic.
Have direct control over the boss's behaviour.	

Design 2: Create a growBehaviour and growAction to handle the growing logic.

Pros	Cons
The grow logic is separated from the boss, the boss does not have to handle the grow logic.	Increases the code complexity as more classes need to be maintained.
Highly reusable design as other creatures can reuse growAction and growBehaviour.	
Easier to extend	

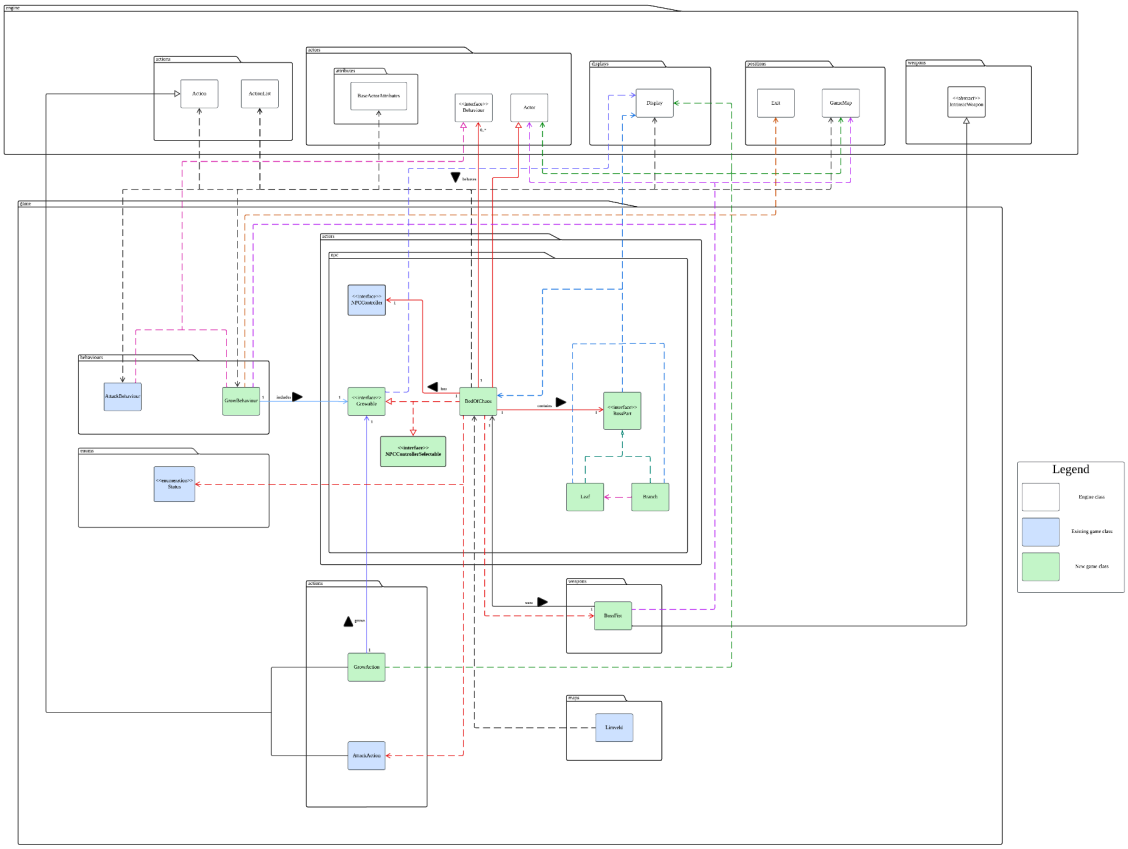
D: Implement the calculation of damage by the boss.

Design 1: Let the boss use default intrinsic weapon, use a variable to calculate the boss damage each round and reset the intrinsic weapon damage.

