Requirement 1: The Ancient Woods

New Enemy Classes

By choosing to implement both new enemies: Red Wolf and Forest Keeper, as child classes extending from the abstract *Enemy* class, this ensured the Open/Closed Principle was followed. Some modifications needed to be made at the child class and parent class levels, however, these were small and could be used for future child classes.

As both enemies live in the Ancient Woods and any resident of the Ancient Woods can follow any actor hostile to them, a Status was added, “RESIDENT_ANCIENT_WOODS”. This could easily be extended from in future for new classes which require special behaviours and are from the ancient woods. Additionally, this meant no violation of the Do not Repeat Yourself (DRY) principle would occur here. Instead, this functionality could be implemented in the *Enemy* class.

A check was added in the findAction method, located in the *Enemy* class, whereby if the enemy had the status “RESIDENT_ANCIENT_WOODS”, then, an instance of FollowBehaviour would be added to their behaviour HashMap. By referencing the Behaviour interface from the *Enemy* class, this also provided another layer of abstraction, by way of the Dependency Inversion Principle, which means less effort would be required to modify the system in the future. *Enemy* does not rely on FollowBehaviour but instead the Behaviour interface to get the action for the enemy to execute. A change to the Assignment 1 implementation was made here, adding possible behaviours in the findAction method, rather than during instantiation. This decision was made so the FollowBehaviour could be the first behaviour added to the HashMap (if applicable). This helped ensure FollowBehaviour would be checked to occur before the possibility of attacking or wandering.

An advantage of this implementation was that an enemy from the Ancient Woods would prioritise following actors which are hostile to them over attacking, ensuring that, where possible, they would still be following the actor. A disadvantage however, is that the code may be more difficult for a new developer to understand when starting the project. This is because the behaviours are no longer being added in the constructor method, where it was very clear that the enemy would be able to execute one of the behaviours given. Instead, as the behaviour HashMap is refreshed on each turn, some confusion may arise from whether the available behaviours may change between turns.

However, this alternative implementation would be flawed, as the status “RESIDENT_ANCIENT_WOODS” cannot be added until an instance of the concrete class has been created, which does not occur until the constructor method in the *Enemy* class has been called. Therefore, a check to see if the status existed, would always return false and FollowBehaviour would never be able to be added.

Follow Behaviour

Implementing Follow Behaviour involved three distinct scenarios. To improve readability and the ability to understand the code, a new function was added for each scenario, with a main

function added to run each of these, in sequence. These could have been left as one major function, however, it would be difficult to quickly determine what was happening in the function. Although splitting these up reduced brevity, they also decreased the time a future developer would require to understand the functions. The first two functions returned booleans to indicate if the main function should traverse the proceeding functions. No actions would need to be returned, as these functions do not require the actor to move or perform any other action.

The first function checks if the actor, who may follow a hostile actor, is not yet doing so. If, in the actor's surroundings, is another actor, who possesses the status "HOSTILE_TO_ENEMY", this actor will be set to TARGET, a static variable. As the target could not be passed in as a parameter when creating a FollowBehaviour, making the found target a static variable, would ensure any future references to TARGET in the FollowBehaviour class in future, would reference the same object.

Alternatively, if the player had been passed in, this would likely increase the complexity of the system, making it difficult to understand the system. This would be because a reference to a specific target would need to be passed in, likely requiring a reference to that target throughout the entire system. This implementation is disadvantageous, as it would violate the Open/Closed Principle, with the system unable to take any other possible targets, without modification to existing classes.

This first function would return false if the actor was already following their target, allowing the second function, which checks whether the target is within the actor's surroundings, to be called. The surroundings of the actor are checked for the TARGET, returning true if so, as this means the actor cannot move any closer to the TARGET and are free to execute a different behaviour.

Otherwise, the final function would be called, checking to see if any MoveAction within the actor's surroundings would bring them closer to the TARGET, in which case, this action would be called.

This interaction is represented in the sequence diagram for this requirement, chosen as it is the most complex interaction which was implemented for Requirement 1. It is represented with the RedWolf class, however, the same interaction is possible with any resident of the Ancient Woods.

New Spawning Grounds

The implementation chosen for the Huts and Bushes was very similar to that of the various graveyards in Assignment 1. These extended from the newly renamed *SpawningGround* abstract class, which contained the code needed for any spawning ground: the ability to spawn an enemy, with a certain chance of this each turn. Using this class implemented the Open/Closed Principle, extending from the current system, without requiring modification (beyond the name) of the class.