**REQ2: Inhabitant of the Stormveil Castle**

The diagram represents an object-oriented design for requirement 2 of assignment 3 and displays the necessary classes and their relationships with each other in order to fulfil the needs of the overall system. These new classes make use of pre existing code from the engine and game, allowing us to easily extend our system.

Since this requirement is focused on adding new enemies and spawners to the game to populate the Stormveil Castle map, a multitude of new classes, which inherit from the abstract classes developed in Assignment 2, were created. This highlights the highly extensible design we have created and allows us to minimise our repeated code easily.

The first two classes we implemented were the Barracks and Cage classes, respectively. These two classes extend the previously created Spawner abstract class and hence inherit all necessary methods and attributes of the Ground and Spawner classes, respectively. In doing so, we simply need to implement the spawnEnemy abstract method in each class to spawn their associated enemy (GodrickSoldier and Dog). These design choices allow us to fulfil DRY and open closed principles all at once, as we not only maximise code reuse by extending our previous classes to implement new features but also avoid modifying our old classes. This also fulfils the Liskov substitution principle, as both Barracks and Cage can be used as generic Spawner or Ground objects. An alternative to this approach would have been to make Barracks and Cage extend Ground directly, but this does not make sense as we would have to rewrite all our code to spawn enemies again, which is readily available in the Spawner class.

The next two classes we will be discussing are the Dog and GodrickSoldier classes, respectively. These two classes extend our previously defined Enemy class, allowing them to benefit from all of the previous logic and behaviour systems that we have implemented in Assignment 2. In this way, we avoid repeating code unnecessarily, as we simply need to input the correct stats into each of the class constructors and create a new GODRICK species to prevent them from attacking each other. Once again, this approach fulfils the DRY, open-closed, and LSP principles nicely as we have extended a previously designed abstract class in such a way that Dog and GodrickSoldier can take the place of any Enemy or Actor in the system. The only downside to this approach is that we have to ensure that both Dog and GodrickSoldier are given the GODRICK capability in their constructors. This can be avoided by simply creating another class called GodrickEnemy which extends enemy and adds said capability in its constructor, then forcing Dog and GodrickSoldier to inherit from it. However, this approach involves creating an entirely new class just to reduce two lines of code into one, which is a pointless optimisation as there is no other shared logic between these two classes.

Finally, we have added an additional weapon called HeavyCrossbow which is used by the GodrickSoldier. This weapon extends the TradableWeapon class, allowing it to be purchased and sold by merchants without needing to repeat the same code in the class itself. This weapon has been implemented as a melee weapon as we have opted not to implement the

optional ranged functionality, and hence it behaves exactly like any other weapon with no special skills, such as the Club. By extending the TradableWeapon class, we once again achieve the DRY, open-closed, and LSP principles in this implementation, allowing us to trade this weapon item with ease while maintaining its usability as both a WeaponItem and Weapon within the system.