Pros	Cons
Simple and easy to implement as no extra classes needed.	Boss directly handles the calculation of damage which violates the encapsulation.
Easy to track the calculation of damage.	Repetitive step as the boss has to reset the damage every round.
Flexible for small temporary calculation of damage.	Not reusable code.

Design 2: Create a new BossFist weapon which will calculate the damage itself.

Pros	Cons
The damage calculation is encapsulated and not reachable by the boss.	More complex code setup as more classes were introduced.
Reusable code as the weapon can be shared or extend	
Easy to maintain and modify the calculation of damage.	



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

The design of the branch and leaf is using a recursion design. Each branch has an arrayList to store all the child branches and leaves it has for the calculation of damage.
The growth of the branch is designed using a growBehaviour to prioritise attack action over the growAction if the player is nearby.

The design also has a connascence of type, interfaces like Growable and BossPart allowing classes like GrowAction, GrowBehaviour and BedOfChaos to depend only on the abstraction level, enable us to have more growable entities in the future without changing the existing logic code.

The Bed Of Chaos class was implemented using the provided Actor class from the engine and the NPC Controller from previous code. This adheres the Open-Closed principle as it utilises the existing code without modifying it and also shared behaviour is maintained by the NPC Controller. The leaf and branch part of the boss were implemented using an interface called BossPart. The design follows the Liskov substitution principle by enabling the boss to treat the leaf and the branch as a BossPart polymorphically and also declare a clear contract for the parts. Moreover, the growing logic was encapsulated from the boss using growAction and growBehaviour. The design adheres to Single Responsibility Principle as the boss does not have to handle the growing logic. Providing a better readability and maintainability. Finally the damage calculation of the boss is encapsulated in the BossFist class which supports the Single Responsibility Principle, the boss does not need to handle the calculation of the damage.

The design of Bed Of Chaos applies multiple SOLID principles to ensure the maintainability and extensibility. It is a friendly design for future extension and reduces code duplication (DRY).

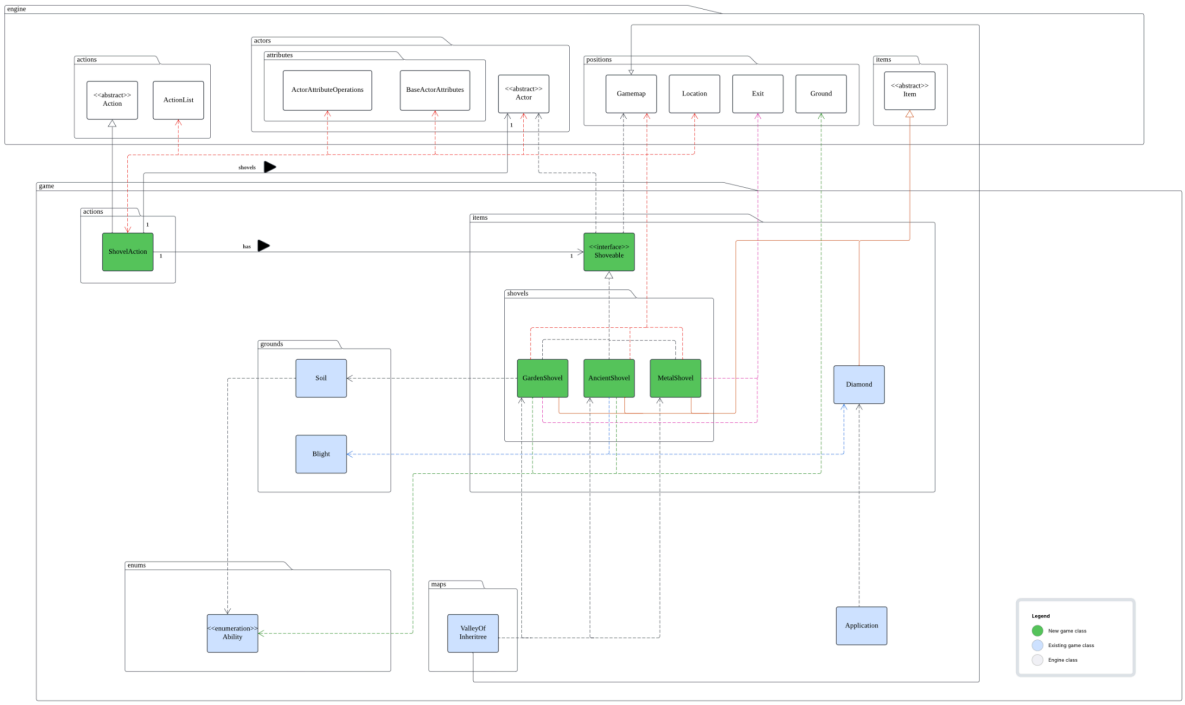
Assignment 3 Requirement 3 Design Rationale

