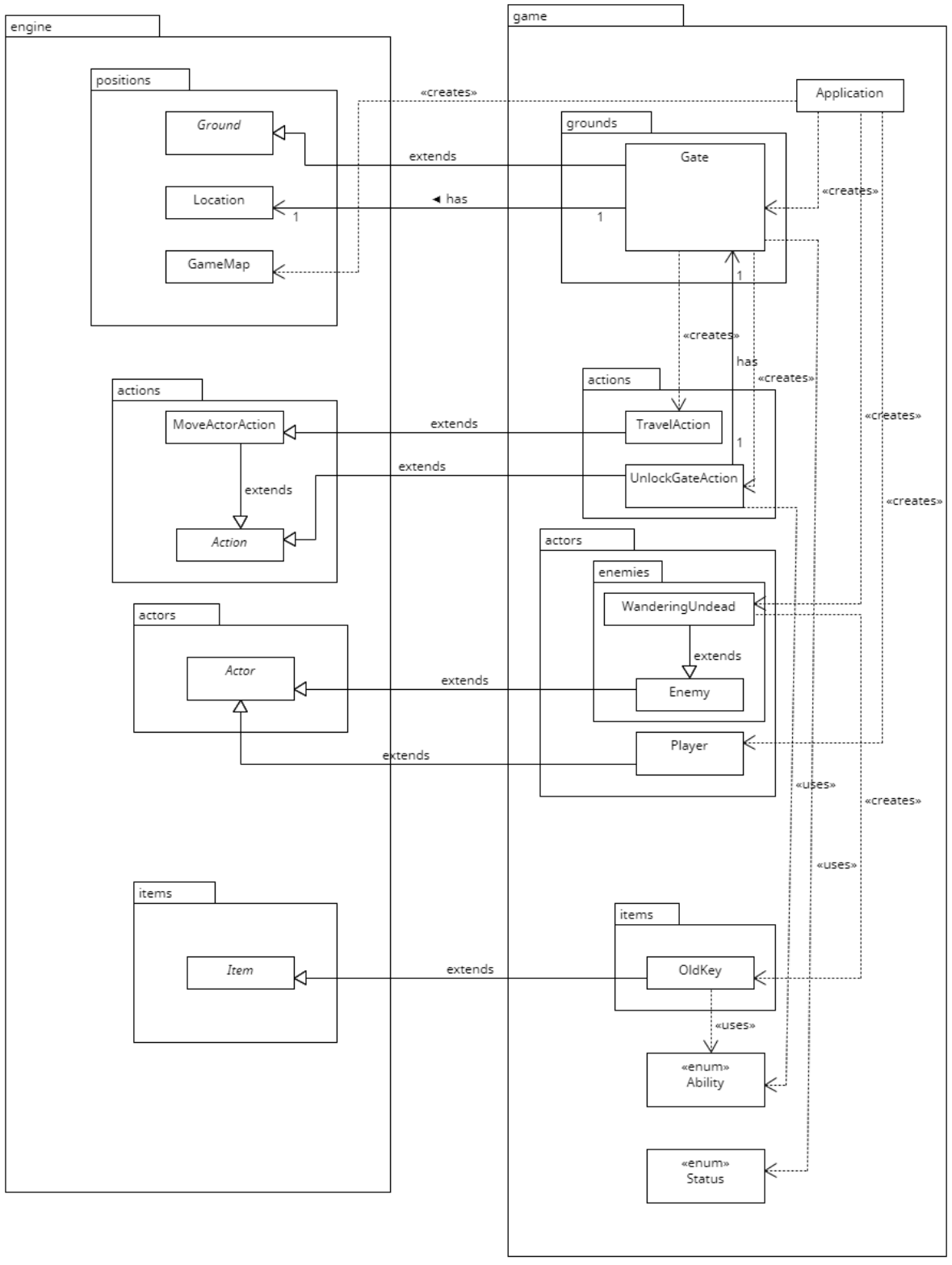
REQ4 UML Diagram.



The primary objectives of REQ4 are to introduce a new game map called the Burial Ground, a gate for travelling to this map, and an OldKey to unlock the gate, which may be dropped by WanderingUndead.

To fulfil this, I made an OldKey class that extends the Item abstract class from the engine. It also has a dependency on the Ability enum class, this allows the Gate class to check if an Actor possesses an instance of OldKey without needing to rely on techniques like the ‘instanceof’ operator or checking the class name. This is a positive aspect of adhering to LSP because it means that OldKey can be used interchangeably with other Item subclasses for methods like Pick up and Drop, without breaking the expected behaviour of the Gate class.

I made an UnlockGateAction class which extends the Action abstract class and handles unlock gate actions in the game. It has an association with the Gate class and can override the gate's status when the gate is successfully opened. This design allows for the addition of new unlock-related features in later implementations while following the Single Responsibility Principle (SRP).

The Gate class extends the Ground abstract class and thus inherits relevant methods from it. It returns a TravelAction class, which extends the MoveActorAction class. I did this because TravelAction shares common functionality with MoveActorAction, which eliminates the need to repeat code (DRY). This action handles Actor teleportation, and provides a named destination instead of just a direction for better user understanding.

TravelAction maintains the ability to be used interchangeably with its base class MoveActorAction, aligning with the Liskov Substitution Principle (LSP) by not altering the core behaviour expected from such action classes. This adherence to LSP ensures that using TravelAction as a substitute for MoveActorAction doesn't introduce unexpected issues or behaviour changes down the line.

Alternative Implementation:

Just reuse MoveActorAction to handle the transport of actors to a new map. This reduces the need of adding a whole new class just for the benefit of showing the destination to the user. However, it will be more confusing to the user, they will not be shown the map name that they will be taken to.

Advantages:

Future extensibility

This implementation allows for the addition of new features related to unlocking and travelling without extensive modifications to existing code, which promotes extensibility and scalability.

High Cohesion

The use of separate classes like OldKey, UnlockGateAction, and TravelAction for handling specific aspects of the game (key, gate unlocking, and travelling) results in a well-organised codebase. Each class has a clear responsibility, which makes it easier to maintain and understand. Each of these classes exhibits a strong sense of cohesion because they are focused on a single, well-defined task.

Disadvantages:

Performance Overhead

Abstraction and modularity often come with a slight performance overhead. This overhead could impact the game's performance as the amount of different elements in the game grows.

Code Bloat

This may lead to code bloat if too many classes with minimal functionality are created. This can result in a larger codebase than necessary, potentially making the game harder to maintain.