Design 1: We introduce the Shoveable interface, where it consists of the method that each shovel class should have. Newly introduced shovels can implement this interface. Additionally, a higher-level class such as Shovel Action associates with the Shoveable interface, rather than directly with concrete shovel classes.

Pros	Cons
Flexibility: By defining the new interface, new types of shovels can be added easily without modifying existing code.	Complexity: The introduction of interfaces classes adds complexity especially in smaller projects where such abstraction may not be needed.
Code Reusability: Common methods are abstracted into higher-level actions where the ShovelAction class can work with any class implementing the Shoveable interface.	Learning Curve: New developers may find the approach harder to understand initially, especially when abstract concepts such as polymorphism and interfaces are used extensively.
Loose coupling: the interactions between classes are based on interfaces, reducing dependencies between classes. Allow changes in one class without affecting others.	

Design 2: In this approach, we create concrete classes directly for specific tasks without defining a common interface. Each shovel class performs its specific functionality directly, and other classes interact with them by directly referencing the concrete implementations.

Pros	Cons
Simplicity: Directly implementing the methods without abstraction of interfaces or inheritance makes the code more straightforward.	Lack of Flexibility as the design becomes tightly coupled to the specific implementations of each shovel, making it harder to extend or modify in the future.
Less overhead: Fewer classes and interfaces to manage, making the design potentially simpler for smaller systems.	Scalability issues as the game grows and more shovel types are introduced, this will result in bloated classes and difficulty adding new shovels.
Quick implementation because it is simple and straightforward, using concrete classes can lead to faster implementation.	



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

The design with interfaces and abstraction generally provides better long-term benefits, especially in larger systems where flexibility, scalability and maintenance are important. I have introduced the new ShovelAction class where it has the actions of shoveling an item or ground. This class interacts with the Shoveable interface to carry out the shovel action. It executes the shovel functionality by calling the shovel() method.

This design also exhibits various types of connascence, where it enhances its modularity and maintainability. Specifically, there is a connascence of name between the ShovelAction class and the Shoveable interface as the shovel() method must be consistently named across all implementing shovel classes. The design also demonstrates connascence

of type, with ShovelAction depending on objects that implement the Shoveable interface.

This design also follows the open/closed principle where by defining the Shoveable interface, my design is open to adding new types of shovel, where a new class is needed that implements the shoveable interface and add its specific behavior in the shovel() method. Without the need of modifying existing classes such as the Ancient, Garden, or Metal shovel classes. ShovelAction class on the other hand is closed for modification because its logic is enough for any object implementing the Shoveable interface where adding new shovels does not require changing ShovelAction class. The shovels introduced all inherit from the Item class, where they share the basic properties of name, display character, and whether the item is portable. This follows the inheritance principle where it can reduce redundancy, as the properties of shovel items are common across all types. Also, the use of interfaces and higher-level classes ensure that the design remains flexible and maintainable in the long term, making it easier to extend the game with new shovel items or actions while preserving clean and modular code.

Assignment 3 Requirement 4 Design Rationale

Note: Enumeration constants from the Status class are used to identify the actors who provide affection, affectionate actors who are capable of receiving items. Magic numbers and booleans can be used, but they complicate code complexity. No need to add an enumeration constant for affectionate actors because it causes overengineering. Also, it can be checked by looking presence of the affection level attribute. Furthermore, no need to check whether the actor is affectionate for GiveAction since the enumeration constant for receiving items is only for affectionate actors.

A: Set appropriate emotions after receiving affection

Design 1:

- Check whether the affectionate actor has received affection inside the “playTurn” method.
- If present, set the emotions based on the current affection level directly inside the “playTurn” method for each affectionate actor class.

Pros	Cons
Easy to implement with simple logic.	Can cause the “playTurn” method to have multiple responsibilities.
	Harder to maintain or debug.

Design 2:

- Implement a class called EmotionHandler which handles the logic behind setting emotions after receiving affection.
- Include the EmotionHandler as an attribute for each affectionate actor class.
- Check whether the affectionate actor has received affection inside the “playTurn” method and set emotions if it is true.

Pros	Cons
Provides a clear separation of concerns between the original responsibility of the “playTurn” method and setting up emotions after receiving affection.	It might cause overengineering if there are a few affectionate actors in the game design.
Easy to maintain and debug if there is an issue in setting up emotions.	
Code modifications related to handling emotions can be made in a single class instead of multiple classes.	

Note: Since the “tick” method is not present in the Actor abstract class, it is placed inside the “playTurn” method as it behaves similarly to the “tick” method.

B: Set emotions correctly for each affectionate actor after receiving affection.

A class called Emotion is created to store the emotion details of an affectionate actor and set the emotion inside them if its condition has been fulfilled. The implementation logic behind emotion can be implemented inside each affectionate actor, but needs to be refactored all when there is a small change inside the implementation, which is a code smell (“Shotgun Surgery:)

A TreeMap is used to ensure that available emotions for the actor are in the correct priority and order. Since TreeMap only stores 2 objects per entry, a new class called EmotionConditionPair is used to combine the emotion and its condition as a pair, and it is used as a value for the TreeMap alongside the priority key.

Alternatively, it is possible to implement the EmotionCondition interface that extends the Condition interface, which sets the emotion inside its subclass if the condition has been fulfilled. However, it interferes with the original responsibility of the conditional subclass and needs to do a “shotgun surgery” for all EmotionCondition subclasses if any changes are made in setting up emotions, which is a sign of code smell.

The affectionate actor classes will initialise all attributes from the Emotion class since emotions vary for each affectionate actor. It is possible to make the Emotion class an abstract class and have a representation of emotions extending the abstract class to provide more scalability. However, it causes overengineering and increases code complexity as it currently has fewer emotions and also increases coupling to its subclasses, so the implementation is infeasible.

C: Implement the emotional effects of the affectionate actor when receiving affection for several game turns

Create a status effect class for each emotion and provide an attribute for a list of side effects inside them. These classes fully utilise the game engine and fulfil a clear separation of concerns between the emotional effects and the actor class. It also enhances code reusability.

Alternatively, implement the emotional effects directly inside the “playTurn” method for each affectionate actor, but it can cause multiple responsibilities for the method and does not fully utilise the game engine provided. Also, it creates a “bloater” (long method), which is a sign of a code smell.

D: Implement the affection level attribute for each affectionate actor

Create an attribute class called AffectionAttribute, which extends the ActorAttribute interface that is used for the affection level attribute, as the alternative BaseActorAttribute is not suitable for affection level, as it is automatically initialised to maximum level. The mentioned attribute can be updated to adjust to the initial level, but it causes connascence of execution.

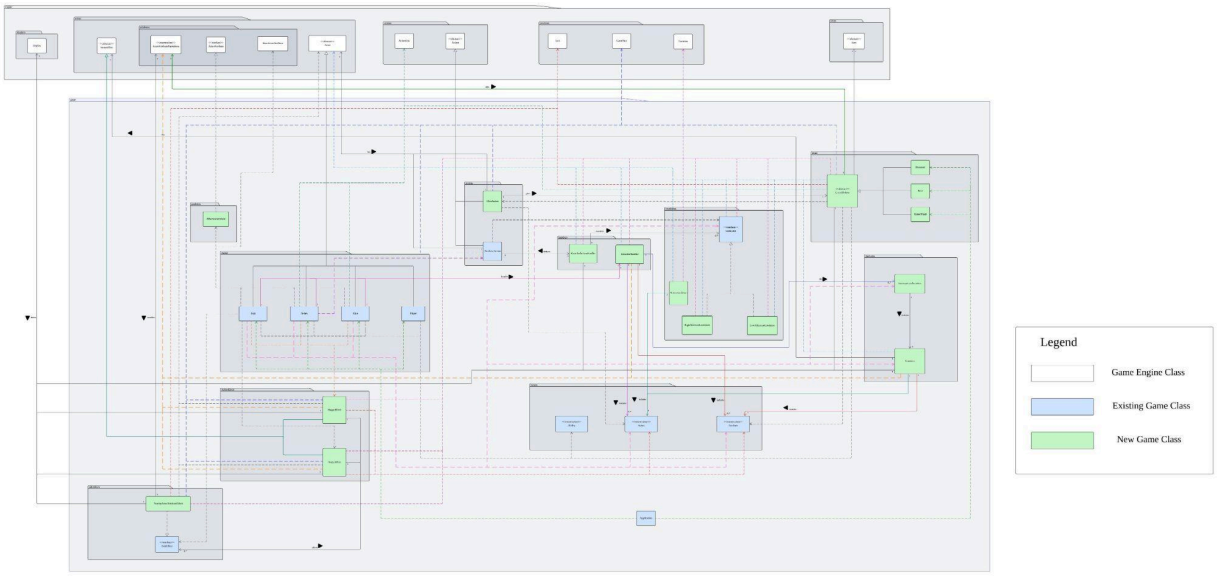
The initial, minimum and maximum affection levels are contained inside the class to reduce code redundancy across affectionate actors (DRY).

E: Implement price deduction for merchant classes when they are happy.

Implement a class called RuneDeductionHandler which handles all Rune deductions based on their corresponding conditions. It enables code reusability, particularly in future scenarios where price deductions are applied.

The “handleRuneDeduction” method from RuneDeductionHandler is placed inside the “execute” method in the PurchaseAction class to obtain the final price after deduction, to enhance the separation of concerns between the rune deduction and the “execute” method, which provides code scalability and modularity.

Alternatively, the implementation logic behind rune deduction can be placed directly in the “execute” method, but it causes the method to have multiple responsibilities.



(The Requirement 4 UML Diagram is blurred due to its large size, please refer to the "req4-uml.pdf" for more details.)

The diagram above shows the final design of requirement 4, which utilises each design in parts A, B, D, E, F, and G and also Design 2 in part C stated above to implement the affection level system, which includes "giveable" items and the emotional effects of the affectionate actors.

Status enumeration constants are used to identify actors that provide affection, and affectionate actors that can receive items, which reduces connascence (CoC) and allows code maintainability and extensibility.

A class called Emotion is used to set the emotion of an affectionate actor based on the affection level. Moreover, A class called EmotionHandler is created to set the correct emotion based on its conditions. This provides a clear separation of concerns between the affectionate actor classes and handling emotions (SRP), and reduces code smells for these

classes. Also, it reduces the number of dependencies for Actor classes since the implementation is not placed in them (ReD).

Emotion status effect classes are created to allow affectionate actors to have emotions for certain game turns, and they fully utilise the game engine and enhance code reusability (DRY). It can also be upcast and execute a method from its superclass (LSP).